UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

ESTUDIO Y ANÁLISIS DE PROTOCOLOS DE FIRMAS DIGITALES ANÓNIMAS CON ORIENTACIÓN A "SMARTCARDS"

Autor: Álvaro Marcos Escalona
Tutor: Francisco de Borja Rodríguez Ortiz

Junio 2017

ESTUDIO Y ANÁLISIS DE PROTOCOLOS DE FIRMAS DIGITALES ANÓNIMAS CON ORIENTACIÓN A "SMARTCARDS"

Autor: Álvaro Marcos Escalona
Tutor: Francisco de Borja Rodríguez Ortiz

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

Resumen

Hasta la aparición de la criptografía de clave pública los métodos de cifrado y descifrado utilizaban una clave simétrica idéntica de forma que un receptor no podría saber con seguridad la identidad del emisor del mensaje si hubiese más de dos posibles (que tuvieran acceso a la clave simétrica). El concepto de criptografía asimétrica o criptografía de clave pública permite solucionar dicho problema. En los criptosistemas de clave pública existen dos tipos de clave diferentes, la privada y la pública, permitiendo cifrar un mensaje con la clave pública del receptor para que solamente éste pueda descifrarlo con su clave privada. Además, el emisor puede cifrar previamente el mensaje (o una representación de él como un MAC o hash) con su clave privada de forma que una vez el receptor ha usado su clave privada para descifrar, puede usar la clave pública del emisor para comprobar que efectivamente el mensaje proviene de dicho emisor. Esta propiedad se denomina firma digital y actualmente es ampliamente utilizada en multitud de servicios de Internet.

Uno de los problemas actuales es la gran cantidad de datos que obtienen ciertas aplicaciones de los usuarios cuando se hace uso de ellas. En ocasiones puede ser deseable o necesario preservar el anonimato de las personas que usan cierto servicio asegurando que dichas personas pertenecen a un grupo de usuarios válidos. Esta idea puede llevarse a cabo mediante la firma grupal, que es una evolución de las firmas digitales que permite producir mensajes en nombre de un grupo de usuarios. Este tipo de firmas tiene numerosas aplicaciones prácticas como anonimizar conexiones de red, controlar el acceso de miembros de un grupo a recursos o lugares, tomar acuerdos de forma colegiada entre diferentes organizaciones o voto electrónico, entre otras.

Por otro lado, una tarjeta inteligente (en adelante smartcard) es una tarjeta de un tamaño (en general) similar a una tarjeta de crédito o más pequeña, con un circuito integrado que ofrece ciertos servicios previamente programados. Las smartcards surgieron en los años 60 y han evolucionado continuamente desde entonces, permitiendo incluir módulos hardware especializados para la ejecución de algoritmos de cifrado y firma digital. Además, las smartcards son unos dispositivos muy útiles a la hora de almacenar datos importantes dentro de ellas. Esto se debe al hecho de que el propietario tiene el control de la tarjeta a la hora de utilizarla dentro de una aplicación o servicio que requiera dicha información y a que puede proteger dicha información con un número PIN para restringir su acceso.

En este trabajo se profundizará en los conceptos de firma grupal, privacidad y anonimia, así como en la evolución de las smartcards y sus características. También se mencionarán varios artículos que proponen soluciones para obtener anonimia y privacidad utilizando smartcards. Posteriormente, se propondrán varios protocolos que combinan las firmas grupales y las smartcads dentro de la misma aplicación o servicio. Estos protocolos están diseñados para ofrecer a los clientes de una aplicación o servicio privacidad en sus comunicaciones, así como diferentes niveles de anonimato en función del protocolo y de las necesidades del servicio. De esta manera se demostrará que se puede combinar de forma práctica y útil las firmas grupales con el control de la información que permite una smartcard. Se discutirá también la seguridad de dichos protocolos indicando las suposiciones realizadas en cada momento y se introducirán las tecnologías utilizadas para el desarrollo de los protocolos propuestos.

Palabras Clave

smartcard, javacard, firma digital, firma grupal, criptografía de clave pública, criptografía de clave privada, firma grupal revocable, firma grupal corta, firma trazable, firma ciega, prueba de conocimiento cero, seguridad, privacidad, anonimidad, anonimato, desvinculación, anonimato justo, revocabilidad.

Abstract

Until the birth of public key cryptography, the encryption and decryption methods used an identical symmetric key so that a receiver could not know for sure the identity of the sender of the message if there were more than two possible (who had access to the symmetric key). The concept of asymmetric cryptography or public key cryptography solves this problem. In public key cryptosystems, there are two different types of keys, the private and the public one, allowing a message to be encrypted with the public key of the receiver so that he/she is the only one who can decrypt it with his/her private key. In addition, the issuer can pre-encrypt the message (or a representation of it like a MAC or hash) with his/her private key so that once the receiver has used his/her own private key to decrypt, he/she can use the public key of the issuer to check that the message actually comes from the issuer. This property is called digital signature and is currently widely used in many Internet services.

One of the current problems is the large amount of data that certain applications obtain from users when they are using them. Sometimes it may be desirable or necessary to preserve the anonymity of people using a certain service by ensuring that those people belong to a valid group of users. This idea can be performed by means of the group signature, which is an evolution of the digital signatures that allows to produce messages on behalf of a group of users. This type of signatures has numerous practical applications such as anonymizing network connections, controlling the access of members of a group to certain resources or places, making collegiate agreements between different organizations or electronic voting, among others.

On the other hand, a smart card (hereinafter smartcard) is a card of a size (generally) similar to a credit card or smaller, with an integrated circuit that offers certain services previously programmed. Smartcards appeared in the 1960s and have evolved continuously since then, allowing to include specialized hardware modules for the execution of encryption algorithms and digital signature inside them. In addition, smartcards are very useful devices when storing important data inside them. This is due to the fact that the owner has the control of the card when using it within an application or service that requires such information and that you can protect that information with a PIN number to restrict the access to it.

In this document, the concepts of group signature, privacy and anonymity, as well as the evolution of smartcards and their characteristics will be explored. Several articles that propose solutions to obtain anonymity and privacy using smartcards will be also mentioned. Subsequently, several protocols will be proposed combining group signatures and smartcads within the same application or service. These protocols are designed to offer to the customers of an application or service privacy in their communications, as well as different levels of anonymity depending on the protocol and the needs of the service. In this way it will be shown that group signatures can be combined in a practical and useful way with the control of the information that a smartcard allows. The security of these protocols will also be discussed, indicating the assumptions made at each moment and the technologies used for the development of the proposed protocols.

Key words

smartcard, javacard, digital signature, group signature, public key cryptography, private key cryptography, revocable group signature, short group signature, traceable signature, blind signature, zero knowledge proof, security, privacy, anonymity, unlinkability, fair anonymity, revocability.

Agradecimientos

En primer lugar, agradecer a mi familia el apoyo recibido en las decisiones que he tomado durante la carrera y, en general, durante toda mi vida. También, agradecer a mi tutor su consejo durante este trabajo que tan interesante me ha resultado y a Jesús Díaz que me permitiese utilizar su librería de firmas durante su realización.

Querría también agradecer a todos los profesores que he tenido durante estos años todo lo que he podido aprender de ellos.

Merecen una mención especial mis fantásticos compañeros de carrera que han hecho que estos años pasasen volando.

Por último, agradecer a Alba que estos tres últimos años hayan sido los mejores de mi vida.

Índice general

Índice de figuras

Índice de tablas

Glosario

- **ZKP**: Zero Knowledge Proof. Prueba de Conocimiento Cero.
- **RSA**: Rivest Shamir Adelman. Uno de los criptosistemas de clave asimétrica más conocidos.
- **APDU**: Application Protocol Data Unit.
- **AES**: Advanced Encryption Standard. Uno de los criptosistemas de clave simétrica más conocidos.
- **CBC**: Cipher Block Chaining.
- **ECB**: Electronic Codebook.
- **PKI**: Public Key Infraestructure.
- **DDH**: Decisional Diffie Hellman.
- **LDH**: Linear Diffie Hellman.
- **q-SDH**: q-Strong Diffie Hellman.
- **XOR**: operación OR exclusiva.
- **PKCS**: Publick Key Cryptography Standards. Grupo de estándares de criptografía de clave pública publicados por los laboratorios RSA Laboratories.
- **OAEP**: Optimal Asymmetric Encrytion Padding. Esquema de relleno utilizado habitualmente al encriptar con RSA.
- **Trapdoor**: función trampilla.
- **Smartcard**: tarjeta con circuitos integrados que permite ejecutar cierta lógica programada.
- **USB**: Universal Serial Bus.
- **Función de un solo sentido**: tipo de función que es difícil de invertir.
- **Función hash**: tipo de función que dado un valor de entrada de longitud variable produce un código de longitud fija.
- **CPU:** Central Processing Unit. Hardware dentro de un ordenador o dispositivo programable encargado de interpretar las instrucciones de un programa informático mediante operaciones aritméticas, lógicas y de entrada/salida del sistema.
- **NPU**: Numerical Processing Unit. Co-procesador encargado de ayudar a la CPU a la hora de realizar operaciones numéricas, especialmente durante procedimientos criptográficos.
- **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.
- **Algoritmo determinista**: algoritmo que cada vez que se resuelve el mismo problema con las mismas entradas lo resuelve con mismo resultado y ejecución.
- **Método de resolución no determinista**: algoritmo que cada vez que se resuelve el mismo problema con la misma entrada puede variar el resultado y la ejecución.
- **MCD**: máximo común divisor.
- **DNI**: Documento Nacional de Identidad.
- **PIN**: Personal Identification Number. Número Personal de Identificación.
- **ISO/IEC**: International Organization for Standardization/International Electrotechnical Commission. Organizacion Internacional para la Estandarización/Comisión Internacional Electrotécnica.
- **OECD**: Organisation for Economic Co-operation and Development. Organización para la Cooperación Económica y el Desarrollo.
- **CEN**: Comite Européen de Normalisation. Comité Europeo de Normalización.
- **ETSI**: European Telecommunications Standards Institute. Instituto Europeo de Estándares de Comunicación.

1. Introducción

En este trabajo se introduce brevemente el origen y evolución de las smartcards así como su clasificación según sus características. Posteriormente se menciona la importancia de seguir un proceso de estandarización dentro de las smartcards y las organizaciones encargadas de ello. Llegados a este punto se explicarán los conceptos necesarios para introducir la firma digital y su evolución, la firma grupal, que supone un gran avance en cuanto a la protección del anonimato de los usuarios. Con el fin de demostrar las ventajas que puede suponer la combinación de las firmas grupales y las smartcards, se han diseñado e implementado varios protocolos que proporcionarán a los usuarios privacidad y diferentes niveles de control del anonimato y se ha discutido la seguridad de estos protocolos.

1.1. Motivación

Desde su creación, el uso de las firmas digitales se ha extendido rápidamente en Internet ya que permite identificar al autor de un determinado mensaje. La firma digital permite, por ejemplo, operar en bolsa en la red identificando los autores de los mensajes tanto para los inversores como para los agentes de bolsa. De este modo, los agentes no podrán falsificar una operación arriesgada que ha resultado en pérdidas como emitida por un inversor, ni los inversores podrán negar haber solicitado una operación que haya sido firmada por ellos. En definitiva, la firma digital ofrece una prueba no falsificable que vincula un mensaje con su autor. Los primeros trabajos referentes a firmas digitales surgen a partir de la década de los 70 y guardan una estrecha relación con los criptosistemas de clave pública **[15, 21]**. La firma digital puede utilizarse junto con un método de cifrado para asegurar que un mensaje solo es conocido por el emisor y el receptor y permitir identificar el autor de dicho mensaje.

Actualmente existe un gran número de servicios y aplicaciones en Internet en los que se almacenan, gestionan y utilizan datos de sus usuarios. En muchos casos, puede ser deseable por el tipo de servicio (un foro de preguntas entre alumnos y profesores o un repositorio de información confidencial) o necesario por exigencia legal (como los datos médicos **[31]**) que la identidad de los usuarios sea anónima. Esto implica que ni el receptor del mensaje ni un observador externo puedan ser capaces de determinar el autor de un mensaje. A primera vista, puede parecer que el concepto de identificar al autor de un mensaje con la firma digital se enfrenta al concepto de anonimia, pero a través de las firmas grupales se puede encontrar solución a este problema. Con las firmas grupales se puede identificar un mensaje como emitido por un usuario perteneciente a un grupo de usuarios válidos permaneciendo la identidad del autor oculta dentro del grupo.

Por otro lado, la facilidad de producción y su bajo coste, junto con los avances en la capacidad y rendimiento de sus componentes, hacen de las smartcards unos dispositivos atractivos para la implementación de numerosas aplicaciones y servicios. Gracias a los mecanismos de protección de información con los que cuentan las smartcards y al componente extra de seguridad que supone poder llevarla físicamente, un usuario puede controlar cómo y cuándo se accede a ella durante el uso del servicio. Todas estas razones hacen de las smartcards un dispositivo atractivo para almacenar datos que permitan identificar a un usuario dentro de un servicio o aplicación. En este punto resulta muy interesante plantearse el diseño de varios sistemas que, gracias al uso de las

firmas grupales y las smartcards, proporcionen a sus usuarios diferentes niveles de anonimato en función de las características del servicio que se proporciona.

1.2. Objetivos

El objetivo principal de este trabajo es el de estudiar y analizar la posibilidad de combinar en un protocolo un esquema de firmas anónimas con una smartcard, de forma que el servicio que lo implemente ofrezca a sus usuarios privacidad y diferentes niveles de anonimato (según las características del protocolo). Para ello se han llevado a cabo los siguientes hitos:

- Estudiar tanto las características y limitaciones generales de las smartcards como la manera de transmitir información a una smartcard y de obtenerla de ella.
- Estudiar y analizar diferentes esquemas de firma grupal (viendo las bases matemáticas en las que se sustentan y las propiedades de cada uno de ellos). Se han estudiado especialmente los esquemas de la librería de firmas **libgroupsig [17]** junto con los programas de dicha librería que implementan las diferentes funcionalidades que ofrecen los esquemas de firmas.
- Implementar una prueba de concepto basada en los protocolos diseñados que demuestre la posibilidad de incluir firmas grupales y smartcards dentro del mismo servicio.
- Simular el comportamiento de nuestra smartcard con las implementaciones realizadas y obtener medidas de tiempo para asegurar la usabilidad de los sistemas implementados.

1.3. Estructura del documento

Con el fin de alcanzar los objetivos propuestos se ha desarrollado la siguiente memoria que está organizada como se indica:

- **Estado del arte**: En este capítulo se explica qué son las smartcards, sus orígenes y desarrollo. También se explican los conceptos de privacidad, anonimia, desvinculación y anonimato justo motivando su importancia. Posteriormente se introducen las pruebas de conocimiento cero como método para demostrar cierto conocimiento sin revelarlo y se relaciona este concepto y los anteriores con la firma grupal (que es una evolución de la firma digital como se verá). Por último, se reflexiona sobre los problemas actuales de privacidad en smartcards y sus soluciones, incluyendo al final un breve resumen del nuevo reglamento de la Unión Europea que regulará el tratamiento de datos debido a su relación con este trabajo.
- **Análisis y diseño de los protocolos propuestos**: En este capítulo se detalla el diseño de tres protocolos con el fin de proporcionar a los usuarios del sistema que los implemente diferentes niveles de anonimato. Se podrá observar que cada protocolo incrementa la calidad del anonimato para los usuarios con respecto al anterior. También se discutirá la seguridad de dichos protocolos en determinadas circunstancias y se explicarán las herramientas y tecnologías utilizadas para el desarrollo de los anteriores protocolos.
- **Pruebas y resultados**: Pruebas de rendimiento de la smartcard para diferentes comandos y referencia al manual del usuario que describe como probar la funcionalidad de los protocolos.
- **Conclusiones y trabajo futuro**: En esta sección se describen las conclusiones obtenidas de la realización de este trabajo y posibles retos relacionados para llevar a cabo en un futuro.
- **Anexos**: Información detallada o ampliada que permite llevar a cabo los hitos descritos en el apartado anterior y entender en profundidad el funcionamiento e implementación de los protocolos.

2. Estado del arte

En este capítulo se recopilan el actual estado de desarrollo, las bases teóricas de las tecnologías y los resultados utilizados en el trabajo.

2.1. Smartcards

Las tarjetas inteligentes o smartcards surgieron en 1968 gracias a los alemanes Jürgen Dethloff y Helmut Grötrupp a partir de la idea de incorporar un circuito integrado dentro de una tarjeta de identificación. En 1974 a partir de la patente para smartcards publicada por el francés Roland Moreno la industria de los semiconductores es capaz de producir los circuitos integrados necesarios para esta tecnología a un precio aceptable. Desde entonces, se ha incrementado su capacidad de almacenamiento y procesamiento y se ha popularizado su uso en diversas aplicaciones como el pago del teléfono público, las tarjetas SIM, tarjetas de pago, identificación y transporte de información. A partir del tipo de funcionalidad para el que han sido creadas podemos dividir las smartcards en dos grandes grupos (**[6]** Cap. 2):

1. **Tarjetas de memoria**: en este grupo se engloban las tarjetas destinadas a almacenar datos. Su manufactura suele tener un bajo coste y pueden contener una lógica integrada que permite la protección de ciertos datos contra la manipulación. Se usan normalmente en sistemas de identificación o de tarjetas pre-pago donde el coste es un factor importante a tener en cuenta.
2. **Tarjetas microprocesadoras**: Fueron usadas por primera vez en Francia como tarjetas bancarias y contienen un microprocesador que se puede programar libremente. La funcionalidad estará restringida por tanto al espacio de almacenamiento disponible y a la capacidad de dicho procesador. Es importante mencionar que en cada generación se incrementa la capacidad de procesamiento y almacenamiento de los circuitos integrados que utilizan este tipo de tarjetas. Las posibles aplicaciones para las tarjetas microprocesadoras incluyen el control de acceso a equipos y áreas restringidas, firmas digitales, compras electrónicas y almacenamiento seguro de datos, entre otras. También es posible tener una tarjeta microprocesadora con distintas aplicaciones que sirva para varias tareas. Normalmente las tarjetas de este tipo cuya aplicación requiere el uso de algoritmos criptográficos, hacen uso de un coprocesador criptográfico para reducir en gran medida el tiempo de las operaciones realizadas.

También se puede dividir el conjunto de smartcards en base a la forma de transmisión de los datos (ver **Figura 2.1.**). En este caso se pueden dividir en tres grupos dependiendo de si necesitan un contacto eléctrico entre la tarjeta y el terminal, no lo necesitan ("contactless smartcards") o pueden utilizar ambas opciones para comunicarse con el terminal ("combicards"). Las tarjetas que no necesitan dicho contacto normalmente pueden estar a una distancia de entre unos centímetros hasta pocos metros del terminal y su principal ventaja es facilitar el manejo y la identificación rápida. En el marco de este trabajo, dado que la seguridad es un factor muy importante, se ha preferido considerar el uso de una tarjeta de contacto para evitar la posible acción de un oponente en el caso de que la tarjeta no requiriese contacto.
En lo referente a la transmisión de datos entre el lector y la tarjeta existen principalmente dos protocolos (**T = 0** y **T = 1**; **[18]** Cap. 10 y Cap. 11 respectivamente). El protocolo **T = 0** fue el primer protocolo estándar utilizado en smartcards y su diseño permite transmitir información a nivel de byte. Por otro lado, el protocolo **T = 1** está diseñado para transmitir información a nivel de bloque entre la smartcard y el lector. En un futuro es probable que se incluya un protocolo que permita transmitir datos mediante USB. Este protocolo requiere de un hardware específico dentro

del microprocesador de la tarjeta, pero actualmente está presente en gran parte de ellos. Las ventajas que presenta frente a los dos anteriores son su mayor capacidad de transmisión de datos y que actualmente ya es estándar dentro de la industria de los ordenadores. La transmisión efectiva de información entre la smartcard y el lector de tarjetas se realiza mediante unidades de comunicación denominadas **APDUs** (Application Protocol Data Unit, cuya estructura y detalles se pueden consultar en el **Anexo N**).

Por último, podemos observar en la **Figura 2.2.** la arquitectura básica del microprocesador de una smartcard de contacto con un coprocesador. En la memoria ROM de las tarjetas se guardarán las instrucciones del sistema operativo de la smartcard que se ejecutarán en la CPU y normalmente tiene un tamaño de 16 a 400 KB (en este caso además de la CPU se dispone de la NPU para optimizar las operaciones numéricas). La memoria EEPROM contendrán extensiones al sistema operativo y las aplicaciones que se hayan instalado dentro de la smartcard con sus datos asociados (su capacidad suele oscilar entre 1 y 500 KB). La memoria RAM tiene una capacidad que suele oscilar entre 256 bytes y 16 KB. Se puede comprobar que hay diferencias notables en las capacidades de las smartcards dependiendo del propósito para el que han sido diseñadas. A pesar de ello, es importante tener en cuenta la gestión de los recursos de la tarjeta y en especial de las diferentes memorias a la hora de programar una aplicación para smartcards.

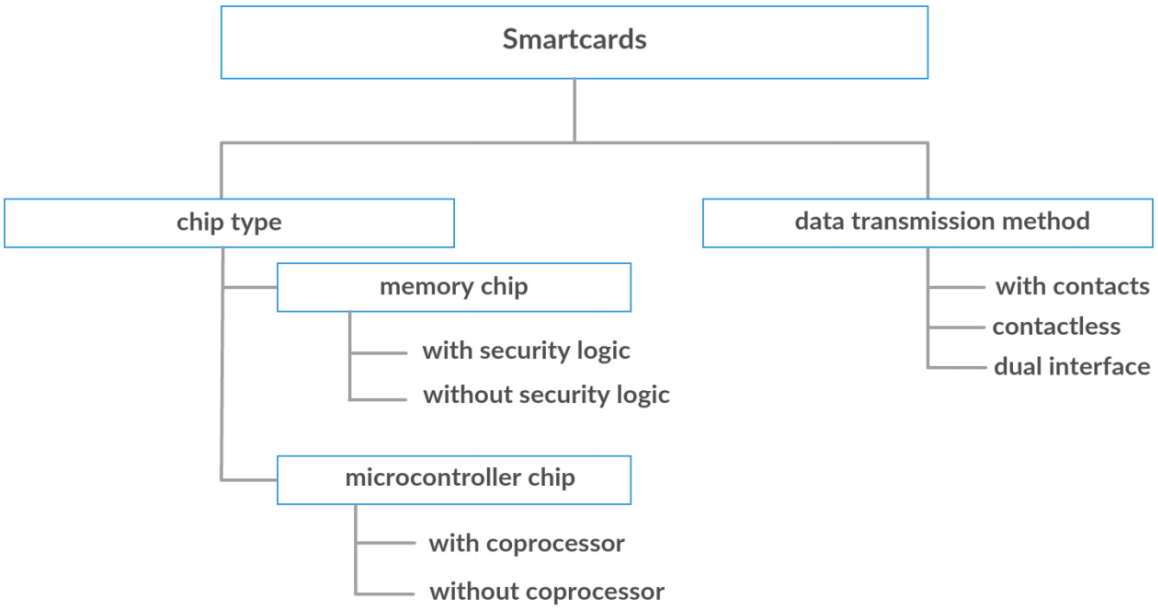

Figura 2.1. Clasificación de tarjetas con chip en base al tipo de chip y al método de transmisión de datos utilizado.

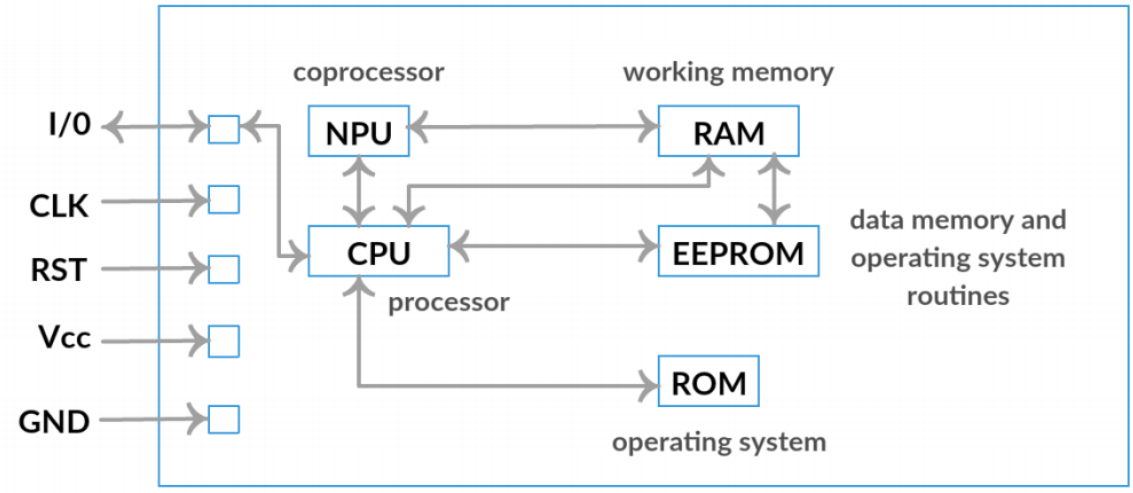

Figura 2.2. Arquitectura básica del microprocesador de una smartcard de contacto con un co-procesador.

2.1.1. La estandarización de las smartcards

Un requisito previo al uso de las smartcards en todo el mundo es la existencia de unos estándares nacionales e internacionales que las regulen. Las smartcards normalmente son un componente de un sistema complejo, lo que implica que las interfaces de comunicación entre la smartcard y el resto del sistema deben estar especificadas e implementadas de forma precisa. En el caso de que este proceso se llevase a cabo de forma individual, nos encontraríamos con que sería necesario un tipo distinto de smartcard para prácticamente cada sistema. Para definir correctamente el concepto de estándar se ha considerado la definición proporcionada por la **ISO/IEC**:

Estándar: aquél documento producido por consenso y adoptado por una organización reconocida, el cual define reglas, directrices o características para ciertas actividades o sus resultados con el objetivo de lograr un grado óptimo de regulación en un contexto dado.

Nota: Los estándares deben basarse en resultados establecidos por la ciencia, la tecnología y la experiencia con el objetivo de promocionar una serie de beneficios optimizados a la sociedad.

El estándar **ISO/IEC 7816**, en particular las partes 3 y 4 **[18, 19]**, es especialmente importante para este trabajo ya que especifica los protocolos y comandos de transmisión de información. En el caso de Europa la organización encargada de coordinar con la **ISO** la estandarización de las smartcards es la **CEN**. También es importante mencionar en el caso de Europa a la **ETSI** debido a su contribución en la extensión del uso internacional de las smartcards. A continuación, se muestra una imagen (**Figura 2.3.**) con los grupos de trabajo de la **ISO/IEC** y los estándares de los que son responsables.

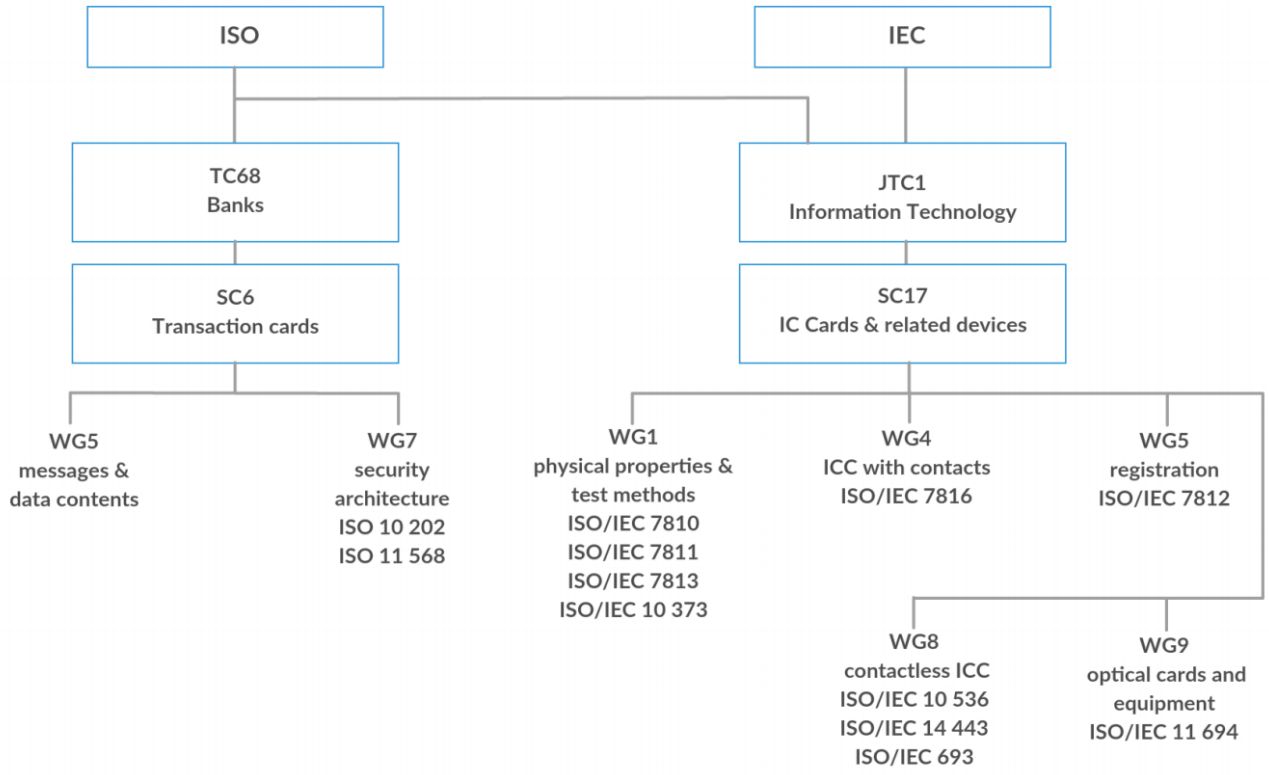

Figura 2.3. Organización y grupos de trabajo para la estandarización internacional de las smartcards.

2.2. Privacidad, anonimato, desvinculación y anonimato justo

Como se ha mencionado antes en este documento, existen productos, sistemas o aplicaciones informáticas en los que puede ser deseable o necesario (debido a que una ley así lo indique; ver apartado **2.6.1.**) que un usuario cuente con la capacidad de controlar con quién comparte su información dentro de una aplicación. La privacidad suele ser un requisito deseable ya que implica que el contenido de la comunicación sólo es conocido por el emisor y los receptores indicados. El anonimato permite que no se relacione información o datos de la aplicación con la identidad de un usuario. En el siguiente apartado se profundiza en estos conceptos y se introduce dos nuevos relacionados: la desvinculación y el anonimato justo.

2.2.1. Privacidad (Privacy)

La privacidad consiste en que una persona, grupo u organización puedan decidir cuándo, cómo y en qué cantidad, información acerca de ellos es revelada a otras personas **[4]**. Una manera de conseguir privacidad es minimizando la cantidad de información que se comparte y el tiempo que esta información compartida está disponible. Como se ha mencionado en la introducción de este apartado, la privacidad garantiza que la información de un mensaje perteneciente a la comunicación entre varias partes (por ejemplo, Alice y Bob) solo pueda ser conocida por dichas partes (salvo que una de ellas revele dicha información o un oponente averigüe la forma de descifrar los mensajes). Este problema se ha intentado resolver de múltiples formas desde la Historia Antigua con métodos como el criptosistema del César que han ido evolucionando

constantemente hasta la criptografía moderna (**AES [14]**, **RSA [15]**). Recordemos que en el resumen se comentó que la clave que utilizan estos criptosistemas puede ser simétrica o asimétrica. Existe una aproximación intermedia que consiste en usar un protocolo de clave asimétrica (**RSA**, **Diffie-Hellman** o **ElGamal** cuyas particularidades se pueden consultar en **[7]**) para intercambiarse un "secreto compartido" que funcionará como clave en un futuro protocolo de clave simétrica para intercambiar información. Estos protocolos de clave asimétrica (pública) consisten de forma general en lo siguiente:
Cada usuario del sistema tiene dos claves, una privada (que sólo conoce dicho usuario) y una pública (accesible por todo el mundo). A la hora de comunicarse dos usuarios, como por ejemplo Alice y Bob, Alice cifrará el mensaje para Bob con la clave pública de Bob y solamente él podrá descifrarlo con su propia clave privada y equivalentemente a la inversa.

Es importante mencionar que estos protocolos también permiten la posibilidad de firmar los mensajes dado que Alice puede cifrar primero el mensaje con su propia clave privada, y el resultado cifrarlo con la pública de Bob. De esta forma Bob descifrará primero con su clave privada y el resultado lo descifrará de nuevo usando la clave pública de Alice, asegurándose así de que sólo ella pudo haber enviado ese mensaje (pues nadie más conoce la clave privada de Alice). Para asegurar correspondencia entre las claves públicas y las identidades (personas, equipos, organizaciones, etc.) que están detrás de ellas existen unas empresas denominadas autoridades certificadoras (**Anexo E**).

2.2.2. Anonimato/Anonimia (Anonymity)

Utilizando los mismos actores que en el apartado anterior, el anonimato supone que Alice puede enviarle un mensaje a Bob sin que Bob (ni otro usuario del sistema o externo) sepa que el mensaje proviene de ella. Para que sea posible que exista anonimato, es lógico pensar en la necesidad de que exista un conjunto de posibles autores del mensaje destinado a Bob y que todos puedan producir mensajes de forma anónima como Alice, es decir, un **conjunto de anonimia [4]**.

En caso de fijarnos en la anonimia para todo el **conjunto de anonimia** y no para un usuario en particular, se considera que el sistema posee **anonimia global máxima** si dada una acción en el sistema (por ejemplo, enviar un mensaje), todos los usuarios del conjunto de anonimia tienen las mismas probabilidades de haber realizado dicha acción ($1/K$ en un grupo de K usuarios por ejemplo). Dado que los usuarios del sistema no tienen por qué actuar del mismo modo, el concepto de **anonimia global máxima** es difícil de conseguir en la práctica.

A la hora de estudiar la anonimia de un sistema, se define la **calidad de la anonimia** como el grado de anonimia que el sistema proporciona a los usuarios así como la estabilidad de dicha anonimia frente a cambios en el sistema. Desde el punto de vista de un atacante, puede haber una diferencia entre la anonimia de un usuario a priori y después de que dicho atacante haya observado el sistema (**delta de anonimia**). Asumiendo que un atacante no olvida la información que haya obtenido del sistema anteriormente, la anonimia de un sistema no puede crecer para un momento pasado por lo que, esta delta nunca podrá ser positiva. Por ejemplo, si en un sistema anónimo con $\boldsymbol{n}$ usuarios, un oponente observase una acción $\boldsymbol{A}$ que pudiese atribuirse a cualquiera de ellos con probabilidad $\boldsymbol{1/n}$, añadiésemos otros $\boldsymbol{n}$ usuarios posteriormente ($\boldsymbol{2n}$ en total), la acción $\boldsymbol{A}$ seguiría siendo anónima solamente entre un grupo de $\boldsymbol{n}$ usuarios (la **delta de anonimia** permanece constante para dicha acción). Es más, si el atacante pudiese relacionar futuras acciones del mismo usuario con $\boldsymbol{A}$ (mediante el momento del día, patrones o algunas características de los mensajes) podría reducirse el **conjunto de anonimia** para la acción $\boldsymbol{A}$ y las relacionadas a un número inferior a $\boldsymbol{n}$ siendo la **delta de anonimia** negativa.

2.2.3. Desvinculación (Unlinkability)

La **desvinculación** es una propiedad importante que afecta a la **calidad de la anonimia** de los sistemas y que implica que un atacante de un sistema concreto no pueda relacionar varias acciones de un mismo usuario dentro del sistema con dicho usuario. Por ejemplo, que un usuario solicite el acceso a varios recursos de una biblioteca virtual sin que un atacante sepa que esas peticiones las ha realizado el mismo usuario. Del mismo modo que con la anonimia, se puede establecer una **delta de desvinculación** para medir la diferencia en el nivel de desvinculación de un sistema para un atacante a priori y tras haber realizado observaciones e interacciones con el sistema. Aplicando la misma suposición que antes, esta delta nunca puede ser positiva. En caso de que sea cero se considera que el sistema preserva el nivel de desvinculación con respecto al conjunto de acciones que se tenga en cuenta. Como se puede observar, un sistema en el que se puedan vincular varias acciones con el mismo usuario reduce el **conjunto de anonimia** para dicho usuario y por tanto la calidad de dicha anonimia.

2.2.4. Anonimato justo (Fair anonymity)

Un sistema con anonimato justo permite generar mensajes anónimos a sus usuarios pero además, permite revocar este anonimato a los usuarios que incumplan ciertas normas de uso preestablecidas (por ejemplo, el deterioro intencionado de unas instalaciones accesibles por el grupo de usuarios o el borrado intencionado de información accesible por todos). Para garantizar el anonimato justo es necesario la existencia de las siguientes propiedades:

- **Trazabilidad**: permite identificar la clave utilizada para firmar un cierto mensaje. Esto permitirá relacionar todos los mensajes firmados con dicha clave (pasados y futuros) lo que puede ser útil en caso de querer revocar la anonimia de un usuario, localizar los mensajes de usuarios sospechosos o en caso de que un usuario desee demostrar que la firma de un mensaje es suya.
- **Revocabilidad**: permite designar una o varias claves para que no puedan volver a ser utilizadas para firmar mensajes en nombre del grupo. De este modo, la trazabilidad es necesaria para poder generar revocabilidad.

El anonimato justo surge de la necesidad de contrarrestar y evitar el mal uso del anonimato. Se pueden encontrar las primeras menciones a este tipo de anonimato en la referencia **[2]** donde se utiliza la firma grupal para obtener este tipo de anonimato. Uno de los problemas que surgió a la hora de implementar el anonimato justo fue encontrar una manera eficiente de revocar la anonimia. En la referencia **[1]** se propuso un esquema cuyas claves públicas tenían un tamaño fijo independientemente del número de miembros del grupo. A pesar de que este esquema permitía revelar el autor de una firma, no permitía trazar a estos usuarios, y por tanto no se podían relacionar todos los mensajes firmados por el individuo revelado sin desvelar la identidad del resto de usuarios. A consecuencia de esto, sería necesario restablecer las claves del resto de miembros para revocar a uno de ellos. Posteriormente, en la referencia **[2]** se propone que una autoridad fuese capaz de revelar una función **trapdoor** que pudiese ser utilizada para **trazar** y relacionar diferentes firmas emitidas por un individuo sin revelar su identidad. La autoridad podría además revelar la identidad del firmante y revocar su anonimia posteriormente. El problema de esta propuesta es la necesidad de confiar solamente en una entidad a la hora de gestionar la anonimia. Una solución a este problema es la propuesta en **[33]**, en la cual a la hora de revocar la anonimia de los usuarios hay una serie de entidades que colaboran entre sí y envían sus respectivos secretos compartidos a otra autoridad que actúa como juez. En caso de que decida penalizar cierta acción de un usuario, podrá combinar los secretos compartidos para obtener la función **trapdoor** correspondiente como en el caso anterior. A pesar de los avances en los

esquemas que implementan anonimato justo, un problema común es la falta de existencia de un estándar para revocar la anonimia. Con respecto a este problema, es de especial interés la referencia **[34]** donde se propone el uso del estándar **X.509 PKI** incluyendo extensiones para los métodos de revocación de identidad actuales que permitan el uso de esquemas de firmas anónimas y justas.

La firma grupal será la principal herramienta que se utilice durante este trabajo a la hora de obtener anonimato. En particular, se hará uso de los esquemas implementados en la librería de firmas **libgroupsig [17]**.

2.3. Pruebas de conocimiento cero (ZKP)

El concepto de "conocimiento cero" fue propuesto en los años 80 por los investigadores Shafi Goldwasser, Silvio Micali y Charles Rackoff **[36]** mientras trabajaban en sistemas interactivos de pruebas en los cuales un probador **P** intenta convencer a un verificador **V** de que conoce la solución a un problema matemático sin desvelar la solución. Evidentemente, dicho problema debe ser lo suficientemente complejo para que un atacante que no posea todas las herramientas para obtener una solución al problema no pueda engañar a **V**. Este concepto surge de la necesidad en aplicaciones reales de que la parte correspondiente a **V** adquiera el mínimo conocimiento posible de información sobre **P [9]**. Una **ZKP** debe cumplir tres propiedades:

- **Completitud**: Si la prueba que proporciona **P** es válida, entonces **V** estará convencido de la validez de la prueba con una probabilidad muy alta.
- **Validez**: Si **P** miente, entonces **V** lo podrá detectar con una probabilidad muy alta.
- **Conocimiento cero**: En caso de que la prueba de **P** sea correcta, **V** no podrá aprender nada acerca de la solución de **P**.

Un ejemplo sencillo de una **ZKP** es el que se describe en **[8]**. Este ejemplo consiste en una cueva circular con una puerta en el medio que separa ambos lados. En este caso el probador **P** entra en la cueva sin que **V** sepa el lado que ha escogido y quiere demostrar a **V** que posee la llave que abre la puerta y puede salir por cualquiera de los dos lados (ver **Figura 2.4.**). Si **V** solicita a **P** que salga por la izquierda hay una probabilidad de ½ de que éste fuera el lado por el que **P** había entrado y pueda salir sin usar la llave. Si repetimos este procedimiento varias veces desde el principio (por ejemplo **n)**, al final la probabilidad de que **P** haya podido engañar a **V** todas las veces y no tenga la llave es de $½^n$, con lo que el verificador puede estar seguro con una alta probabilidad de que **P** conoce la llave.

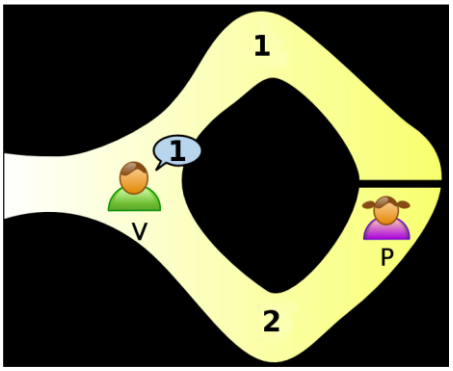

Figura 2.4. Prueba de conocimiento cero de la cueva.

Un ejemplo de prueba de conocimiento cero más complejo se explica en el **Anexo O**. Las **ZKP** pueden ser **interactivas** (como el ejemplo de la cueva) si **V** solicita a **P** la solución para varias entradas. En cambio, serán **no interactivas** si la información inicial enviada por **P** es suficiente para convencer a **V** del conocimiento que debe tener **P**.

Extrapolando el concepto de prueba de conocimiento cero a un sistema informático se podría, por ejemplo, utilizar una función unidireccional (**Anexo D**) para evitar que un servidor guarde las contraseñas de los usuarios que quieran acceder a los servicios que controla. De esta manera los usuarios utilizarían la función unidireccional $\boldsymbol{F(x)}$ para calcular la imagen de sus contraseñas y el servidor comprobaría que la imagen corresponde con el valor que tiene guardado sin conocer las contraseñas de los usuarios. Como se verá más adelante, las pruebas de conocimiento cero son la base de numerosos esquemas de firmas grupales a la hora de demostrar la pertenencia a cierto grupo de forma anónima.

2.4. Firma digital

Aunque ya se ha introducido el concepto previamente en el documento, en este apartado se explica de forma más extendida en qué consiste una firma digital.

La firma digital es uno de los avances más importantes de la criptografía de clave pública ya que proporciona una serie de propiedades de seguridad que, de otro modo, serían difíciles de implementar. Para firmar un mensaje en su nombre, Alice puede usar un algoritmo de firma que reciba como argumentos el mensaje y la clave privada de Alice. Este algoritmo produce un mensaje firmado que es verificable por Bob utilizando la clave pública de Alice. El hecho de que un mensaje esté firmado digitalmente proporciona las siguientes propiedades:

- **Autenticación**: Debe permitir identificar al autor del mensaje y la fecha de éste.
- **No repudio**: El autor del mensaje debe ser incapaz de demostrar que el mensaje no es suyo cuando realmente sea suyo. Además, el autor del mensaje debe ser identificable por terceros en caso de disputa.

Para conseguir estas propiedades la firma debe cumplir una serie de características:

- Debe ser un patrón de bits que dependa del contenido del mensaje.
- Debe utilizar información única perteneciente al emisor del mensaje para prevenir el repudio del mensaje y su falsificación.
- Debe ser relativamente fácil de producir, reconocer y verificar.
- Debe ser computacionalmente costoso falsificar una firma, bien construyendo una nueva para un mensaje ya existente o produciendo otro mensaje para una firma dada.

A partir de estas características, existen diversas formas de implementar la firma digital (utilizar una función hash por ejemplo) y diversos esquemas criptográficos que pueden ser utilizados (**RSA**, **ElGamal**, **Schnorr**) **[7]**.

2.5. Firmas grupales

Las firmas grupales son un tipo de firma digital y fueron propuestas por David Chaun y Eugéne van Heyst en 1991 como un método para conseguir anonimato en aplicaciones y sistemas informáticos **[5]**. A la hora de demostrar la pertenencia al grupo por parte de los miembros y la

validez de los mensajes firmados se utilizan generalmente pruebas de conocimiento cero complejas, evitando de esta manera revelar la identidad de los miembros del grupo.

La mayoría de esquemas de firmas grupales incluyen la figura del administrador del grupo (***GM***) que es el responsable de la creación del grupo y, en ocasiones, también de su gestión. Las dos funcionalidades principales serían la de **firmar** los mensajes en nombre del grupo por parte de sus miembros y la de **verificar** la validez de dichos mensajes por parte del ***GM*** o la autoridad designada para ello. También permiten en caso de disputa que el ***GM*** descubra la identidad del miembro que generó una firma y que, en ciertos esquemas, **revoque** la capacidad de la clave de dicho miembro para generar firmas válidas.

Uno de los primeros problemas que surgieron con las firmas grupales es que la longitud de las firmas no fuera incrementándose con el número de miembros del grupo (problema solucionado en **[1]**). Además, surgió también la necesidad de ser capaces de determinar qué mensajes habían sido firmados con una determinada clave (**trazabilidad**) **[2]** con el fin de detectar, por ejemplo, todos los mensajes generados por un miembro que haya incumplido las normas (sin revelar la identidad de los miembros del grupo). También se han realizado esfuerzos en la dirección de obtener firmas grupales más cortas y rápidas de calcular **[3]** manteniendo en algunos casos propiedades como la trazabilidad **[30]**.

Es importante tener en cuenta que la anonimia de las firmas grupales está sujeta al número de miembros del grupo. Para este trabajo resultan de especial interés los esquemas de firmas grupales de la librería **libgroupsig [17]**. Se puede encontrar una explicación del funcionamiento y los fundamentos matemáticos de cada uno de los esquemas actuales de la librería en el **Anexo B**.

2.6. Smartcards y privacidad

Como se ha mencionado previamente, las smartcards son dispositivos que proporcionan diversas facilidades de uso para ciertas aplicaciones y servicios añadiendo la ventaja de su relativamente bajo coste. Dado que dichas smartcards pueden contener información que no deseamos que sea revelada a ciertas personas u organizaciones, es interesante pensar en las posibles maneras de conseguir privacidad utilizándolas y en los problemas más comunes que deberemos solucionar. Un problema importante que viene rápidamente a la mente es la posibilidad de que nuestras smartcards sean falsificadas. Una de las soluciones propuestas para este problema es la de incluir datos biométricos y/o personales de cada individuo en su smartcard **[11, 12]**. Cuantos más datos se incluyan mayor seguridad podrán proporcionar los sistemas en los que se usen las tarjetas a nivel de evitar falsificaciones, abusos, terrorismo, etc. pero con el importante coste de la pérdida de privacidad por parte del individuo pudiendo, en algunos casos, vulnerarse algunos derechos civiles/humanos individuales. En 1980 la **OECD** publicó una serie de principios para proteger la privacidad de la información y el traspaso de datos personales entre diferentes países **[13]**. Entre ellos cabe destacar:

- La limitación al almacenamiento de datos personales.
- La limitación de uso y revelación de datos personales en base a los propósitos para los que fueron recogidos teniendo en cuenta el consentimiento personal.
- Que las prácticas, algoritmos y desarrollos que utilicen datos personales sean públicos.
- La protección de los datos frente al acceso no autorizado y su revelación.

Actualmente, con el desarrollo de las smartcards se facilitan diferentes herramientas para ayudar a preservar la privacidad de sus usuarios. Entre ellas encontramos:

- **Cifrado**: Las smartcards actuales son capaces de realizar algoritmos de cifrado modernos para proteger los datos intercambiados con el exterior y para firmar los mensajes enviados.
- **Autenticación**: Se puede controlar el acceso a las aplicaciones de la smartcard por determinados dispositivos, usuarios u otras aplicaciones otorgando a cada uno un nivel de privilegios deseado.
- **Protección de datos**: Las smartcards permiten proteger datos dentro de ellas otorgando acceso a ellos solamente mediante previa autorización introduciendo una clave, un código PIN o un dato biométrico.

En algunos casos puede ser deseable o necesario que el nivel de privacidad del servicio sea tal que los usuarios de éste sean anónimos. Un ejemplo de esto pueden ser servicios como la atención médica telemática. Con respecto a este tipo de servicios, han sido propuestos diversos esquemas que utilizan smartcards para autenticar al usuario en un sistema de forma anónima, algunos de los cuales han sido probados inseguros para ciertos ataques en artículos posteriores. Uno de los principales problemas durante este trabajo ha sido el escaso número de fuentes en las que se hable sobre la implementación de los esquemas propuestos en simuladores o tarjetas reales. Por esta razón, es importante mencionar las referencias **[22, 23, 24, 25, 26]**, en particular **[24, 25]** en las que se habla sobre los problemas y se plantean soluciones a la hora de implementar los esquemas en la tecnología Java Card (que será la utilizada en el desarrollo de los protocolos propuestos más adelante en este documento).

Por último, es importante mencionar que en todo servicio que utilice smartcards y datos personales no debería recaer sobre las smartcards todo el peso de proteger la privacidad de dichos datos. Debe tenerse en cuenta qué agentes participan en el servicio y cuál es la información que necesitan para funcionar correctamente, asegurando que el tratamiento de dicha información en todos los puntos respeta la legalidad vigente en el tiempo y lugar del servicio y proporciona la máxima privacidad posible a los usuarios. Además, debe dejarse claro en todo momento el tipo de información recogida, así como el uso y tratamiento de ésta.

2.6.1. Reglamento (UE) 2016/679

El nuevo Reglamento General de Protección de Datos **[35]** fue redactado el 27 de abril de 2016 y será de obligatorio cumplimiento en España a partir del 25 de mayo de 2018. Este reglamento ha surgido con el fin de unificar la legislación con respecto a las empresas y organizaciones que tratan datos de ciudadanos europeos, evitando así diferentes criterios y regímenes sancionadores en función del país donde se realice el tratamiento. Debido a la inminente obligatoriedad de este nuevo reglamento y a la relación que tiene el tratamiento de los datos con este trabajo, se ha considerado interesante mencionar de forma resumida algunos de los aspectos que regula:

- **Ámbito territorial**: Estarán sujetas a este reglamento las empresas que traten datos personales de ciudadanos residentes en la Unión Europea (UE) aunque la empresa esté establecida fuera de la UE.
- **Consentimiento del interesado**: El consentimiento para autorizar el tratamiento de los datos debe ser libre, específico, informado e inequívoco. El consentimiento puede ser implícito cuando se deduzca de una acción del interesado como por ejemplo, navegar en una web aceptando el uso de "cookies" para monitorizar dicha navegación. Este consentimiento puede ser retirado por el interesado en cualquier momento y el responsable del tratamiento de los datos debe en todo momento ser capaz de demostrar el

consentimiento del interesado. En caso de que el tratamiento de los datos sea necesario para cumplir las obligaciones legales del responsable no será necesario dicho consentimiento.

- **Información al interesado**: La información a los interesados, tanto respecto a las condiciones de los tratamientos que les afecten (incluyendo el plazo de conservación de los datos y posibles transferencias internacionales) como en las respuestas a los ejercicios de derechos, deberá proporcionarse de forma concisa, transparente, inteligible y de fácil acceso con un lenguaje claro y sencillo (en formato escrito o electrónico).
- **Derechos de los interesados**: Se reconocen los siguientes derechos:
  1. El **derecho de acceso** a los datos del interesado pudiendo obtener una copia de los datos personales objeto del tratamiento.
  2. El **derecho al olvido** (es una manifestación de los derechos de cancelación u oposición en el entorno online).
  3. El **derecho a la limitación del tratamiento de los datos** que supone que, a petición del interesado, no se aplicarán a sus datos personales las operaciones de tratamiento que en cada caso corresponderían. Se podría solicitar este derecho, por ejemplo, mientras el responsable determina si procede atender una solicitud de cancelación u oposición por parte del interesado.
  4. El **derecho a la portabilidad** de los datos de un interesado de un responsable a otro cuando el interesado lo solicite.
  5. El **derecho a no ser objeto de una elaboración de perfiles** basada únicamente en el tratamiento automatizado, cuando la decisión que pueda ser tomada a consecuencia de la misma pueda producir efectos jurídicos que puedan afectar al interesado, y con derecho a reclamar al responsable una intervención humana y a impugnar la decisión.
- **Anonimización**: Los principios de protección de datos no serán aplicables a información o datos anónimos, es decir, aquellos que no permitan relacionarlos con una persona natural. Esto implica que el reglamento no se aplicará al procesamiento de información anónima, aunque sea con propósitos estadísticos o de investigación.
- **Evaluación de riesgos**: Las empresas deberán evaluar el grado de riesgo que supone para los interesados el tratamiento de sus datos, debiendo realizar una evaluación del impacto cuando dicho tratamiento presente alto riesgo para los derechos y libertades de los interesados.
- **Medidas de seguridad**: Se establecerá una política de seguridad que garantice mediante medidas prácticas que los niveles de seguridad se corresponden a la protección de datos necesaria según los riesgos previstos.
- **Protección de los datos**: Las empresas deberán garantizar la protección de los datos en cualquier fase del tratamiento, asegurar que el tratamiento se realiza para fines específicos y aplicar técnicas de minimización de los datos.

Dentro de este resumen se ha considerado especialmente relevante por su relación con este trabajo el punto que informa de que los datos anónimos no estarán sujetos a los principios de protección de datos. A consecuencia de ello, siempre que podamos asegurar el anonimato de los datos tendremos mucha más libertad en lo respectivo a su tratamiento.

3. Análisis y diseño de los protocolos propuestos

En este apartado se proponen tres protocolos diferentes como posibles implementaciones para utilizar las firmas grupales en smartcards. En los dos primeros estamos partiendo de la base de que el esquema de firmas utilizado es **CYP06** (**Anexo B.4.**), es decir, permite generar firmas anónimas (dentro de un grupo) de mensajes de una longitud en torno a 362 bytes y el administrador del grupo puede revelar y trazar las firmas de los mensajes en caso necesario y revocar una clave de grupo. En el último se asume que el esquema de firmas utilizado es el de **CS97** (explicado en el **Anexo B.1.)**.
En los protocolos que se use cifrado **RSA** se ha supuesto que todos los agentes tienen acceso a las claves **RSA** públicas del resto y que el algoritmo de cifrado utilizado es **RSA-2048-PKCS#1-OAEP [40]**. Esto último implica que se procesará el texto plano antes de encriptarlo con **RSA** de forma que su longitud sea un múltiplo de la longitud de la clave e incluyendo aleatoriedad para así evitar que el mismo texto plano genere como resultado el mismo texto encriptado y que se puedan desencriptar de forma parcial partes del texto cifrado.
Con el fin de evitar el acceso al servicio por parte de usuarios no autorizados se ha supuesto que los usuarios válidos se han registrado previamente (por ejemplo, si fuesen alumnos de la Universidad Autónoma de Madrid o miembros de una comunidad de vecinos que utilizan un servicio anónimo). Además, los usuarios que vayan a tener acceso al servicio dispondrán de un ordenador con el software necesario para ejecutar las diferentes funcionalidades que ofrecen los esquemas de firmas mencionados anteriormente. Los protocolos se han planteado de forma incremental en cuanto al anonimato que proporcionan a sus usuarios (desde la pseudonimia del **Protocolo 0**, pasando por el control de la anonimia por parte de los usuarios en el **Protocolo 1**, al anonimato del **Protocolo 2**). Antes de pasar a los protocolos se explicará la siguiente notación:

$\{\boldsymbol{Salida_1}, \boldsymbol{Salida_2}, \dots, \boldsymbol{Salida_m}\} \leftarrow \boldsymbol{f}(\boldsymbol{Param_1}, \boldsymbol{Param_2}, \dots \boldsymbol{Param_n})$: Se utiliza para indicar un proceso llamado $\boldsymbol{f}$ que se lleva a cabo en uno de los sistemas que compone el protocolo. El proceso recibe como argumentos de entrada los parámetros $(\boldsymbol{Param_1}, \boldsymbol{Param_2}, \dots \boldsymbol{Param_n})$ generando $\{\boldsymbol{Salida_1}, \boldsymbol{Salida_2}, \dots, \boldsymbol{Salida_m}\}$ al finalizar.

3.1. Protocolo 0

En este protocolo se suponen dos tipos actores. El gestor de grupo (en adelante $\boldsymbol{GM}$) y un usuario (en adelante $\boldsymbol{U_i}$) dado que la funcionalidad será la misma para todos los usuarios. La idea de este protocolo es la siguiente: Un usuario válido podrá acceder al sistema e identificarse para obtener su clave de grupo. Posteriormente, con dicha clave de grupo generará un mensaje anónimo para solicitar una clave simétrica que podrá utilizar para enviar mensajes al grupo bajo el pseudónimo que identifica a su clave simétrica.

El protocolo consta de los siguientes pasos (ver **Figura 3.1.**):

1. En primera instancia el $\boldsymbol{GM}$ creará el grupo mediante el método $\boldsymbol{setup}$:

   $\{\boldsymbol{PU_{GR}}, \boldsymbol{PR_{GM}}\} \leftarrow \boldsymbol{setup}(\mathbf{1}^{\boldsymbol{k}})$

   El parámetro de seguridad $\boldsymbol{k}$ indica la complejidad computacional del algoritmo criptográfico en tiempo polinómico respecto a $\boldsymbol{k}$. Este método genera la clave pública de grupo $\boldsymbol{PU_{GR}}$ (necesaria para diversas acciones como firmar, verificar, incorporarse al grupo, etc.) y la clave privada del administrador $\boldsymbol{PR_{GM}}$ (necesaria para dar de alta miembros dentro del grupo o

revelar la firma de un mensaje). Además, el servidor tendrá una base de datos o archivo con las credenciales que identifiquen a los futuros usuarios del servicio (por ejemplo, el acceso controlado a un edificio) como el DNI, el nombre y los apellidos.

2. Una vez está creada la base del grupo, cada futuro usuario de la plataforma que quiera utilizar el servicio deberá primero obtener la clave pública del grupo para después realizar una petición de $\boldsymbol{join}$:

$$\{\boldsymbol{y_i}, \boldsymbol{cert_i}\} \leftarrow \boldsymbol{join}(\boldsymbol{PU_{GR}}, \boldsymbol{x_i})$$

Esta petición de $\boldsymbol{join}$ consta de tres fases: En la primera, $\boldsymbol{U_i}$ generará un número aleatorio $\boldsymbol{x_i}$ (que será su clave secreta de miembro) y enviará un mensaje cifrado con la clave pública **RSA** del $\boldsymbol{GM}$ solicitando la entrada e incluyendo una marca temporal $\boldsymbol{T1}$ y una firma **RSA** del mensaje o su propia clave pública ($\boldsymbol{PU_{U_i}}$) como firma (esto solo es necesario para que el $\boldsymbol{GM}$ sepa que clave utilizar para responder a los mensajes). En la segunda fase, el $\boldsymbol{GM}$ contestará a $\boldsymbol{U_i}$ incluyendo otra marca temporal $\boldsymbol{T2}$ junto con la anterior y cifrando con la clave pública **RSA** de $\boldsymbol{U_i}$. Al estar presente la primera marca temporal $\boldsymbol{T1}$, $\boldsymbol{U_i}$ puede asegurarse que ha sido el $\boldsymbol{GM}$ quien ha descifrado su primer mensaje. Esto es así porque al estar encriptado el primer mensaje con la clave pública del $\boldsymbol{GM}$ sólo el $\boldsymbol{GM}$ puede descifrarlo y obtener el valor de $\boldsymbol{T1}$ para incluirlo en el segundo mensaje que, de no haber podido descifrar el mensaje, no podría obtener. Entonces, en la tercera fase, $\boldsymbol{U_i}$ le enviará al $\boldsymbol{GM}$ un mensaje incluyendo $\boldsymbol{T2}$, una firma **RSA** o su clave pública y su DNI u otros datos que le identifiquen de forma unívoca junto con $\boldsymbol{y_i = P_1 x_i}$ con $\boldsymbol{P_1}$ perteneciente a $\boldsymbol{PU_{GR}}$. De esta forma, el $\boldsymbol{GM}$ no puede conocer el valor de la clave secreta de miembro de grupo $\boldsymbol{x_i}$. Una vez el $\boldsymbol{GM}$ ha validado el DNI en su lista de clientes válidos, generará un certificado $\boldsymbol{cert_i}$ que enviará a $\boldsymbol{U_i}$ cifrado con su clave pública **RSA**. Este certificado permite que las firmas generadas a partir de él y la clave privada $\boldsymbol{x_i}$ asociada a $\boldsymbol{y_i}$ sean consideradas firmas válidas. Además, guardará $\boldsymbol{cert_i}$ e $\boldsymbol{y_i}$ relacionados en una tabla a la que solamente él/ella tenga acceso para poder utilizarlo en caso de ser necesario trazar la firma en un futuro. Este tipo de petición en tres fases evita que un oponente capture los mensajes y pueda volver a utilizarlos para su propio beneficio (ataque por repetición).

3. Una vez $\boldsymbol{U_i}$ tiene la clave privada y el certificado puede solicitar una clave simétrica para comunicarse con el servidor firmando un mensaje mediante $\boldsymbol{sign}$:

$$\{\boldsymbol{SIG_m}\} \leftarrow \boldsymbol{sign}(\boldsymbol{m}, \boldsymbol{PU_{GR}}, \boldsymbol{cert_i}, \boldsymbol{x_i})$$

Después de tener la firma del mensaje realizará una petición de tres fases similar al paso anterior para solicitar la clave simétrica $\boldsymbol{K_i}$. Al final de la tercera fase incluirá, en este caso, $\boldsymbol{SIG_m}$, $\boldsymbol{m}$ y la traza temporal pertinente. Tras recibir este mensaje el $\boldsymbol{GM}$ generará una clave simétrica para **AES-CBC-256** y la enviará cifrada con la clave **RSA** del usuario $\boldsymbol{U_i}$ junto con un identificador $\boldsymbol{ID_i}$. Además, guardará la relación entre el mensaje firmado y la clave simétrica proporcionada para poder trazar las acciones de un usuario en caso de ser necesario.

4. El usuario $\boldsymbol{U_i}$ introducirá su clave simétrica en su smartcard personal $\boldsymbol{SC_i}$ y protegerá la clave con un PIN.

5. A la hora de acceder al servicio, el usuario $\boldsymbol{U_i}$ podrá utilizar su smartcard para cifrar el comando pertinente ($\boldsymbol{cmd}$), recibido desde el ordenador a través de un lector de tarjetas, con la clave simétrica. Se le solicitará su identificador de usuario $\boldsymbol{ID_i}$ para incluirlo en claro en el mensaje enviado al $\boldsymbol{GM}$. Se incluirá además una traza temporal para evitar una posible

repetición de un comando por parte de un intruso, siendo responsabilidad del ***GM*** asegurar que el mensaje no está repetido y guardar los mensajes procesados correctamente en una base de datos segura. El identificador $\boldsymbol{ID_i}$ estará presente en claro en los mensajes para permitir al ***GM*** conocer qué clave simétrica utilizar en cada caso. El mensaje pasará por el ordenador con lector de tarjetas conectado al servidor del ***GM***.

Este protocolo presenta la ventaja de que es sencillo de implementar y tiene un bajo coste computacional para la tarjeta, dada la sencillez de cifrar utilizando un algoritmo de clave simétrica como **AES**. La desventaja es que no genera anonimia para los usuarios sino pseudonimia. Esto se debe a que, en todo momento, el ***GM*** puede ver la secuencia de comandos solicitados y mensajes enviados por un miembro en particular al estar cifrados con la misma clave.

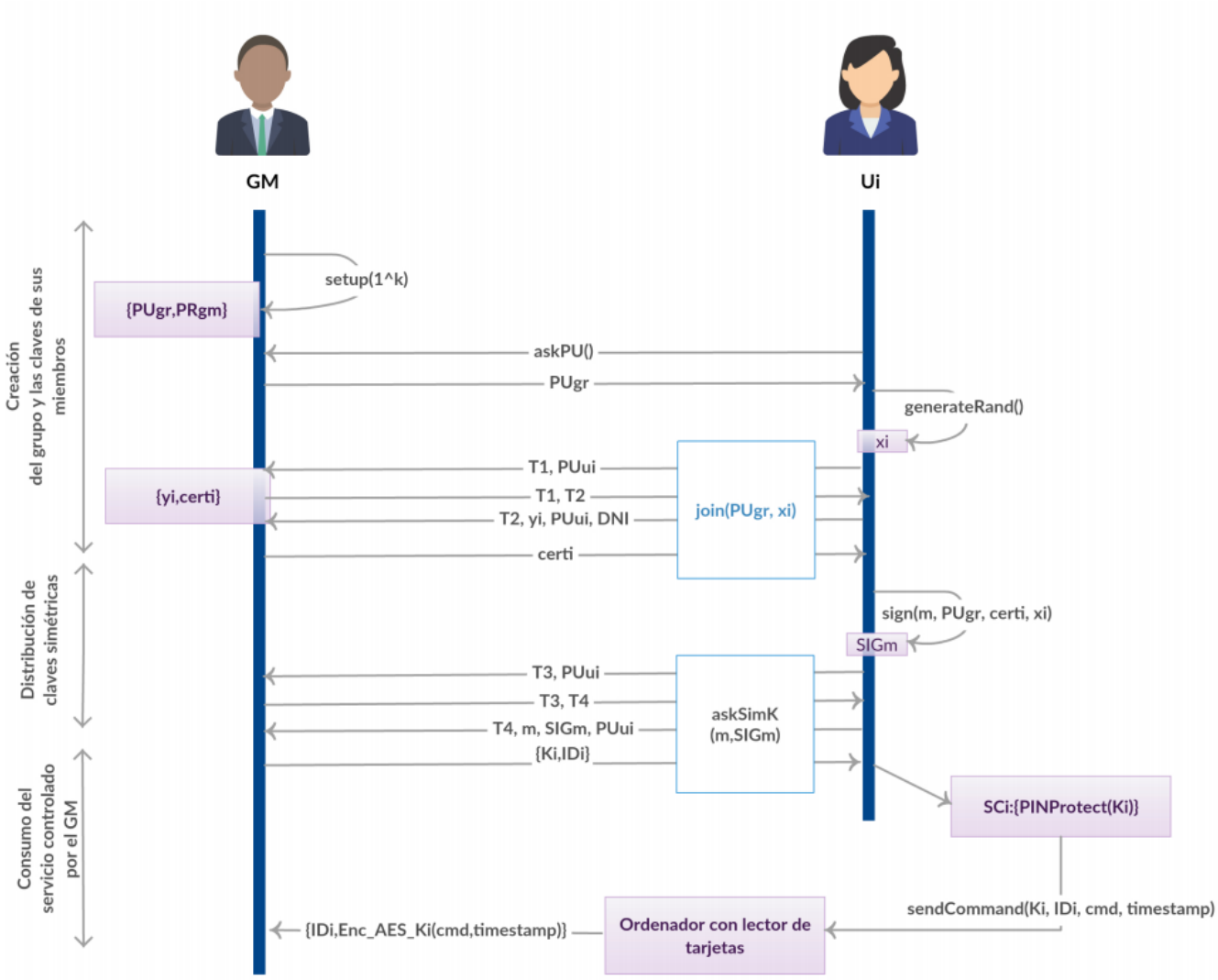

Figura 3.1. Protocolo 0.

3.1.1. Requisitos funcionales y no funcionales para el Protocolo 0

3.1.1.1. Requisitos funcionales

- **RF1: Aleatoriedad.** Tanto el $\boldsymbol{GM}$ como los usuarios tienen la capacidad de generar números aleatorios o pseudoaleatorios para las partes del código necesarias.
- **RF2: Autenticación de los usuarios.** Información identificativa de los usuarios (como un DNI y/o un número secreto que hayan recibido previamente) será guardada en el servidor con el fin de poder autenticar a los usuarios que pueden consumir el servicio.
- **RF3: Privacidad de las comunicaciones.** Los mensajes entre el servidor y los usuarios van cifrados bien con un cifrado de clave simétrica o uno de clave asimétrica asegurando que solo cada usuario concreto y el servidor conocen el contenido de las comunicaciones.
- **RF4: Conexión segura entre cliente y servidor.** Los mensajes se intercambian entre cliente y servidor sin peligro de suplantación de identidades o ataques por repetición. Esto lo aseguran las claves públicas y simétricas intercambiadas en el protocolo, las marcas temporales introducidas en los mensajes y las firmas grupales utilizadas.
- **RF5: Pseudonimia de los usuarios al consumir el servicio.** Los usuarios no pueden ser identificados de forma real ya que se ocultan bajo una clave simétrica con el servidor que ha sido obtenida mediante una petición firmada con firmas grupales. El hecho de usar la misma clave simétrica no impide relacionar mensajes cifrados con la misma clave, por lo que se obtiene pseudonimia y no anonimia.
- **RF6: Protección de las claves de acceso.** Las claves de acceso al servicio están protegidas dentro de cada smartcard mediante un número PIN que impide su lectura y uso desde el exterior.

3.1.1.2. Requisitos no funcionales

- **RNF1: Múltiples conexiones.** El servidor debe ser capaz de soportar varias peticiones simultáneas de usuarios de forma concurrente.
- **RNF2: Velocidad de respuesta.** El servidor debe poder responder de forma rápida a las peticiones de los usuarios y no saturarse con un número de peticiones no muy elevado.
- **RNF3: Eficiencia de las smartcard.** Las funciones de las smartcard para generar los mensajes a la hora de consumir el servicio que proporciona el servidor deben realizarse en un tiempo razonable.
- **RNF4: Privacidad de la smartcard.** Se entiende que la smartcard es de uso exclusivo de cada usuario y que no se generan varias smartcards con claves iguales, ya que esto iría en detrimento de dicho usuario si alguno de estos casos se diera y no se comunicase para invalidar dicha clave de acceso.
- **RNF5: Simplicidad.** El usuario debe disponer de una interfaz sencilla para introducir datos dentro de su tarjeta.

3.1.2. Casos de uso y diagrama de secuencia para el Protocolo 0

En este protocolo tendremos al servidor, controlado por el administrador del grupo, y a los múltiples usuarios que puedan utilizar los servicios. Para simplificar vamos a suponer que el servicio se trata del acceso a unas instalaciones o a un repositorio de documentos. En este caso las posibles acciones de los usuarios serán las siguientes (ver **Figura 3.2.**):

- Solicitar acceso al grupo.
- Solicitar una clave simétrica para comunicarse.
- Mandar un mensaje de acceso a las instalaciones o al repositorio.
- Mandar un mensaje de salida de las instalaciones o del repositorio.

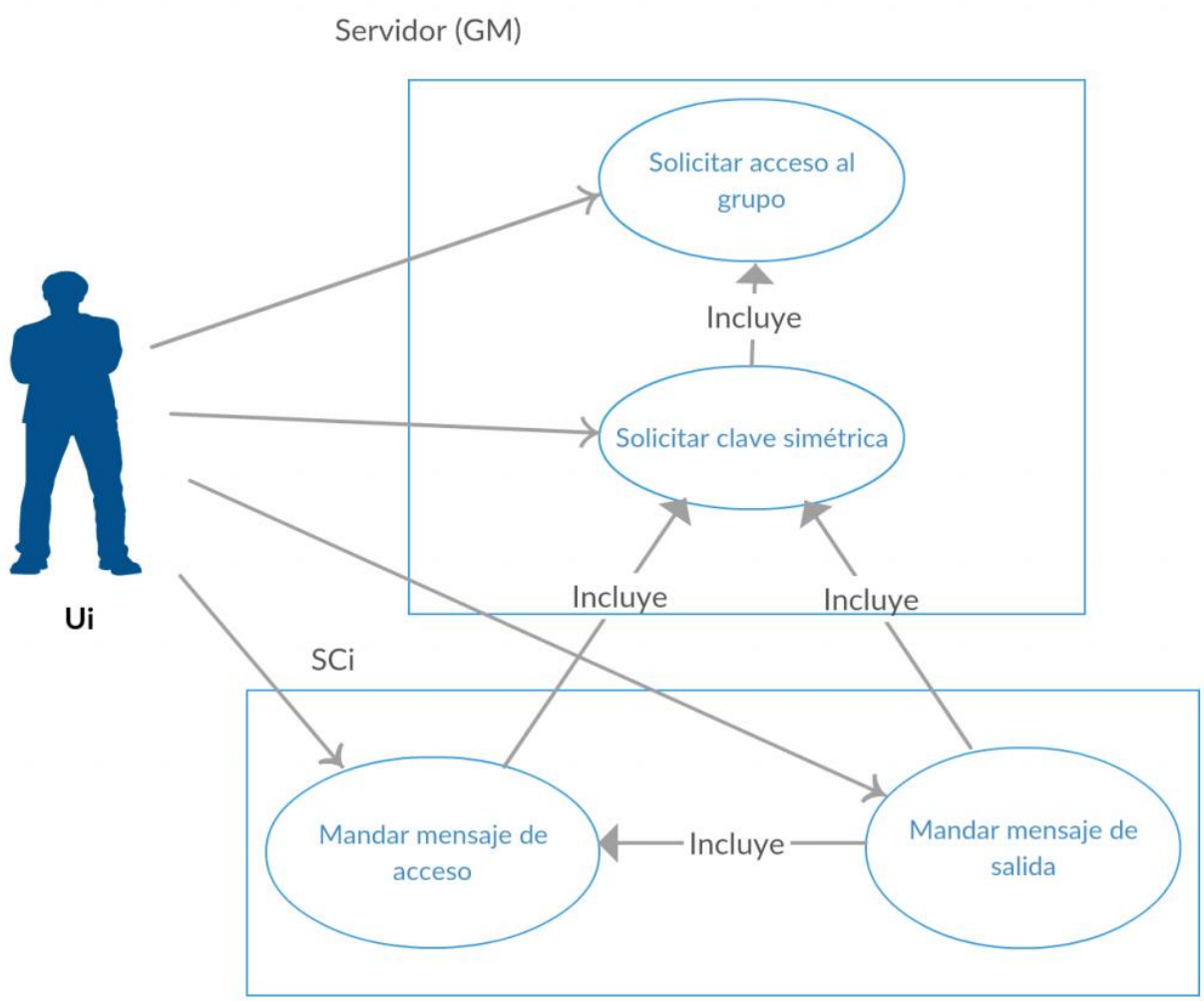

Figura 3.2. Diagrama de casos de uso Protocolo 0.

Los procesos que se llevan a cabo suponen, en general, cifrar o descifrar utilizando una clave simétrica o asimétrica y, en alguna ocasión, firmar el mensaje. Además, el servidor controlado por el ***GM*** comprobará que los mensajes enviados por los usuarios son correctos. A continuación son mostrados los procesos en un diagrama de secuencia (ver **Figura 3.3.**):

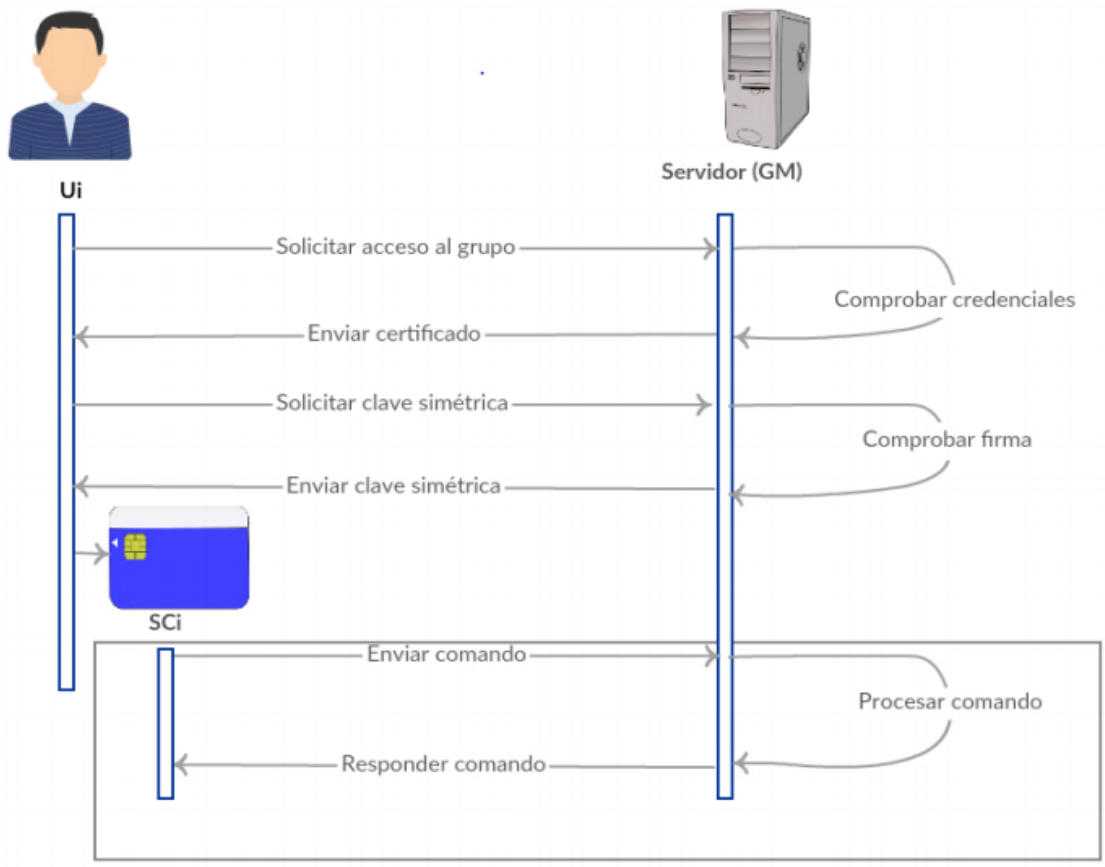

Figura 3.3. Diagrama de secuencia Protocolo 0.

3.2. Protocolo 1

La idea de este protocolo es implementar una forma sencilla de anonimia. Para ello, una vez obtenida la clave de miembro, se podrán generar mensajes de texto aleatorio y firmarlos en nombre del grupo. Varios pares (mensaje, firma) podrán ser introducidos en la tarjeta y utilizados para acceder al sistema (se usarían de forma secuencial repitiéndose al llegar al último). Si renovamos los mensajes de la tarjeta con suficiente frecuencia, nunca usaremos el mismo par (mensaje, firma) y por lo tanto conseguiremos anonimia ya que no se podrán trazar los mensajes mientras que, si repetimos alguno de los pares para interactuar con el sistema, perderíamos anonimia al poder relacionar los pares que sean iguales.

En este protocolo los dos primeros pasos son idénticos a los del **Protocolo 0**. En este punto, al terminar el segundo paso, el usuario $\boldsymbol{U_i}$ tendrá su certificado y su clave privada para poder firmar mensajes en nombre del grupo. Entonces el usuario firmará una serie de mensajes para poder incluirlos en la smartcard. Redondeando la firma a unos 400 bytes y suponiendo unos mensajes de 100 bytes de longitud (los mensajes pueden ser cadenas de texto aleatorio ya que lo que importa es que su firma sea válida) como máximo tenemos que cada par (mensaje, firma) ocuparía 500 bytes en la smartcard. Teniendo en cuenta una memoria flash de 40 Kbytes podríamos incluir 50 pares (mensaje, firma) en la tarjeta ocupando algo más de la mitad de su memoria no volátil. Una vez el usuario ha introducido los mensajes y las firmas en la smartcard protegiéndolos con el PIN podrá utilizarlos para enviar comandos al sistema. A partir de aquí, puede ser la propia smartcard

la que cifre el mensaje, su firma, el comando y una marca temporal con la clave **RSA** pública del ***GM***, o se puede dejar este trabajo al ordenador con el lector de tarjetas que consideramos seguro en el sentido de que recibirá el par (mensaje, firma) a través del lector y simplemente los cifrará y enviará sin guardarlos. Se ha considerado la segunda opción en el esquema mostrado del **Protocolo 1** (ver **Figura 3.4.**)dado que este ordenador simplemente debe pertenecer al servicio que controla el ***GM*** y cumplir los requisitos de poder ser accedido solamente mediante el lector de tarjetas para cifrar con **RSA** y comunicarse exclusivamente con el ***GM***. A la hora de enviar un comando la tarjeta seleccionará un par (mensaje, firma) que no haya utilizado anteriormente (hasta que no queden más en cuyo caso volverá a empezar desde el principio).

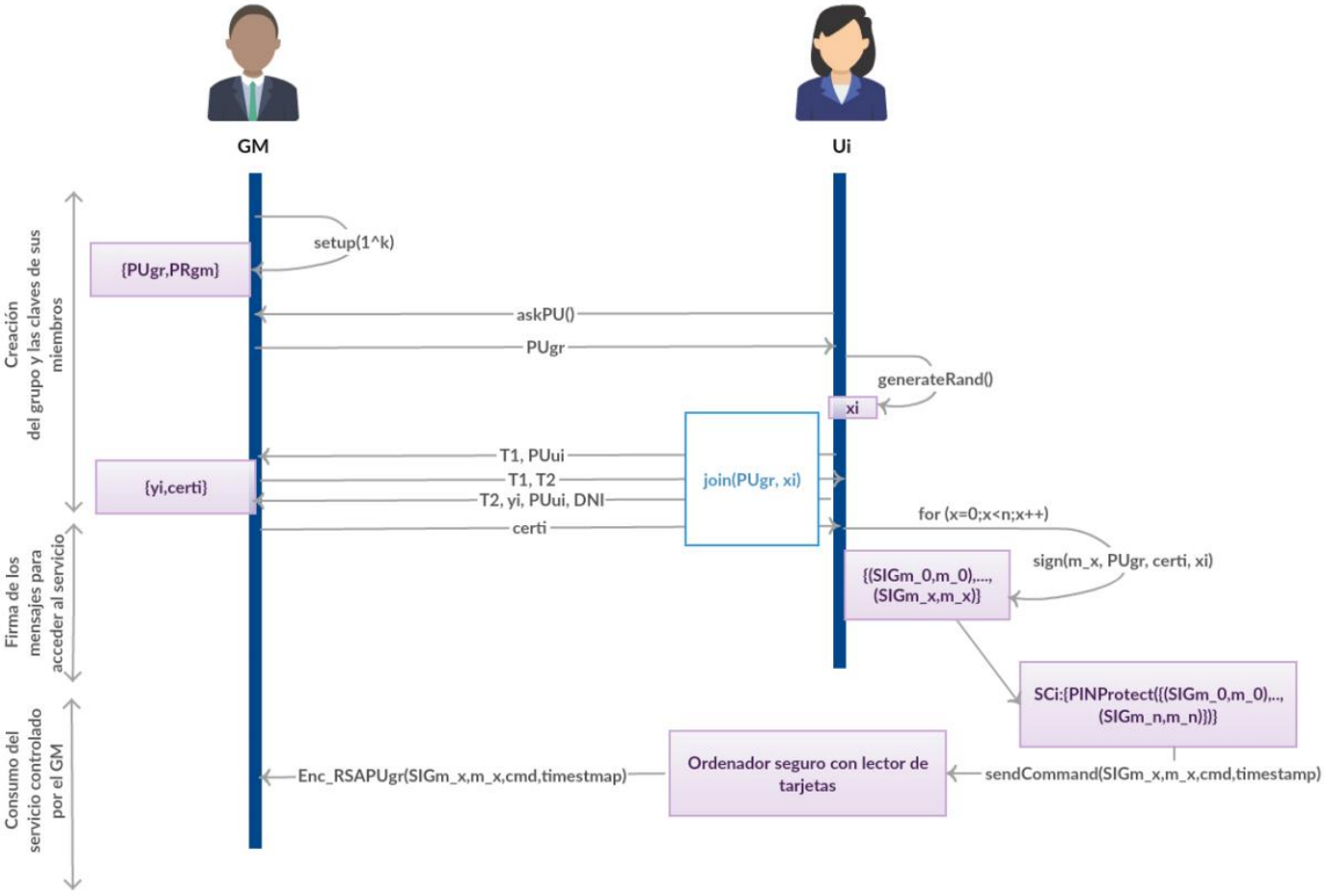

Figura 3.4. Protocolo 1.

Este protocolo presenta la ventaja de que el usuario puede renovar cada cierto tiempo los pares (mensaje, firma) de su tarjeta de forma que sólo se utilice un par cada vez. De este modo, se conseguiría **anonimato** para el usuario si el servicio lo utilizasen varios usuarios al mismo tiempo. Los problemas que presenta es que si un par se usa más de una vez se puede establecer una conexión entre los comandos con el mismo par (aunque sería menor a la de utilizar siempre la misma clave como en el **Protocolo 0**) y que si solamente hay un usuario utilizando el servicio en una franja temporal (por ejemplo, si a un edificio solo accede un usuario durante un día, enviaría un comando para entrar y otro para salir cada uno con un par distinto) se podrían establecer relaciones entre diferentes pares de (mensaje, firma). Una posible variación de este protocolo sería considerar que los ordenadores seguros con lector de tarjetas pertenecen a otra entidad que colabora con el ***GM*** en la gestión del sistema. Debido a ello, el ***GM*** compartiría y actualizaría la

lista con las claves de miembro revocadas con dichos ordenadores. En este punto, los ordenadores podrían mandar a la tarjeta una secuencia aleatoria de bits de tamaño suficientemente grande (que denominaremos $\boldsymbol{C}$) para superar a una firma y su mensaje. La tarjeta comprobaría que todos los bits de $\boldsymbol{C}$ no son ceros y en ese caso, la utilizaría para realizar una operación XOR con una firma y su mensaje que tuviese guardados previamente. A continuación enviaría el resultado de la operación **XOR** (que denominaremos $\boldsymbol{R}$) al ordenador, que dentro de un procedimiento cerrado desharía la operación **XOR** y validaría la firma sin guardarla. En caso de ser válida la firma permitiría el acceso del miembro y enviaría $\boldsymbol{R}$ al $\boldsymbol{GM}$ guardándose $\boldsymbol{C}$ para poder revelar el miembro que generó la firma en caso de que el $\boldsymbol{GM}$ lo solicite. Con esta variación solo sería necesario un par (mensaje, firma) dentro de la tarjeta y se conseguiría anonimia siempre que el procedimiento para validar la firma no guarde ningún registro de la firma validada.

3.2.1. Requisitos funcionales y no funcionales para el Protocolo 1

3.2.1.1. Requisitos funcionales

- **RF1: Aleatoriedad.** Tanto el $\boldsymbol{GM}$ como los usuarios tienen la capacidad de generar números aleatorios o pseudoaleatorios para las partes del código necesarias.
- **RF2: Autenticación de los usuarios.** Información identificativa de los usuarios (como un DNI y/o un número secreto que hayan recibido previamente) será guardada en el servidor con el fin de poder autenticar a los usuarios que pueden consumir el servicio.
- **RF3: Privacidad de las comunicaciones.** Los mensajes entre el servidor y los usuarios estarán encriptados con un cifrado de clave asimétrica asegurando que solamente cada usuario concreto y el servidor conocen el contenido de las comunicaciones entre ellos.
- **RF4: Conexión segura entre cliente y servidor.** Los mensajes se intercambian entre cliente y servidor sin peligro de suplantación de identidades o ataques por repetición. Esto lo aseguran las claves públicas **RSA** utilizadas en el protocolo, las marcas temporales introducidas en los mensajes intercambiados y las firmas grupales utilizadas.
- **RF5: Control de la anonimia por parte de los usuarios.** Al introducir en la tarjeta previamente mensajes firmados como miembros del grupo, los usuarios pueden consumir los servicios del servidor de forma anónima siempre que no utilicen más de una vez el mismo mensaje para enviar una petición al servidor (o siempre que utilicen la variación del protocolo descrita anteriormente). En caso de no renovar los mensajes antes de utilizar más de una vez uno o varios mensajes, se podría establecer una conexión entre peticiones con los mismos mensajes (aunque no entre peticiones con mensajes diferentes del mismo miembro) por lo que se obtendría lo que denominamos como anonimia débil o soft-anomimia.
- **RF6: Protección de los mensajes y sus firmas.** Los mensajes y sus firmas utilizados para acceso al servicio están protegidas dentro de cada smartcard mediante un número PIN que impide su lectura y uso desde el exterior.
- **RF7: Independencia del ordenador seguro.** El actor denominado “Ordenador seguro” es ajeno a los intereses del administrador del grupo y de cualquier usuario. Simplemente se encarga de leer el comando, cifrarlo para que solamente pueda leerlo el servidor y borrar el comando.

3.2.1.2. Requisitos no funcionales

- **RNF1: Múltiples conexiones.** El servidor debe ser capaz de soportar varias peticiones simultáneas de usuarios de forma concurrente.
- **RNF2: Velocidad de respuesta.** El servidor debe poder responder de forma rápida a las peticiones de los usuarios y no saturarse con un número de peticiones no muy elevado.
- **RNF3: Eficiencia de las smartcard.** Las funciones de las smartcard para generar los mensajes a la hora de consumir el servicio que controla el servidor deben realizarse en un tiempo razonable.
- **RNF4: Privacidad de la smartcard.** Se entiende que la smartcard es de uso exclusivo de cada usuario y que no se generan varias smartcards con mensajes firmados por el mismo miembro, ya que esto iría en detrimento de dicho usuario si esto sucediera y no se comunicase para invalidar dichas tarjetas.
- **RNF5: Simplicidad.** El usuario debe disponer de una interfaz sencilla para introducir los mensajes y sus firmas dentro de la tarjeta y para generar dichos mensajes y firmas.

3.2.2. Casos de uso y diagrama de secuencia para el Protocolo 1

En este protocolo tendremos al servidor, controlado por el administrador del grupo, y a los múltiples usuarios que puedan utilizar los servicios. También estará el "Ordenador seguro" que permitirá a los usuarios cifrar los mensajes con los comandos destinados al servidor. Para simplificar vamos a suponer que el servicio se trata del acceso a unas instalaciones o a un repositorio de documentos. En este caso las posibles acciones de los usuarios serán las siguientes:

- Solicitar acceso al grupo.
- Firmar mensajes e introducirlos en la smartcard.
- Mandar un mensaje de acceso a las instalaciones o al repositorio.
- Mandar un mensaje de salida de las instalaciones o del repositorio.

Para este protocolo existe un proceso de solicitud de entrada al grupo que permite obtener un certificado para firmar mensajes de forma anónima. El resto de procesos suponen utilizar el certificado para generar firmas que pueda utilizar la smartcard a posteriori para hacer peticiones al servidor. Además, dicho servidor comprobará que los mensajes enviados por los usuarios son correctos. A continuación, son mostrados el diagrama de casos de uso (**Figura 3.5.**) y los procesos en un diagrama de secuencia (**Figura 3.6.**):

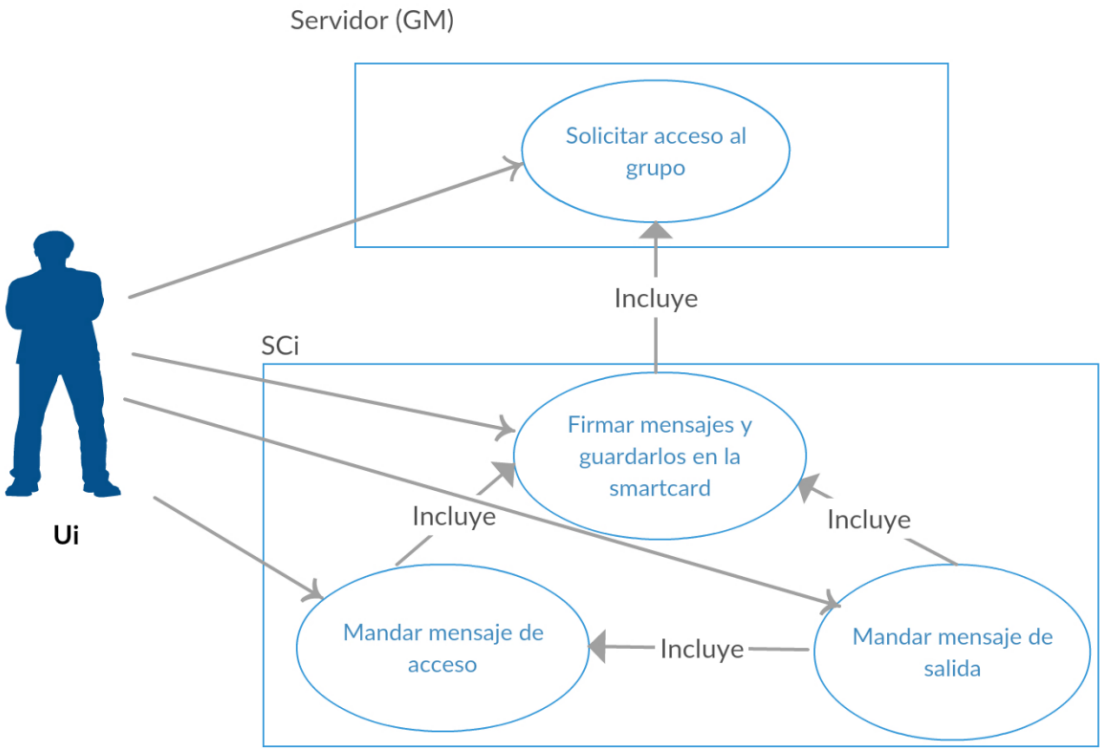

Figura 3.5. Diagrama de casos de uso Protocolo 1.

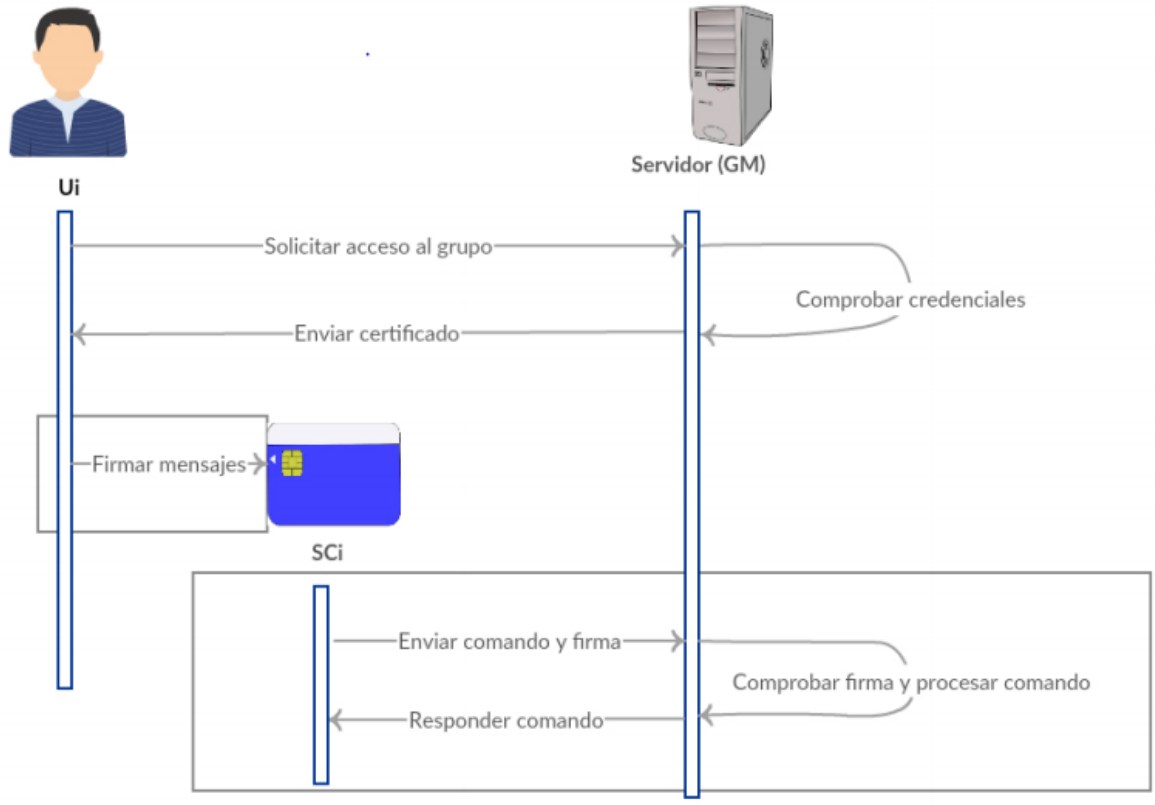

Figura 3.6. Diagrama de secuencia Protocolo 1.

3.3. Protocolo 2

En este protocolo (**Figura 3.7.**) la idea es que la tarjeta sea capaz de generar de forma autónoma mensajes y firmarlos en nombre del grupo. Se ha elegido por su sencillez respecto a las utilidades que proporciona el esquema de firmas **CS97** (**Anexo B.1.**). A pesar de ser un protocolo relativamente sencillo hemos identificado los siguientes desafíos en la implementación cuya resolución aún está pendiente:

- Obtención eficiente de raíces primitivas módulo **n** (siendo **n** un módulo **RSA**) para poder obtener el generador del grupo cíclico en el que se basa parte de la clave pública del grupo de firmas.
- Generación de números aleatorios pertenecientes a las unidades de $\boldsymbol{Z_n}$ dentro de la tarjeta para poder ocultar al $\boldsymbol{GM}$ los valores necesarios a la hora de generar las firmas de los mensajes.
- Multiplicación eficiente de dos elementos de un grupo cíclico dentro de la tarjeta.

En lo referente al diseño, el **Protocolo 2** comparte los dos primeros pasos con el **Protocolo 0** (del mismo modo que el **Protocolo 1**). Una vez que el usuario $\boldsymbol{U_i}$ ha recibido su certificado del $\boldsymbol{GM}$ en este protocolo, introduce tanto el certificado $\boldsymbol{cert_i}$ como la clave $\boldsymbol{x_i}$ dentro de la smartcard y los protege con el PIN. Cada vez que $\boldsymbol{U_i}$ desee enviar un comando al servidor que controla el $\boldsymbol{GM}$ podrá entonces firmar el mensaje correspondiente dentro de su tarjeta. En este protocolo la tarjeta deberá firmar un par (comando, traza temporal) que constituirá el mensaje $\boldsymbol{m\_x}$ a enviar al $\boldsymbol{GM}$. Posteriormente, del mismo modo que en el **Protocolo 1**, tanto el mensaje obtenido como su firma podrían cifrarse con la clave pública **RSA** del $\boldsymbol{GM}$ desde la tarjeta o desde el ordenador con lector considerándolo seguro en las mismas condiciones expuestas por el **Protocolo 1**. Por último, este ordenador se encargaría de cifrar el mensaje y enviarlo al servidor.

La gran ventaja que presenta el **Protocolo 2** es que la tarjeta puede generar las firmas de los comandos que necesite de forma autosuficiente preservando el anonimato de su usuario a no ser que sea revelado por el $\boldsymbol{GM}$. Otra ventaja es que, se pueden enviar comandos concretos a la tarjeta desde el lector para que los firme, permitiendo que el **Protocolo 2** tenga más aplicabilidad directa a un mayor tipo de servicios que el **Protocolo 1**. Esto es así porque en el **Protocolo 1** se utilizaban mensajes aleatorios para el acceso a las diferentes partes del servicio (se podrían guardar diferentes mensajes en la tarjeta con diferentes comandos pero esto reduciría la cantidad de mensajes que se pueden usar para cada acción). Por ello, el **Protocolo 2** sería más aplicable al consumir servicios que necesitasen información adicional en los comandos (no solamente acceso al servicio y salida). La principal desventaja es el tiempo necesario por la tarjeta para generar la firma debido a la necesidad de firmar todos los mensajes que se envían al $\boldsymbol{GM}$.

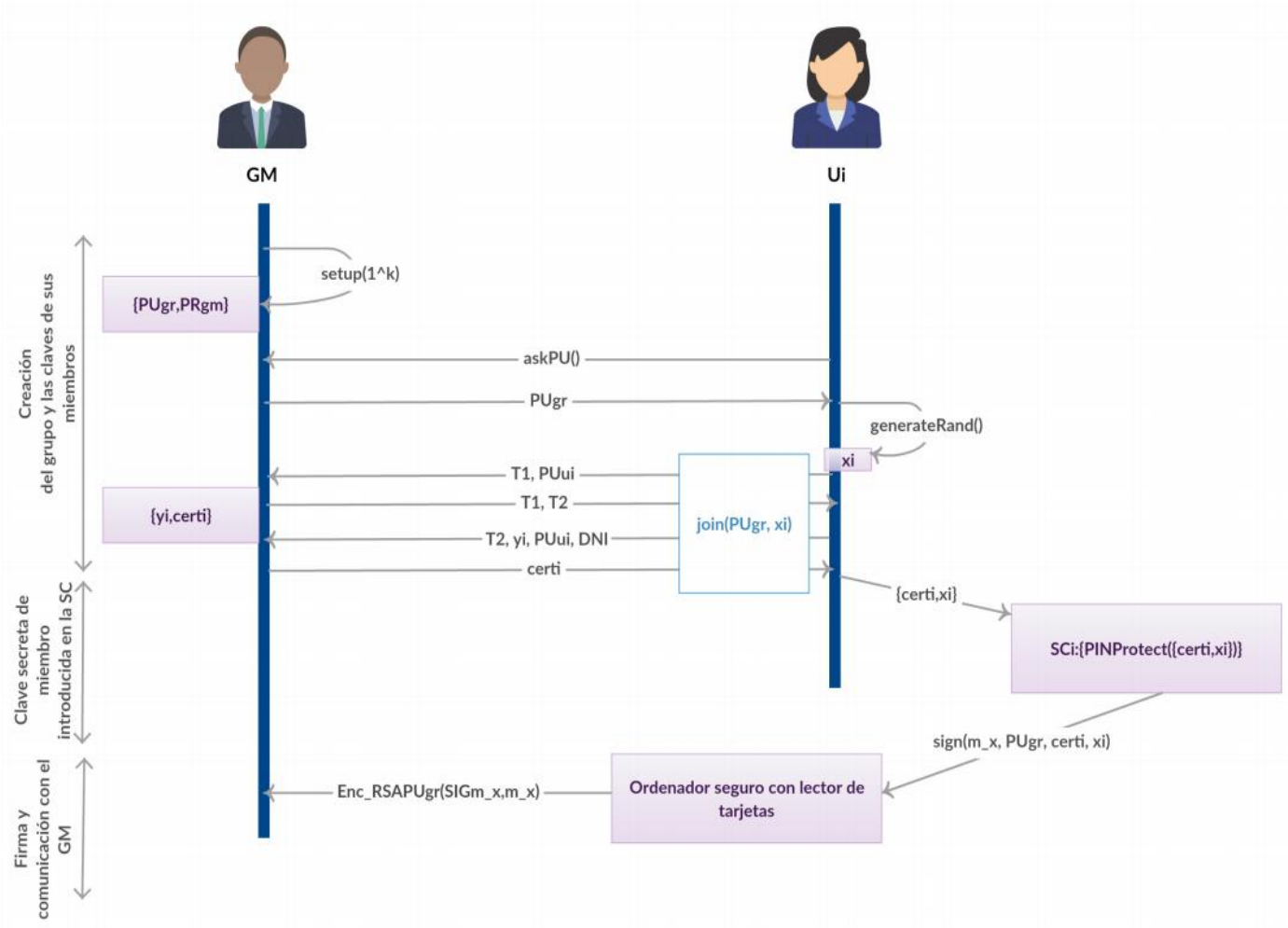

Figura 3.7. Protocolo 2.

3.3.1. Requisitos funcionales y no funcionales para el Protocolo 2

3.3.1.1. Requisitos funcionales

- **RF1: Aleatoriedad.** Tanto el ***GM*** como los usuarios tienen la capacidad de generar números aleatorios o pseudoaleatorios para las partes del código necesarias.
- **RF2: Autenticación de los usuarios.** Información identificativa de los usuarios (como un DNI y/o un número secreto que hayan recibido previamente) será guardada en el servidor con el fin de poder autenticar a los usuarios que pueden consumir el servicio.
- **RF3: Privacidad de las comunicaciones.** Los mensajes entre el servidor y los usuarios van encriptados con un cifrado de clave asimétrica asegurando que solamente cada usuario concreto y el servidor conocen el contenido de las comunicaciones entre ellos.
- **RF4: Conexión segura entre cliente y servidor.** Los mensajes se intercambian entre cliente y servidor sin peligro de suplantación de identidades o ataques por repetición. Esto lo aseguran las claves públicas **RSA** utilizadas en el protocolo, las marcas temporales introducidas en los mensajes intercambiados y las firmas grupales utilizadas.

- **RF5: Anonimia de los usuarios.** Las smartcards serán capaces de generar mensajes firmados de manera independiente de forma que, gracias a las firmas grupales se consigue anonimia completa para su propietario dentro del grupo de usuarios.
- **RF6: Protección del certificado y la clave de miembro**. Tanto el certificado como la clave secreta de miembro estarán protegidos dentro de la smartcard mediante un número PIN que impide su lectura y uso desde el exterior.
- **RF7: Independencia del ordenador seguro.** El actor denominado "Ordenador seguro" es ajeno a los intereses del administrador del grupo y de cualquier usuario. Simplemente se encarga de leer el comando, cifrarlo para que solamente pueda leerlo el servidor y borrar el comando.

3.3.1.2. Requisitos no funcionales

- **RNF1: Múltiples conexiones.** El servidor debe ser capaz de soportar varias peticiones simultáneas de usuarios de forma concurrente.
- **RNF2: Velocidad de respuesta.** El servidor debe poder responder de forma rápida a las peticiones de los usuarios y no saturarse con un número de peticiones no muy elevado.
- **RNF3: Eficiencia de las smartcard.** Las funciones de las smartcard para generar los mensajes y firmarlos para poder consumir el servicio del servidor deben realizarse en un tiempo razonable.
- **RNF4: Privacidad de la smartcard.** Se entiende que la smartcard es de uso exclusivo de cada usuario y que no se generan varias smartcards con el mismo certificado y clave de miembro, ya que esto iría en detrimento de dicho usuario en caso de producirse y que no se comunicase para invalidar dicha clave y certificado.
- **RNF5: Simplicidad.** El usuario debe disponer de una interfaz sencilla para introducir el certificado y la clave de miembro dentro de la tarjeta.

3.3.2. Casos de uso y diagrama de secuencia para el Protocolo 2

En este protocolo tendremos al servidor controlado por el administrador del grupo y a los múltiples usuarios que puedan utilizar los servicios. También estará el "Ordenador seguro" que permitirá a los usuarios cifrar los mensajes con los comandos destinados al servidor. Para simplificar vamos a suponer que el servicio se trata del acceso a unas instalaciones o a un repositorio de documentos. En este caso las posibles acciones de los usuarios serán las siguientes:

- Solicitar acceso al grupo.
- Mandar un mensaje de acceso a las instalaciones o al repositorio (firmados por la smartcard).
- Mandar un mensaje de salida de las instalaciones o del repositorio (firmados por la smartcard).

A continuación se muestran los diagramas de casos de uso (**Figura 3.8.**) y de secuencia (**Figura 3.9.**) del **Protocolo 2**:

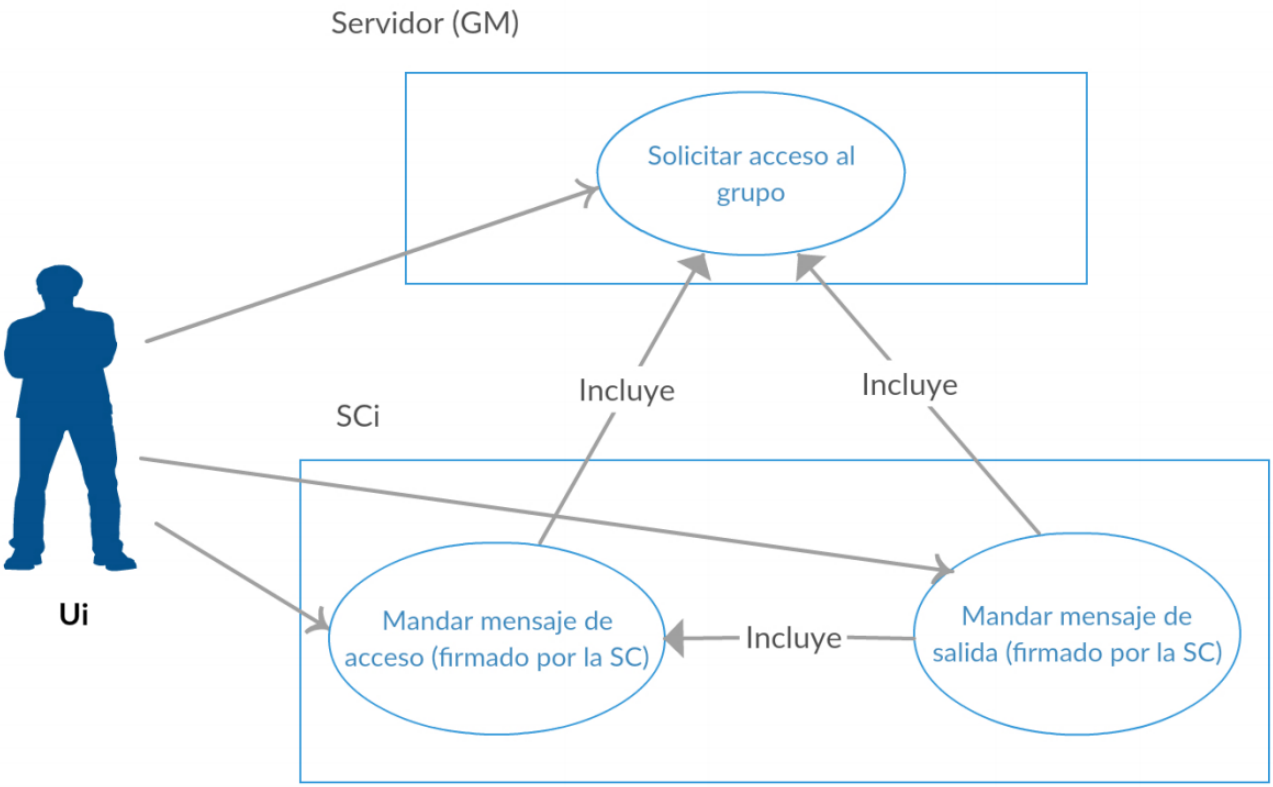

Figura 3.8. Diagrama de casos de uso Protocolo 2.

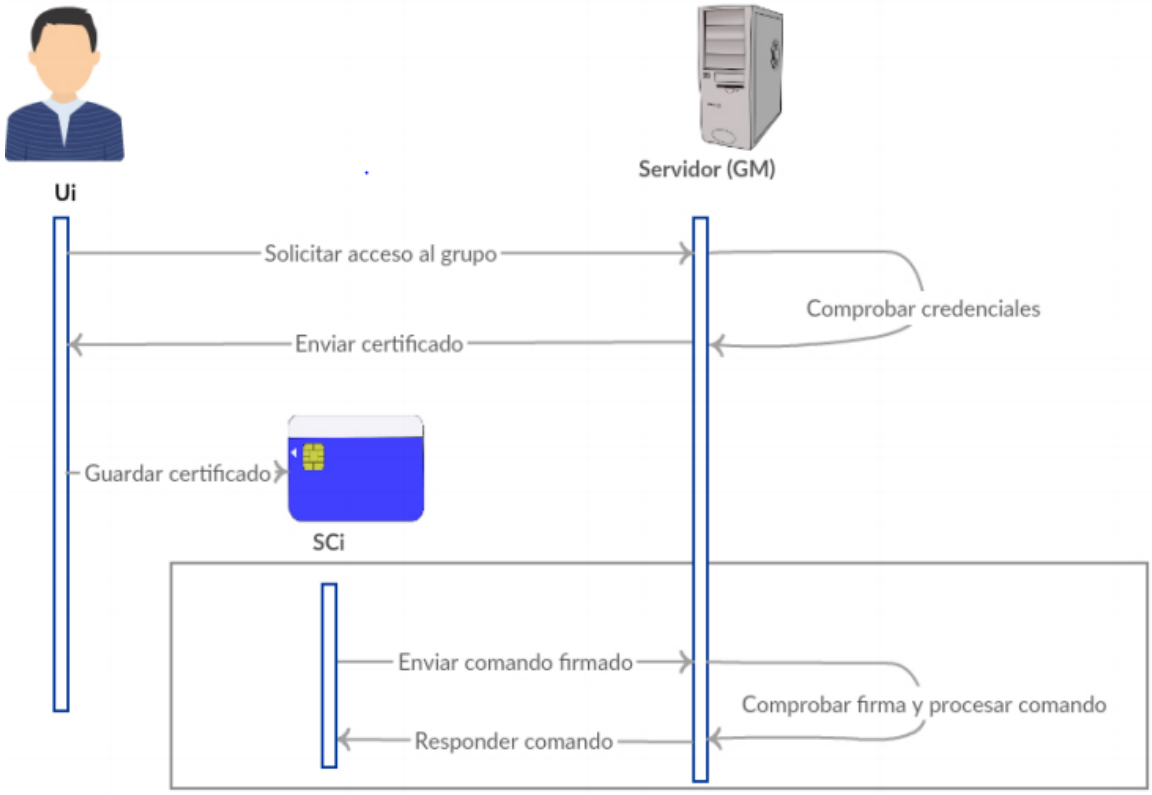

Figura 3.9. Diagrama de secuencia Protocolo 2.

En este protocolo es necesario que cada usuario realice primero un proceso de solicitud de certificado para poder firmar mensajes en nombre del grupo. Posteriormente, este certificado es guardado y protegido con un PIN en la smartcard que ya puede firmar de forma autónoma todos los comandos necesarios. Por su parte, el servidor comprobará la validez de las firmas y los comandos recibidos y los procesará.

3.4. Estudio de la seguridad de los protocolos propuestos

Ahora se analizará informalmente el nivel de seguridad de los protocolos propuestos anteriormente en base a la información que un atacante pudiera obtener de ellos. Este análisis se ha realizado tomando como ejemplo la referencia **[20]**. Para los tres protocolos se hacen las siguientes suposiciones básicas:

1. Todas las funciones unidireccionales que se utilizan son irrompibles.
2. El directorio de claves públicas es seguro, todo el mundo tiene acceso a él y no puede ser falsificado.
3. Solamente cada usuario conoce sus propias claves privadas (tanto la clave privada **RSA** como su clave privada de miembro de grupo).
4. El atacante puede obtener cualquier mensaje de la red.
5. El atacante puede iniciar una conversación con cualquier usuario de la red y recibir mensajes de cualquier usuario de la red.
6. Los esquemas de firmas grupales utilizados son seguros cumpliéndose las hipótesis en las que se basan. En este sentido, un usuario no puede crear una firma falsa que parezca válida para el administrador ni puede revelar la identidad de un firmante dada una firma (salvo el administrador de grupo en caso necesario).

Si nos fijamos detenidamente, podemos observar que en los tres protocolos un atacante puede obtener la clave pública de grupo, pero puesto que es pública esto no supone una vulnerabilidad en la seguridad.

En lo referente a la solicitud de entrada al grupo, podemos comprobar que un usuario válido $\boldsymbol{U_i}$ tendrá que incluir información personal que solamente él/ella conozca (y que solamente será válida una vez) y, además, incluirá marcas temporales y firmas en sus mensajes o su propia clave pública **RSA**. Primero, debido a la inclusión de la firma o la clave pública de $\boldsymbol{U_i}$ en los mensajes, y dado que no puede ser falsificada la clave pública del administrador de grupo, la comunicación será indescifrable para el atacante puesto que solamente el administrador y cada usuario conocen sus claves privadas respectivamente. Además, la modalidad de **RSA** utilizada permite rellenar con un padding seguro los bloques cifrados y que dos encriptaciones del mismo texto plano no den como resultado el mismo texto cifrado. En segundo lugar, la inclusión de las marcas temporales asegura a cada una de las partes que su mensaje anterior ha sido leído por el destinatario adecuado (dado que ha podido descifrarlo para obtener dicha marca) y permite que no se pueda realizar un ataque por "replay" repitiendo las mismas peticiones más tarde.

En el caso particular del **Protocolo 0** hay dos puntos a tener en cuenta:

- La obtención de la clave simétrica para **AES** sigue un proceso similar al explicado para la solicitud de unión al grupo por lo que, basándonos en que solo miembros del grupo pueden generar firmas válidas, este proceso del protocolo también es seguro ante un posible atacante que no pertenezca al grupo. Por otro lado, dado que no se pone límite en

el número de claves simétricas que pueda pedir un miembro del grupo (para aumentar los posibles pseudónimos que pueda utilizar), un atacante perteneciente al grupo podría llegar a denegar el servicio si colapsase la base de datos con sus peticiones de claves simétricas (pero no podría sacar información adicional). Teniendo en cuenta que se podría identificar, denegar el acceso y penalizar legalmente al usuario que realizase este mal uso se ha considerado que este incidente tiene una probabilidad de ocurrencia y repercusión baja.

- Para acceder al sistema, el ordenador de acceso le enviará un "desafío" a la tarjeta que consistirá en una marca temporal. Esta marca temporal permite que una vez recibido el mensaje por el administrador de grupo no se pueda volver a usar el mismo mensaje para acceder al sistema impidiendo que un atacante (en caso de que lograra simular ser el ordenador seguro de acceso) pueda reutilizar dicho mensaje para acceder al servicio. En caso de que directamente se quedase con el mensaje del usuario, dado que éste no recibiría el acceso por parte del servidor podría reportarlo al momento permitiendo invalidar dicha clave simétrica y el posible uso del mensaje anterior.

Por último, para los tres protocolos hay que mencionar que el programa del ordenador seguro con lector de tarjetas interactuará con la tarjeta previa autorización de un número PIN. Si un adversario lograse averiguar dicho número y copiar la tarjeta podría utilizar el servicio haciéndose pasar por el usuario al que ha duplicado la tarjeta. Por otro lado, dado que el programa que interactúa con la smartcard no guarda ningún mensaje nos aseguramos que las firmas grupales o mensajes enviados a través de él al administrador de grupo utilizando la clave pública **RSA** del administrador solamente son conocidos por el usuario y el administrador.

3.5. Desarrollo de los protocolos

Ahora se explicarán las herramientas utilizadas para el desarrollo de los protocolos y algunas de las consideraciones tomadas durante éste.

3.5.1. Criptografía en Python y Javacard

Las funciones criptográficas utilizadas durante el desarrollo han sido **AES-256-CBC** y **RSA-2048-PKCS#1-OAEP**. En el caso de **AES** se ha utilizado tanto la implementación proporcionada por Python como por Javacard (respectivamente para el servidor y la tarjeta). También se ha considerado en la implementación realizada de los protocolos que los mensajes cifrados con **AES** (**Anexo L**) tendrían la longitud de un bloque (128 bits). A pesar de esto, se ha elegido el modo **CBC** para que el cambio sea fácil y seguro en caso de querer modificarlo en un futuro para utilizar mensajes de longitud mayor a un bloque (sin que ésta sea necesariamente múltiplo del bloque). El modo de **RSA** (**Anexo M**) escogido realiza un tratamiento previo del texto plano incluyendo un "padding" para generar aleatoriedad y que el texto se ajuste a la longitud de la clave. Con ello evitamos que el mismo texto plano produzca el mismo texto cifrado y que se puedan desencriptar partes del texto cifrado por parte de un atacante aumentando la seguridad. A la hora de utilizar este modo en Python para encriptar es necesario dividir el texto en bloques del tamaño adecuado (en función al tipo de "padding" utilizado) e irlos juntando después de haberlos encriptado. Para desencriptar basta con dividir en mensaje en bloques de 2048 bits = 256 bytes.

3.5.2. Python Flask

Python Flask **[27]** es un microframework de Python para el desarrollo de aplicaciones web. Posee numerosas extensiones que permiten personalizar la aplicación en función de las necesidades del producto que se desee desarrollar.

A la hora de desarrollar los protocolos propuestos en este trabajo se ha decidido utilizar esta tecnología para implementar el servidor que hace de administrador del grupo por la familiaridad con el lenguaje de programación Python.

Cuando se desarrolla una nueva funcionalidad de un servicio en Flask basta con añadir una etiqueta encima de la función de Python que la implementa. Esta etiqueta es ***@app.route('ruta')*** donde el parámetro ***ruta*** indicará a Flask la dirección url relativa a nuestro servidor que hará que se ejecute nuestra función. Por ejemplo:

```
@app.route('/')

def hello_world():

    return 'Hello, World!'
```

Es interesante mencionar que dentro del parámetro ***ruta*** se pueden indicar url's que dependan de argumentos variables como por ejemplo ***@app.route('/cmd/<string:enc_cmd>')*** que nos permitiría enviar un comando cifrado al servidor que luego podría obtener para descifrarlo añadiendo ***enc_cmd*** como argumento en la función que implementa la funcionalidad de la url. También se puede indicar los métodos de HTTP a los cuales va a responder dicha url (por defecto GET) como un argumento adicional de la etiqueta ***route***.

A la hora de ejecutar nuestro servidor basta con los siguientes comandos:

```
$ export FLASK_APP=myapp.py

$ flask run --host=0.0.0.0
```

Donde indicamos el archivo que contiene una instancia de nuestro servidor Flask y las funciones que implementan la funcionalidad y que queremos que nuestro servidor sea visible de forma externa. Para redes con un tráfico bajo, Flask puede servir peticiones de forma concurrente con la opción ***threaded=true***. Para un servidor en producción con un nivel de tráfico alto se puede conectar Flask con un servidor Apache (u otros) para manejar un gran volumen de peticiones concurrentes.

3.5.3. Java Card

Java Card **[28]** es una tecnología que permite el desarrollo de aplicaciones para smartcards y otros dispositivos confiables que tengan capacidades limitadas de procesamiento y almacenamiento. Permite que varias aplicaciones sean instaladas en la misma tarjeta incluso después de la entrega al usuario final, por lo que ofrece un uso flexible incorporando en cada momento las aplicaciones

que se necesiten. Cabe destacar que, a nivel de lenguaje, Java Card es un subconjunto muy reducido de Java no admitiendo tipos básicos como float, long, char o arrays de más de una dimensión. A pesar de esto, se ha elegido este lenguaje a la hora de programar las tarjetas de nuestros protocolos por la facilidad de aprendizaje, por la portabilidad de las aplicaciones entre diferentes smartcards que soporten la tecnología Java Card y por el desarrollo y mantenimiento del lenguaje que puede dar una compañía como **Oracle**.

3.5.3.1. JCIDE

JCIDE es un entorno de desarrollo integrado para Windows y DOS desarrollado y mantenido por JavacardOS **[29]** que permite implementar aplicaciones utilizando el lenguaje de programación de la tecnología Java Card. Se ha elegido esta herramienta debido a que permite depurar las aplicaciones simulando una smartcard y comunicarse con ella a través de la consola de depuración para enviar y recibir APDUs (**Anexo P**) de la aplicación. Para simular las applets desarrolladas para smartcard tenemos dos opciones en el IDE. En caso de seleccionar la opción **Default** tendremos una tarjeta **A22CR** con 138.53 KB de memoria no volátil. En caso de seleccionar **A40CR** dispondremos de 72.80 KB de memoria. Es importante tener esto en cuenta si se realizan modificaciones en el número de firmas que puede almacenar el **Protocolo 1** dentro de la tarjeta. Tener en cuenta también a la hora de crear el proyecto seleccionar una versión de Javacard 2.2.2 o superior para que no haya problemas de compatibilidad con los programas implementados en este trabajo. Una vez tengamos nuestra applet podemos simular y depurar una tarjeta que tenga dicha applet instalada. Para ello se pueden pulsar los botones **Run** (Ctrl+F8) o **Debug** (F8, primero deben marcarse los "breakpoints" en las líneas de código deseadas) del panel superior del entorno. A continuación, se abrirá una interfaz que lanzará el simulador cargando la applet en nuestra tarjeta y estará a la espera de recibir APDUs.

3.5.3.2. PyApduTool

Esta herramienta desarrollada y mantenida por JavacardOS **[29]** permite conectarse con una smartcard conectada a un lector para instalar/desinstalar applets dentro de ella y enviarle APDUs permitiendo medir el tiempo en procesar las APDUs y obtener la respuesta de la tarjeta. Una gran ventaja es que permite conectarse con la smartcard que se emula con el programa **JCIDE**. Para ello abrir **JCIDE** y navegar a **Tool−>IDE Options**. A continuación, seleccionar la pestaña **Run/Debug** y hacer click en el checkbox **Enable PCSC Interface**. Una vez que hallamos lanzado nuestra applet con **JCIDE** y esta opción marcada, basta con abrir **PyApduTool**, seleccionar el lector correspondiente (en nuestro caso JAVACOS Virtual Contact Reader 0) y pulsar el botón **Connect** (ver **Figura 3.10.**). Una vez conectados a la tarjeta podemos mandarle APDUs individuales desde la pestaña **Apdu** o ejecutar una serie de APDUs desde archivos con extensión **.src** desde la pestaña **Script**.

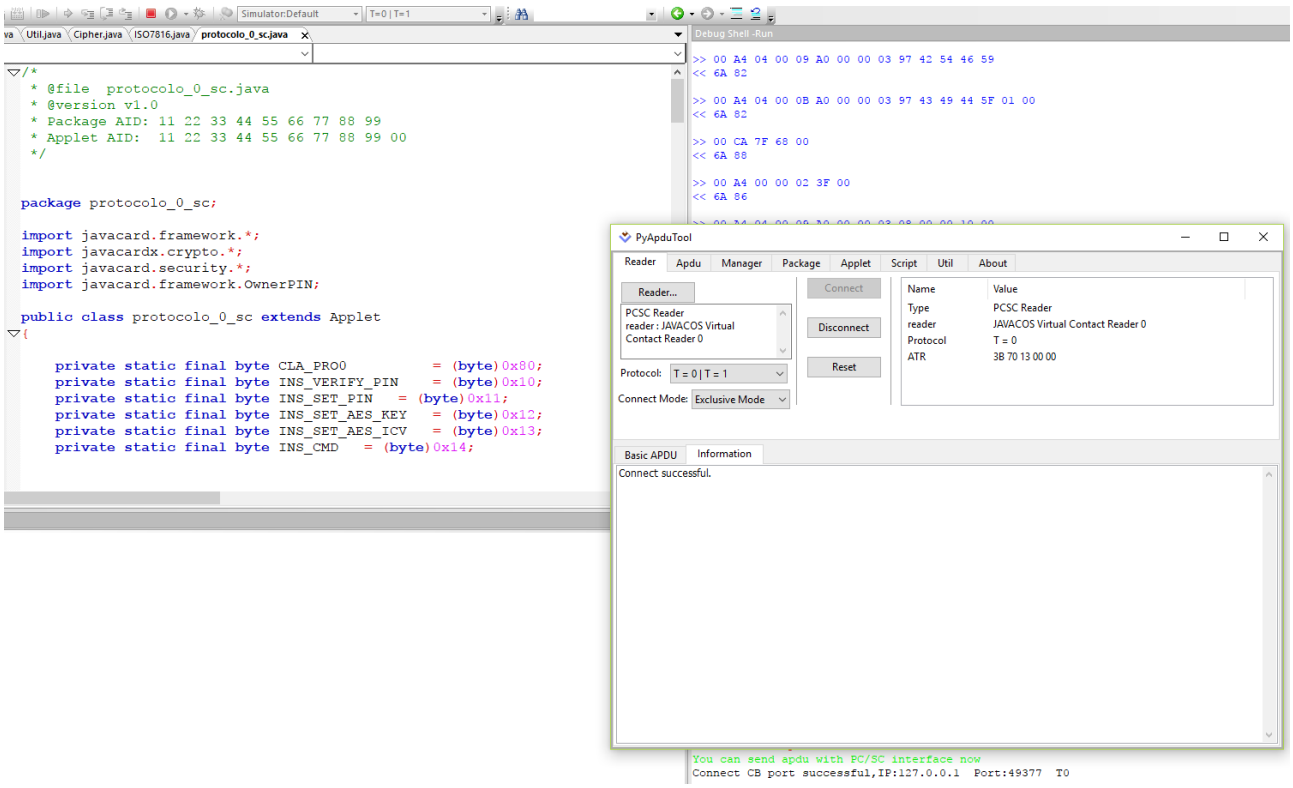

Figura 3.10. PyApduTool.

3.5.4. Libgroupsig

Libgroupsig **[17]** es una librería que implementa los esquemas de firmas grupales **KYT04** (**Anexo B.2.**), **BSS04** (**Anexo B.3.**) y **CYP06 (Anexo B.4.**). Esta librería proporciona además una serie de herramientas que permiten probar el funcionamiento de las diferentes acciones que se pueden llevar a cabo en los esquemas de firmas. La llamada a las funciones de las herramientas se ha encapsulado dentro de unos scripts de bash para permitir configurar ciertos parámetros de entrada desde el cliente y el servidor (la implementación se encuentra en el **Anexo Q**). En los siguientes apartados se describen las funcionalidades de las herramientas utilizadas.

3.5.4.1. Crear grupos

La creación del grupo de firmas se realiza mediante el programa **group_create**. La llamada a este programa se realiza de la siguiente manera:

```
$ group_create SCHEMA ENCODING -d BASEDIR -m ADMINDIR -g GROUPDIR -m MEMBERSDIR
```

Donde ***SCHEMA*** indica el esquema de firmas utilizado (**KYT04**, **BSS04**, o **CYP06**) y ***ENCODING*** indica el formato de codificación de las claves que se van a crear (**b64** o **bin**). El programa creará un directorio base con el nombre del argumento ***BASEDIR*** donde encontraremos los directorios del administrador, el grupo y los miembros llamados con el nombre indicado en sus correspondientes parámetros. Dentro del directorio del administrador encontramos el archivo

mgr.key con la clave privada del administrador del grupo y el archivo **gml** que contendrá información de las claves de los miembros. Será necesario crear también a mano un fichero vacío llamado **crl** que contendrá información sobre las claves que han sido revocadas. Dentro del directorio del grupo encontramos el archivo **grp.key** con la clave pública del grupo. Por último, el directorio de miembros contendrá la información de las claves privadas de los miembros del grupo.

3.5.4.2. Añadir miembros al grupo

Para añadir miembros al grupo y generar sus claves que les permitan firmar mensajes en nombre del grupo, se puede utilizar el programa **join** llamándolo de la siguiente manera:

```
$ join SCHEMA ENCODING GRPKFILE MGRKFILE GMLFILE MEMBERSDIRPATH NSIG
```

En este caso ***SCHEMA*** y ***ENCODING*** indican el esquema de firmas a utilizar y el formato de los ficheros de claves que se van a utilizar respectivamente. Adicionalmente, el programa recibirá la ruta hasta los ficheros **grp.key**, **mgr.key** y del directorio de miembros creados con el programa **group_create** descrito anteriormente. Por último, se le pasará el número de miembros que tendrá el grupo en el parámetro ***NSIG***. Al finalizar el programa se añadirá información sobre las claves al archivo **gml**. Dichas claves deben ser entregadas de forma segura a los miembros del grupo. Por simplicidad, el programa genera las claves completamente en el lado del servidor. Esto permitiría al servidor firmar mensajes en nombre de cualquier miembro del grupo, pero podemos realizar la suposición de que el servicio es confiable en ese sentido y sólo gestiona mensajes de usuarios reales.

3.5.4.3. Firmar mensajes

El programa utilizado para firmar mensajes se llama **sign** y se ejecuta de la siguiente manera:

```
$ sign SCHEMA ENCODING SIGFILE MSGFILE MEMKFILE GRPKFILE
```

Como en los comandos anteriores, ***SCHEMA*** y ***ENCODING*** indican el esquema de firmas a utilizar y el formato de los ficheros con los que se trabaja respectivamente. ***SIGFILE*** la ruta y nombre del fichero donde se guardará la firma y ***MSGFILE*** la ruta y nombre del fichero con el mensaje que se desea firmar. Los dos últimos parámetros indican la ruta y el nombre de los ficheros con la clave de miembro y la clave de grupo en este orden.

3.5.4.4. Verificar mensajes

El servidor que hace las veces de administrador del grupo, puede verificar que las firmas recibidas se hayan realizado con una clave de miembro del grupo mediante el programa **verify** con la siguiente llamada:

```
$ verify SCHEMA ENCODING SIGFILE MSGFILE GRPKFILE
```

De nuevo ***SCHEMA*** y ***ENCODING*** indican el esquema de firmas a utilizar y el formato de los ficheros con los que se trabaja. ***SIGFILE*** corresponderá a la ruta y nombre del fichero en el que hemos guardado la firma del mensaje y ***MSGFILE*** a la ruta y nombre del fichero con el mensaje.

Por último, ***GRPKFILE*** indicará la ruta y nombre del fichero de la clave pública del grupo. El programa devolverá "VALID signature" o "INVALID signature" dependiendo de si la firma recibida es válida para el mensaje o no.

3.5.4.5. Revocar una clave

Para revocar una clave de miembro. El servidor que administra el grupo puede utilizar la siguiente llamada al programa **revoke**:

```
$ revoke SCHEMA ENCODING SIGFILE GRPKFILE MGRKFILE GML CRL
```

Los dos primeros parámetros son iguales al resto de comandos. ***SIGFILE*** será el nombre y ruta del fichero que contenga la firma de un mensaje cuya clave se desee revocar. Por último, ***GRPKFILE***, ***MGRKFILE***, ***GML*** y ***CRL*** serán la ruta y nombre de los ficheros que contienen la clave de grupo, la clave privada de administrador, el archivo de información de claves de los miembros y el archivo con la lista de claves revocadas en este orden. Al finalizar habrá actualizado el archivo con la lista de claves revocadas.

3.5.4.6. Trazar una firma

Cuando el servidor recibe un mensaje y su firma, además de comprobar que dicha firma la ha realizado un miembro del grupo, debe comprobar que la clave con la que se ha realizado la firma no ha sido revocada anteriormente. Para ello puede usar el programa **trace** de la siguiente forma:

```
$ trace SCHEMA ENCODING SIGFILE GRPKFILE CRL MGRKFILE GML
```

Los argumentos que utiliza son los mismos que los del programa **revoke** pero en distinto orden como se puede observar. Al finalizar devolverá "VALID signer" en caso de que la clave utilizada para firmar no haya sido revocada.

3.6. De la pseudonimia a la K-Anonimia

Los tres protocolos anteriores tienen como finalidad permitir a un grupo de usuarios válidos consumir un servicio proporcionando además una funcionalidad de protección de la intimidad del usuario en distintos niveles.

El **Protocolo 0** proporciona el nivel más bajo de intimidad dado que todas las peticiones realizadas con la misma clave simétrica pueden ser relacionadas entre sí. A pesar de esto, la identidad real de los usuarios está protegida por su clave simétrica por lo que se les identifica mediante un pseudónimo que es el ID del usuario. Es interesante mencionar que dado que un usuario del grupo puede poseer varias claves simétricas podría irlas alternando dentro de la smartcard para conseguir un cierto nivel de desvinculación entre sus mensajes.

El **Protocolo 1** permite a un usuario meter mensajes firmados dentro de la smartcard de forma que para identificarse a la hora de consumir un servicio utilice uno de los mensajes dentro de la tarjeta. Este protocolo presenta la ventaja de que puede llegar a proporcionar anonimia en caso de

que, suponiendo por ejemplo 50 mensajes firmados, el usuario los renueve antes de realizar 51 peticiones al servicio (o que se utilice la variación propuesta). Esto permitiría, en un servicio donde las peticiones con la misma tarjeta no sean muchas en el mismo día, autenticarse de forma anónima si se cambian los mensajes firmados asiduamente con un programa para tal propósito. La pérdida de anonimato se produce cuando el usuario utilice más de una vez una cierta firma o en caso de ser la única persona que utilice el servicio en un punto determinado (se podría establecer una relación entre la firma al entrar y al salir). Por estas razones, se ha clasificado el nivel de protección de la intimidad de este protocolo como "**soft-anonimia**".

El **Protocolo 2** permite a la tarjeta generar de forma autónoma firmas de mensajes válidas dentro del grupo de manera que cada usuario con tarjeta se pueda autenticar de forma anónima siempre. Evidentemente, la calidad de dicha anonimia está sujeta al número de miembros del grupo. Por esta razón, en un grupo de tamaño **K** tendremos **K-anonimia**.

4. Pruebas y resultados

Dado que tanto las rutinas que se ejecutan dentro del servidor Flask como los scripts en Python del cliente se ejecutan de forma instantánea y los recursos de una smartcard son bastante limitados, se ha hecho especial hincapié en medir el tiempo que supone para la smartcard procesar cada uno de los diferentes comandos que se le envían mediante C-APDUs (ver **Anexo N**). Para ello se ha utilizado el software **PyApduTool** (ver apartado **3.5.2.2.**) que permite conectarse con el simulador **JCIDE**. Debido a que hay numerosos comandos que primero requieren validar el PIN para ejecutarse, se ha eliminado esta restricción a la hora de realizar las mediciones de cada comando individualmente. Para realizar las mediciones se ha simulado una tarjeta **A22CR** con 138.53 KB de memoria flash y 1.48 KB de memoria RAM.

En la **Tabla 4.1.** que aparece a continuación se pueden apreciar las medidas de tiempo para 500 APDUs para cada comando y la media de tiempo para una sola APDU. En dicha tabla podemos comprobar que todas las APDUs (y sus posibles variaciones en caso de que por ejemplo se modifique el tamaño de los mensajes a cifrar o a guardar dentro de la tarjeta) se ejecutan rápidamente evitando el problema de cuello de botella que podría ocasionar los reducidos recursos de la tarjeta en los diferentes protocolos. Se puede observar que la operación que más penaliza en tiempo respecto a la cantidad de datos es la de escribir dentro de la memoria no volátil de la tarjeta. Se han incluido también las medidas de tiempos para el algoritmo de encriptado **RSA** (que será útil a la hora de realizar potencias en aritmética modular a la hora de implementar el **Protocolo 2**) pudiéndose observar la diferencia de tiempos entre el algoritmo **RSA-2048** y el **RSA-CRT-2048** que optimiza los cálculos utilizando el Teorema Chino del Resto (Chinese Remainder Theorem).

APDU	Tiempo Total (s)	Nº ejecuciones	Media de tiempo de ejecución (s)
Validar PIN 4 dígitos	8,352	500	0,016704
Cambiar PIN 4 dígitos	9,718	500	0,019436
Introducir ICV 128 bits	10,218	500	0,0204036
Introducir AES Key 256 bits	14,092	500	0,028184
Cifrar AES (128 bits)	44,331	500	0,088662
Cifrar AES (256 bits)	44,968	500	0,089936
Cifrar AES (1024 bits)	48,063	500	0,096126
Introducir Mensaje (500 bit)	22,564	500	0,045128
Introducir Mensaje (1500 bit)	47,697	500	0,095394
Introducir Mensaje (2500 bit)	85,250	500	0,1705
Obtener Mensaje (500 bit)	29,800	500	0,0596
Obtener Mensaje (1500 bit)	30,734	500	0,061468
Obtener Mensaje (2500 bit)	35,041	500	0,07082
Encriptar RSA CRT 1024 bits	8,333	500	0,016666
Encriptar RSA 2048 bits (en dos APDUs)	40,615	500	0,08123
Encriptar RSA CRT 2048 bits (en dos APDUs)	25,725	500	0,05145

Tabla 4.1. Tiempo de ejecución de cada APDU en la smartcard

Para probar todas las funcionalidades que ofrecen los protocolos se puede seguir el **Anexo P** que se corresponde con el manual del usuario.

5. Conclusiones y trabajo futuro

5.1. Conclusiones

En este trabajo se ha estudiado la firma grupal como método para generar anonimia, específicamente esquemas de firmas cuya longitud no dependa de la dimensión del grupo y que permitan revocar el anonimato cuando sea necesario. También se han estudiado las posibilidades de los dispositivos denominados "smartcard" a la hora de formar parte de un sistema que garantice la privacidad de sus usuarios en diferentes grados. Para ello se ha realizado una introducción previa de los conceptos necesarios para entender qué son las smartcards y las firmas grupales.

Una vez desarrollados los conceptos de firma grupal y smartcard, se han diseñado (alrededor de ellos) tres protocolos que proporcionan a los usuarios de los sistemas que los implementen diferentes grados de control de su privacidad, en función de las necesidades del servicio, y se ha discutido su seguridad en diferentes situaciones. El **Protocolo 0** proporcionaría pseudonimia a los usuarios (ya que en todo momento el administrador debe saber con qué clave simétrica descifrar las comunicaciones) mientras que el **Protocolo 1** proporciona a los usuarios la capacidad de controlar su anonimato (si renuevan las firmas de la tarjeta antes de tener que repetirlas o se utiliza la variación propuesta serán anónimos). Finalmente, el **Protolo 2** permitiría que la tarjeta generase firmas anónimas válidas para un grupo de forma autónoma.

Partiendo del diseño, se han implementado mediante el uso de diferentes tecnologías dos de los tres protocolos, dejando constancia de los desafíos que habrá que resolver para implementar el tercero. Con las pruebas realizadas sobre la implementación, podemos concluir que los dos primeros protocolos no producen puntos de excesiva sobrecarga en el sistema. Aún a expensas de la implementación del último protocolo, podemos afirmar que es posible combinar las smartcards y las firmas grupales para ofrecer un servicio que proporcione anonimato justo y comunicación segura para sus usuarios.

5.2. Trabajo futuro

Como trabajo futuro se plantean los siguientes retos:

- Solucionar los desafíos pendientes para implementar el **Protocolo 2**.
- Explorar nuevos esquemas de firmas (basados, por ejemplo, en criptografía de curvas elípticas) y métodos de autenticación anónima que sean sencillos de programar y utilizar dentro de una smartcard.
- Automatizar todo lo posible la implementación de los protocolos para ofrecer un mejor servicio a los usuarios.
- Realizar una publicación científica sobre una implementación de los protocolos propuestos (o una mejora o alternativa) en una revista especializada.

Bibliografía

[1] Camenisch, J., Stadler, M. (1997). Efficient group signature schemes for large groups (extended abstract). In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 410–424. Springer, Heidelberg.

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

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

[4] Andreas Pfitzmann, Marit Hansen (Version v0.34 Aug. 10, 2010). A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management.

[5] David Chaum, Eugene van Heyst. (1991). Group Signatures, EUROCRYPT.

[6] Wolfgang Rankl and Wolfgang Effing (2010). Smart Card Handbook: Fourth Edition.

[7] William Stallings (2014). Cryptography and Network Security: Principles and Practice (6th Edition).

[8] 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.

[9]https://blog.cryptographyengineering.com/2014/11/27/zero-knowledge-proofs-illustrated-primer/. [Online; accessed 09-May-2017].

[10] https://cgi.csc.liv.ac.uk/~igor/COMP309/3CP.pdf. [Online; accessed 24-May-2017].

[11] http://www.brighthub.com/computing/hardware/articles/76052.aspx. [Online; accessed 24-May-2017].

[12] Transit Cooperative Research Program-Legal Research Digest 25. [Online; accessed 24-May-2017].

[13] OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data. [Online; accessed 24-May-2017].

[14] NIST-FIPS 197.Advanced Encryption Standard (AES) . [Online; accessed 24-May-2017].

[15] R.L. Rivest, A. Shamir, and L. Adleman (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.

[16] D. Chaum (1984). Blind signature systems. In Advances in Cryptology -- CRYPTO '83, page 153. Plenum Press.

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

[18] INTERNATIONAL STANDARD ISO/IEC 7816-3 Third edition (2006-11-01). Identification cards — Integrated circuit cards — Part 3: Cards with contacts — Electrical interface and transmission protocols.
[19] INTERNATIONAL STANDARD ISO/IEC 7816-4 Second edition (2005-01-15) Identification cards — Integrated circuit cards — Part 4: Organization, security and commands for interchange.

[20] D. Dolev, A. Yao (1983). On the Security of Public Key Protocols.

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

[22] Ruhul Amin, SK Hafizul Islam, G. P. Biswas, Muhammad Khurram Khan and Neeraj Kumar (2015). An Efficient and Practical Smart Card Based Anonymity Preserving User Authentication Scheme for TMIS using Elliptic Curve Cryptography.

[23] Jan HAJNÝ (2010). Anonymous Authentication for Smartcards.

[24] Michael Sterckx, Benedikt Gierlichs, Bart Preneel and Ingrid Verbauwhede (2009). Efficient implementation of anonymous credentials on Java Card smart cards.

[25] Patrik Bichsel, Jan Camenisch, Thomas Groß and Victor Shoup (2009). Anonymous Credentials on a Standard Java Card.

[26] Hajny J., Malina L. (2013) Unlinkable Attribute-Based Credentials with Practical Revocation on Smart-Cards.

[27] Flask (2017). Flask. [Online; accessed 24-May-2017].

[28] Java Card (2017). Oracle. [Online; accessed 24-May-2017].

[29] JavacardOS (2017). JavacardOS. [Online; accessed 24-May-2017].

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

[31] BOE298 (1999). Boe298. [Online; accessed 5-June-2017].

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

[33] Benjumea, V., Choi, S.G., Lopez, J., Yung, M. (2008). Fair traceable multi-group signatures. In: Tsudik, G. (ed.) FC 2008. LNCS, vol. 5143, pp. 231–246. Springer, Heidelberg.

[34] Diaz J., Arroyo D., Rodriguez F.B. (2013) Anonymity Revocation through Standard Infrastructures. In: De Capitani di Vimercati S., Mitchell C. (eds) Public Key Infrastructures, Services and Applications. EuroPKI 2012. Lecture Notes in Computer Science, vol 7868. Springer, Berlin, Heidelberg.

[35] REGLAMENTO (UE) 2016/679 DEL PARLAMENTO EUROPEO Y DEL CONSEJO de 27 de abril de 2016 relativo a la protección de las personas físicas en lo que respecta al tratamiento

de datos personales y a la libre circulación de estos datos y por el que se deroga la Directiva 95/46/CE (Reglamento general de protección de datos). [Online; accessed 15-June-2017 at https://www.boe.es/doue/2016/119/L00001-00088.pdf]

[36] Goldwasser S., Micali S. and Rackoff C. (1985). The Knowledge Complexity of Interactive Proof-System. MIT. MIT. University of Toronto. 1.

[37] J. H. Silverman (1986). The Arithmetic of Elliptic Curves, volume 106 of Graduate Texts in Mathematics. Springer-Verlag.

[38] S. Lang (1973). Elliptic Functions. Addison-Wesley, Reading, MA.

[39] D. Boneh, B. Lynn, and H. Shacham (Dec. 2001). Short signatures from the Weil pairing. In Proceedings of Asiacrypt 2001, volume 2248 of LNCS, pages 514–32. Springer-Verlag.

[40] Bellare, M., Rogaway, P (1995). Optimal asymmetric encryption—how to encrypt with RSA. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 92–111. Springer, Heidelberg.

A. Conocimientos adquiridos

Este trabajo de fin de grado me ha permitido adquirir los siguientes conocimientos:

- Funcionamiento de las pruebas de conocimiento cero.
- Funcionamiento de las firmas digitales y de los avances sobre ellas como la firma ciega y la firma grupal.
- Funcionamiento de la librería de firmas grupales **libgroupsig**.
- Funcionamiento de las tecnologías Python Flask y Javacard.
- Base matemática de las suposiciones en las que se basan los esquemas de firmas grupales como **Decisional Diffie- Hellman** y **Linear Diffie-Hellman**, entre otros.
- Conocimiento de la estructura y el funcionamiento general de una smartcard y de cómo utilizarla dentro de una aplicación o servicio.
- Búsqueda, lectura y comparación de artículos científicos.

Este trabajo me ha resultado muy interesante ya que me ha permitido profundizar en el campo de la seguridad informática y la criptografía (al que veo mucho potencial en el futuro) utilizando además dispositivos que tenemos al alcance fácilmente como son las smartcards. También he podido estudiar la base matemática que hay detrás de algunos esquemas de firmas grupales, permitiéndome utilizar y combinar conocimientos de los dos grados que realizo.

B. Esquemas específicos de firmas grupales

A continuación, se explican cuatro esquemas de firmas grupales. Estos sistemas hacen uso de varias pruebas de conocimiento cero para garantizar la anonimia de los usuarios dentro del grupo. Las pruebas de conocimiento cero de los esquemas consistirán en una serie de operaciones matemáticas para obtener ciertos valores que constituyen un problema complejo (**logaritmo discreto, DDH, LDH**, etc.) y que incluyen algún tipo de clave (oculta utilizando valores aleatorios y operaciones que se puedan invertir en caso de ser necesario obtener la clave) perteneciente a un miembro del grupo. El administrador podrá comprobar que los valores recibidos constituyen una solución correcta al problema y, además, no podrá descubrir directamente con qué clave se han realizado las pruebas que recibe a no ser que realice las operaciones pertinentes para revelarla (en caso de que fuera necesario). En algunos de estos esquemas también se puede introducir la clave relevada en una lista de revocación para que los siguientes mensajes firmados con dicha clave no sean válidos (**trazabilidad**).

B.1. Camenisch 1997 (CS97)

En 1997 Jan Camenisch propone varios esquemas de firmas cuya la longitud de clave grupal y esfuerzo computacional de firmar y verificar una firma es independiente de la dimensión del grupo **[1]**. Esto supone una mejora importante respecto a la mayoría de los anteriores esquemas donde los parámetros antes mencionados crecían de forma lineal con la dimensión del grupo. Además, añadir nuevos miembros al grupo no modifica la clave grupal. Para conseguirlo se basa en las propiedades del logaritmo (**Anexo F**) y del doble logaritmo discreto para construir lo que denomina como "firmas de conocimiento" (dado que sirven como prueba de conocimiento cero y como firma). Para el propósito de este trabajo se ha estudiado el esquema más eficiente de los propuestos en la referencia que se explica a continuación con más detalle.

Consideramos una función hash $\boldsymbol{H}$ resistente a las colisiones de una dimensión de salida aproximada de 160 bits, un grupo cíclico (grupo tal que existe un elemento $\boldsymbol{g}$ a partir del cual pueden expresarse todos los miembros del grupo como potencia de él) $\boldsymbol{G} =< \boldsymbol{g} >$ de orden $\boldsymbol{n}$ (el número de elementos del grupo) en el cual es costoso computacionalmente calcular el logaritmo discreto de un número y un elemento de dicho grupo $\boldsymbol{h}$ cuyo logaritmo discreto en la base $\boldsymbol{g}$ es desconocido. Una manera sencilla de convencer a un verificador del conocimiento de una raíz n-ésima $\boldsymbol{e}$ del logaritmo discreto de un número $\boldsymbol{z} = \boldsymbol{g}^{\boldsymbol{x}^{\boldsymbol{e}}}$ es (si ésta es pequeña) mediante el cálculo de los valores $\boldsymbol{z_i} = \boldsymbol{g}^{\boldsymbol{x}^{\boldsymbol{i}}}$ con $\boldsymbol{i}$ entre $\mathbf{1}$ y $\boldsymbol{e} - \mathbf{1}$. Para no revelar información sobre $\boldsymbol{z}$ ni ninguno de los $\boldsymbol{z_i}$ calculados consideramos la firma denominada $\boldsymbol{E - SKROOTREP}$. Esta firma demuestra el conocimiento de dos valores $(\boldsymbol{a}, \boldsymbol{b})$ cumpliendo que $\boldsymbol{h}^{\boldsymbol{a}}\boldsymbol{g}^{\boldsymbol{b}^{\boldsymbol{e}}}$ sin revelar dichos valores. La firma consiste en lo siguiente:
$\boldsymbol{E - SKROOTREP}\left[(\boldsymbol{a}, \boldsymbol{b}): \boldsymbol{v} = \boldsymbol{h}^{\boldsymbol{a}}\boldsymbol{g}^{\boldsymbol{b}^{\boldsymbol{e}}}\right](\boldsymbol{m}) = \{(\boldsymbol{v_1}, \dots, \boldsymbol{v_{e-1}})\}$ con los $\boldsymbol{v_i} = \boldsymbol{h}^{\boldsymbol{r_i}}\boldsymbol{g}^{\boldsymbol{x}^{\boldsymbol{i}}}$ eligiendo cada $\boldsymbol{r_i}$ de forma aleatoria en $\boldsymbol{Z_n}$ y con $\boldsymbol{r} = \boldsymbol{r_1}\boldsymbol{r_2} \dots \boldsymbol{r_{e-1}}$. De esta manera conseguimos que los $\boldsymbol{v}_i$ sean elementos aleatorios de $\boldsymbol{G}$ y no revelen información sobre $\boldsymbol{z}$ ni los $\boldsymbol{z}_i$.

Basándose en la firma $\boldsymbol{E - SKROOTREP}$ se puede construir una firma eficiente (que llamaremos $\boldsymbol{E - SKROOTLOG}$) para demostrar el conocimiento de una raíz n-ésima $\boldsymbol{d}$ del logaritmo de un número $\boldsymbol{z}$ cuya base es conocida. La firma $\boldsymbol{E - SKROOTLOG}\left[\boldsymbol{d}: \boldsymbol{z} = \boldsymbol{g}^{\boldsymbol{d}^{\boldsymbol{e}}}\right]$ consistirá de lo siguiente:

- La firma $\boldsymbol{E - SKROOTREP}\left[(\boldsymbol{a}, \boldsymbol{b}): \boldsymbol{z} = \boldsymbol{h}^{\boldsymbol{a}}\boldsymbol{g}^{\boldsymbol{b}^{\boldsymbol{e}}}\right](\boldsymbol{m})$
- La firma $\boldsymbol{SKREP}\left[\boldsymbol{k}: \boldsymbol{z} = \boldsymbol{g}^{\boldsymbol{k}}\right](\boldsymbol{m})$. Que demuestra el conocimiento del logaritmo discreto de $\boldsymbol{z}$ en la base $\boldsymbol{g}$.

Dado que el logaritmo discreto de $\boldsymbol{h}$ es desconocido en la base $\boldsymbol{g}$ y solamente se puede conocer una representación de $\boldsymbol{z}$ en las bases de $\boldsymbol{h}$ y $\boldsymbol{g}$ esto supone que $\boldsymbol{a} \equiv \mathbf{0}\ \boldsymbol{mod}\ \boldsymbol{n}$ y por tanto $\boldsymbol{k} \equiv \boldsymbol{b}^{\boldsymbol{e}}\boldsymbol{mod}\ \boldsymbol{n}$ demostrándose así que el probador conoce una raíz n-ésima del logaritmo discreto de $\boldsymbol{z}$ en la base $\boldsymbol{g}$.

Una vez tenemos estas firmas de conocimiento podemos generar el grupo de la siguiente forma:

Se elige un número $\boldsymbol{n}$ que será el módulo **RSA** y dos claves públicas $\boldsymbol{e_1}$ y $\boldsymbol{e_2}$ siendo la última primo relativo de $\boldsymbol{\varphi(n)}$ (la función phi de Euler que devuelve como resultado el número de enteros menores que $\boldsymbol{n}$ y coprimos con él). Se eligen también dos enteros $\boldsymbol{t_1}$ y $\boldsymbol{t_2}$ mayores que $\mathbf{1}$ cuyas raíces $\boldsymbol{e_i} - \boldsymbol{ésimas}$ no pueden ser calculadas sin conocer la factorización de $\boldsymbol{n}$. Tendremos un grupo cíclico $\boldsymbol{G}$ y un elemento $\boldsymbol{h}$ como los mencionados anteriormente. La clave pública del gestor del grupo será $\boldsymbol{PU_{gm}} = \boldsymbol{h}^{\boldsymbol{w}}$ para cierto $\boldsymbol{w}$ aleatorio en $\boldsymbol{Z_n}$ y la clave privada estará formada por $\boldsymbol{w}$ y la factorización de $\boldsymbol{n}$. Por otro lado, los elementos que formarán la clave pública grupal serán $\boldsymbol{PU_{gr}} = (\boldsymbol{G}, \boldsymbol{h}, \boldsymbol{n}, \boldsymbol{PU_{gm}}, \boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{t_1}, \boldsymbol{t_2}, \boldsymbol{g})$.

En caso de querer unirse al grupo, un miembro $\boldsymbol{M}$ realizará lo siguiente:

$\boldsymbol{M}$ tomará $\boldsymbol{PU_{gr}}$ y calculará su clave de miembro $\boldsymbol{y_M} = \boldsymbol{x}^{\boldsymbol{e_1}}\ \boldsymbol{mod}\ \boldsymbol{n}$ para cierto $\boldsymbol{x}$ aleatorio de las unidades de $\boldsymbol{Z_n}$ $(\boldsymbol{Z_n^*})$. Para evitar que el administrador del grupo conozca las claves de los miembros y pueda firmar mensajes en su nombre se puede utilizar el sistema de firmas ciegas **RSA [16]**. De esta forma el miembro $\boldsymbol{M}$ enviará lo siguiente al administrador:

- $\boldsymbol{z_M} = \boldsymbol{g}^{\boldsymbol{y_M}}$.
- $\boldsymbol{y_M^*} = \boldsymbol{r}^{\boldsymbol{e_2}}(\boldsymbol{t_1}\boldsymbol{y_M} + \boldsymbol{t_2})\ \boldsymbol{mod}\ \boldsymbol{n}$ para $\boldsymbol{r}$ aleatorio en $\boldsymbol{Z_n^*}$ donde vemos que $\boldsymbol{y}$ queda oculto.

- $\boldsymbol{F1 = E - SKROOTLOG}\left[x: z = g^{x^{e_1}}\right](')$ donde $'$ es el mensaje vacío. Esta firma asegura el conocimiento de una raíz $\boldsymbol{e_1 - ésima}$ del logaritmo discreto de $\boldsymbol{z}$ en la base $\boldsymbol{g}$.
- $\boldsymbol{F2 = E - SKROOTLOG}\left[r: g^{y_M^*} = (z^{t_1} g^{t_2})^{r^{e_2}}\right](')$.Esta firma asegura que $\boldsymbol{y_M^*}$ está bien formado y por lo tanto el valor de la clave de miembro se ha ocultado de forma correcta.

El administrador de grupo comprobará que $\boldsymbol{F1}$ y $\boldsymbol{F2}$ son correctas y calculará $\boldsymbol{v_M^* = y_M^{*\frac{1}{e_2}}}$ para enviarlo a $\boldsymbol{M}$ de forma que éste podrá obtener su certificado $\boldsymbol{v_M = (t_1 y_M + t_2)^{\frac{1}{e_2}}}$ multiplicándolo por el inverso multiplicativo de $\boldsymbol{r}$ en $\boldsymbol{Z_n}$.

En caso de querer enviar un mensaje al grupo, un miembro $\boldsymbol{M}$ realizará lo siguiente:

Un mensaje ($\boldsymbol{m}$) firmado en nombre del grupo debe poder ser verificado por el administrador del grupo. Para ello es necesario que $\boldsymbol{M}$ convenza al administrador de que tiene una clave de miembro que ha sido certificada. Para demostrarlo generará una firma $\boldsymbol{F}$ que consistirá de lo siguiente:

- $\boldsymbol{z_M^* = h^r g^y}$ para **r** aleatorio en $\boldsymbol{Z_n^*}$.
- $\boldsymbol{d = PU_{gm}^r}$.
- $\boldsymbol{S1 = E - SKROOTREP}\left[(a_1, b_1): z^* = h^{a_1} g^{b_1^{e_1}}\right](\boldsymbol{m})$. Que demuestra que se conoce una representación de $\boldsymbol{z}$ en las bases $\boldsymbol{g}$ y $\boldsymbol{h}$ y una raíz $\boldsymbol{e_1 - ésima}$ de la parte $\boldsymbol{g}$ de $\boldsymbol{z}$.
- $\boldsymbol{S2 = E - SKROOTREP}\left[(a_2, b_2): z^{*t_1} g^{t_2} = h^{a_2} g^{b_2^{e_2}}\right](\boldsymbol{m})$. Que demuestra que se conoce una representación de $\boldsymbol{z^{*t_1} g^{t_2}}$ en las bases $\boldsymbol{g}$ y $\boldsymbol{h}$ y una raíz $\boldsymbol{e_2 - ésima}$ de la parte $\boldsymbol{g}$ de $\boldsymbol{z}$.
- $\boldsymbol{S3 = SKREP}\,[(a_3, b_3): d = PU_{gm}^{a_3}, z = h^{a_3} h^{b_3}]\,(\boldsymbol{m})$. Que garantiza que el par $(\boldsymbol{d}, \boldsymbol{z_M^*})$ está formado correctamente y así, en caso de ser necesario abrir la firma del mensaje, el administrador calculará $\boldsymbol{z_M^{**} = \frac{z_M^*}{d^{\frac{1}{w}}} = z_M}$.

Dado que $\boldsymbol{M}$ solamente puede conocer una representación de $\boldsymbol{z^{*t_1} g^{t_2}}$ en las bases $\boldsymbol{g}$ y $\boldsymbol{h}$ tenemos que $\boldsymbol{a_2 \equiv a_1 \bmod n}$ y que $\boldsymbol{b_2^{e_2} \equiv (t_1 b_1^{e_1} + t_2) \bmod n}$ y, por tanto, al conocer una raíz $\boldsymbol{e_1 - ésima}$ para $\boldsymbol{b_1^{e_1}}$ y una raíz $\boldsymbol{e_2 - ésima}$ para $\boldsymbol{b_2^{e_2}}$ demuestra que conoce una clave de miembro secreta que ha sido certificada. Este esquema de firmas grupales es de especial interés en este trabajo ya que, si bien no proporciona las firmas más cortas (aproximadamente 1,4KBytes), es relativamente sencillo de comprender e implementar y la longitud de la firma es asequible para una smartcard.

B.2. KTY04

KTY04 **[2]** es un esquema de firmas revocables basado en las suposiciones del **logaritmo discreto**, **Strong RSA** y **DDH** (**Anexos F**, **K** y **H** respectivamente). A continuación, se explican brevemente las acciones necesarias para cada una de las funcionalidades del esquema.

En primer lugar se introduce la notación de esfera $\boldsymbol{S(a, b)}$ que representa el conjunto de enteros entre $\boldsymbol{a - b + 1}$ y $\boldsymbol{a + b - 1}$ incluidos. Para una esfera $\boldsymbol{S(2^x, 2^y)}$ se considera la esfera interior $\boldsymbol{S_L^k(2^x, 2^y) = S(2^x, 2^{\frac{y-2}{L} - k})}$.

Para generar el grupo (**setup**) el administrador realizará lo siguiente:

- Generar los parámetros $\boldsymbol{k}$ (número natural) y $\boldsymbol{L} > 1$ (real) y dos primos $\boldsymbol{p'}$ y $\boldsymbol{q'}$ tales que $\boldsymbol{p} = \boldsymbol{2p'} + \boldsymbol{1}$ y $\boldsymbol{q} = \boldsymbol{2q'} + \boldsymbol{1}$ también sean primos.
- Se calculará un módulo **RSA** $\boldsymbol{n} = \boldsymbol{pq}$ y seis valores de forma aleatoria $\boldsymbol{a}$, $\boldsymbol{a_0}$, $\boldsymbol{b}$, $\boldsymbol{y}$, $\boldsymbol{g}$, $\boldsymbol{h}$, pertenecientes a los residuos cuadráticos módulo $\boldsymbol{n}$ ($\boldsymbol{QR}(\boldsymbol{n})$) de orden $\boldsymbol{p'q'}$. Además, se calculan tres esferas $\boldsymbol{S_1}$, $\boldsymbol{S_2}$, y $\boldsymbol{S_3}$ que deben cumplir unas propiedades específicas detalladas en la referencia **[2]** junto con las esferas interiores $\boldsymbol{S_{1_L^k}}$ y $\boldsymbol{S_{3_L^k}}$.
- La clave privada $\boldsymbol{PR_{gm}}$ será el par $\boldsymbol{p}$, $\boldsymbol{q}$ mientras que la clave pública $\boldsymbol{PU_{gm}}$ será la tupla $(\boldsymbol{n}, \boldsymbol{a}, \boldsymbol{a_0}, \boldsymbol{b}, \boldsymbol{y}, \boldsymbol{g}, \boldsymbol{h})$.

Para unirse al grupo (**join**) un usuario $\boldsymbol{U_i}$ y el administrador realizarán el siguiente protocolo:

- Este protocolo implica obtener una representación de una potencia aleatoria $\boldsymbol{x_i'}$ de $\boldsymbol{b}$ perteneciente a $\boldsymbol{S_{1_L^k}}$. Al final de este protocolo $\boldsymbol{U_i}$ recibirá $\boldsymbol{x_i'}$ y el administrador $\boldsymbol{C_i} = \boldsymbol{b^{x_i'}}$.
- Posteriormente el administrador calculará un número primo aleatorio $\boldsymbol{e_i}$ perteneciente a $\boldsymbol{S_{3_L^k}}$ y $\boldsymbol{x_i}$ perteneciente a $\boldsymbol{S_{1_L^k}}$ y enviará a $\boldsymbol{U_i}$ la tupla $(\boldsymbol{A_i}, \boldsymbol{e_i}, \boldsymbol{x_i})$ con $\boldsymbol{A_i} = (\boldsymbol{C_i a^{x_i} a_0})^{e_i^{-1}} \boldsymbol{mod\ n}$. El certificado $\boldsymbol{cert_i}$ estará formado por la tupla $(\boldsymbol{A_i}, \boldsymbol{e_i}, \boldsymbol{x_i}, \boldsymbol{x_i'})$. Dado que $\boldsymbol{A_i}^{\boldsymbol{e_i}} = \boldsymbol{b^{x_i'}\, a^{x_i} a_0}$ el certificado es una representación del logaritmo discreto de una potencia arbitraria y el administrador podrá utilizar la parte $\boldsymbol{x_i}$ que conoce en caso necesario para trazar la firma.

Para identificarse demostrando el conocimiento de un certificado (sin revelar dicho certificado) el usuario $\boldsymbol{U_i}$ realizará el siguiente proceso:

- Calculará los valores $\boldsymbol{T_1} = \boldsymbol{A_i y^r}, \boldsymbol{T_2} = \boldsymbol{g^r}, \boldsymbol{T_3} = \boldsymbol{g^{e_i} h^r}, \boldsymbol{T_4} = \boldsymbol{g^{x_i k}}, \boldsymbol{T_5} = \boldsymbol{g^k}, \boldsymbol{T_6} = \boldsymbol{g^{x_i' k'}}, \boldsymbol{T_7} = \boldsymbol{g^{k'}}$ con $\boldsymbol{r}$, $\boldsymbol{k}$ y $\boldsymbol{k'}$ aleatorios pertenecientes a $\boldsymbol{S_2}$ y calculará una prueba de conocimiento cero a partir de dichos valores que permite demostrar el conocimiento de un certificado de la forma $\boldsymbol{A_i}^{\boldsymbol{e_i}} = \boldsymbol{b^{x_i'}\, a^{x_i} a_0}$.

En caso de necesitar obtener la identificación del usuario de una firma (**open**) el administrador podrá obtener los valores $\boldsymbol{T_1}$ a $\boldsymbol{T_7}$ correspondientes a la prueba de conocimiento de dicho certificado y calcular $\boldsymbol{A} = \boldsymbol{T_2}^{-x} \boldsymbol{T_1}$. Con este valor buscará en la tabla de transcripciones de unión al grupo el $\boldsymbol{A_i}$ que sea igual a $\boldsymbol{A}$.

En caso de querer saber si determinada identificación pertenece a un usuario en concreto, el administrador puede obtener el valor $\boldsymbol{x_i}$ del usuario $\boldsymbol{U_i}$ de la transcripción de su protocolo de unión al grupo (**reveal**). Una vez tiene este valor puede calcular a partir de la identificación los valores $\boldsymbol{T_1}$ a $\boldsymbol{T_7}$ y comprobar si $\boldsymbol{T_5}^{\boldsymbol{x_i}} = \boldsymbol{T_4}$ que será cierto de haber utilizado el $\boldsymbol{x_i}$ correcto.

Por último, un usuario $\boldsymbol{U_i}$ puede demostrar ser el autor de una identificación demostrando el conocimiento del logaritmo discreto de $\boldsymbol{T_6}$ en la base $\boldsymbol{T_7}$ (para $\boldsymbol{T_6}$ y $\boldsymbol{T_7}$ pertenecientes a la transcripción de dicha identificación) dado que conoce $\boldsymbol{x_i'}$.

Este esquema genera firmas de una longitud de unos 1206 bytes que es similar a **Camenish 97** pero es "grande" teniendo en cuenta que una firma con **RSA** de 4096 bits sería de 512 bytes. Presenta la ventaja de que la creación del grupo es un proceso relativamente poco costoso para el administrador del grupo.

B.3. BBS04

BBS04 **[3]** es un esquema de firma grupal que para construir sus pruebas de conocimiento cero se basa en los problemas **q-SDH** y **LDH** (**Anexos J** e **I** respectivamente). Para utilizar este esquema de firmas se consideran dos grupos bilineales (**Anexo C**) $\boldsymbol{G_1}$ y $\boldsymbol{G_2}$ con generadores $\boldsymbol{g_1}$ y $\boldsymbol{g_2}$ respectivamente y un operador bilineal $\boldsymbol{e}$. Se asume además que **LDH** se cumple en $\boldsymbol{G_1}$ y **SDH** se cumple en $(\boldsymbol{G_1}, \boldsymbol{G_2})$. También se considera la existencia de una función hash $\boldsymbol{H}: \{\boldsymbol{0,1}\}^* \rightarrow \boldsymbol{Z_p^*}$ resistente a las colisiones. Para conseguir disminuir la longitud de las firmas se utiliza criptografía de curvas elípticas **[37, 38, 39]**. A modo de breve resumen, se utiliza una curva elíptica para construir una representación de menor tamaño de los elementos de un grupo de orden $\boldsymbol{p}$ (siendo $\boldsymbol{p}$ un número primo de gran tamaño) en el que resolver el problema de **CDH** sea complejo (**Anexo G**) pero se puede resolver el problema de **DDH** fácilmente (**Anexo H**).

Para generar las claves de grupo el administrador realizará lo siguiente:

- Primero selecciona $\boldsymbol{n}$ que indica el número de miembros del grupo.
- Calcula $\boldsymbol{h}$ elegido aleatoriamente entre los elementos de $\boldsymbol{G_1} \setminus \{\boldsymbol{1_{G_1}}\}$ y $\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{y_1}$ de forma aleatoria entre las unidades de $\boldsymbol{Z_p}$ ($\boldsymbol{Z_p^*}$, elementos con inverso en $\boldsymbol{Z_p}$). A partir de estos valores calcula $\boldsymbol{u}, \boldsymbol{v}$ pertenecientes a $\boldsymbol{G_1}$ para que cumplan $\boldsymbol{u^{x_1} = v^{x_2} = h}$ y $\boldsymbol{w = g_2^{y_1}}$. La clave pública de grupo será la tupla $\boldsymbol{PU_{gr} = (g_1, g_2, h, u, v, w)}$ mientras que la clave privada del administrador será el par $(\boldsymbol{x_1, x_2})$.
- Para cada usuario $\boldsymbol{U_i}$ generará un par $(\boldsymbol{A_i, x_i})$ con $\boldsymbol{x_i}$ aleatorio elegido entre los elementos de $\boldsymbol{Z_p^*}$ y $\boldsymbol{A_i = g_1^{1/(y_1 + x_i)}}$. Este par será la clave privada de cada usuario.

Para firmar un mensaje $\boldsymbol{M}$ un usuario $\boldsymbol{U_i}$ realizará lo siguiente:

- Calcular $\boldsymbol{T_1 = u^a}$, $\boldsymbol{T_2 = v^b}$ y $\boldsymbol{T_3 = A_i h^{a+b}}$ para $\boldsymbol{a}$ y $\boldsymbol{b}$ aleatorios en $\boldsymbol{Z_p}$ lo que supone una encriptación lineal de $\boldsymbol{A_i}$. Además, calcula dos valores auxiliares $\boldsymbol{d_1 = x_i a, d_2 = x_i b}$.
- $\boldsymbol{U_i}$ genera ahora los valores $\boldsymbol{r_a, r_b, r_{x_i}, r_{d_1}, r_{d_2}}$ de forma aleatoria entre los elementos de $\boldsymbol{Z_p}$ y los utiliza para ocular los valores de $\boldsymbol{a, b, x_i, d_1}$ y $\boldsymbol{d_2}$. Esto lo consigue calculando los valores $\boldsymbol{R_1 = u^{r_a}, R_2 = v^{r_b}, R_3 = e(T_3, g_2)^{r_{x_i}} * e(h, w)^{-r_a - r_b} * e(h, g_2)^{-r_{d_1} - r_{d_2}}}$, $\boldsymbol{R_4 = T_1^{r_{x_i}} u^{-r_{d_1}}}$ y $\boldsymbol{R_5 = T_2^{r_{x_i}} v^{-r_{d_2}}}$. Una vez calculados estos valores los enviará al administrador que obtendrá el valor $\boldsymbol{c = H(M, T_1, T_2, T_3, R_1, R_2, R_3, R_4, R_5)}$ y se lo enviará a $\boldsymbol{U_i}$. Entonces el usuario calculará $\boldsymbol{s_a = r_a + ca, s_b = r_b + cb, s_{d_1} = r_{d_1} + cd_1, s_{d_2} = r_{d_2} + cd_2}$, $\boldsymbol{s_{x_i} = r_{x_i} + cx_i}$. La firma será la tupla formada por los valores $\boldsymbol{sign_i^M = (T_1, T_2, T_3, c, s_a, s_b, s_{x_i}, s_{d_1}, s_{d_2})}$.

Para verificar una firma el administrador comprobará que se cumple que $\boldsymbol{c = H(M, T_1, T_2, T_3, R'_1, R'_2, R'_3, R'_4, R'_5)}$ con $\boldsymbol{R'_1 = u^{s_a}/T_1^{\,c}}$, $\boldsymbol{R'_2 = v^{s_b}/T_2^{\,c}}$, $\boldsymbol{R'_3 = e(T_3, g_2)^{s_{x_i}} * e(h, w)^{-s_a - s_b} * e(h, g_2)^{-s_{d_1} - s_{d_2}} * \left(\frac{e(T_3, w)}{e(g_1, g_2)}\right)^c}$, $\boldsymbol{R'_4 = T_1^{s_{x_i}} u^{-s_{d_1}}}$ y $\boldsymbol{R_5 = T_2^{s_{x_i}} v^{-s_{d_2}}}$.

En caso de necesitar identificar el autor de una firma $\boldsymbol{sign_i^M}$ para un mensaje $\boldsymbol{M}$ el administrador realizará lo siguiente:

- Verificará que para la firma $\boldsymbol{sign_i^M}$ corresponde con el mensaje $\boldsymbol{M}$.
- Recuperará el valor $\boldsymbol{A = \frac{T_3}{T_1^{x_1} T_2^{x_2}}}$.
- Recorrerá toda la lista de $\boldsymbol{A_i}$ hasta encontrar aquel igual a $\boldsymbol{A}$.

Este esquema presenta la ventaja de que produce firmas de aproximadamente 192 bytes con la desventaja de que para descubrir todos los mensajes de un determinado usuario que haya incumplido las normas de uso es necesario desvelar los autores de todos los mensajes hasta la fecha (ya que hay que iterar en la tabla de $\boldsymbol{A_i}$ perdiendo entonces la anonimia).

B.4. CYP06

CPY06 **[30]** es un esquema de firmas grupales cuyas pruebas de conocimiento cero se basan en las suposiciones de **q-SDH** y **LDH** (**Anexos J** e **I** respectivamente). Se trata de una extensión del esquema **BSS04** del apartado anterior para incluir la funcionalidad de poder trazar una firma (es decir, comprobar si se ha creado con una clave de miembro revocada). Supone un avance respecto a otros esquemas de firmas de este tipo ya que la longitud de la firma es considerablemente menor (salvo la de **BSS04** teniendo en cuenta que este esquema no permite trazar una firma). Del mismo modo que en el esquema **BSS04**, se consideran dos grupos bilineales $\boldsymbol{G_1}$ y $\boldsymbol{G_2}$ con generadores $\boldsymbol{g_1}$ y $\boldsymbol{g_2}$ respectivamente y un operador bilineal $\boldsymbol{e}$. Se asume que **LDH** se cumple en $\boldsymbol{G_1}$ y **SDH** se cumple en $(\boldsymbol{G_1}, \boldsymbol{G_2})$ y la existencia de una función hash $\boldsymbol{H}: \{\boldsymbol{0,1}\}^* \rightarrow \boldsymbol{Z_p}$ resistente a las colisiones. Además, $\boldsymbol{p}$ será un entero de orden 2^{170} y $\boldsymbol{G_1}$, $\boldsymbol{G_2}$ serán subgrupos de orden $\boldsymbol{p}$. Por otro lado, $\boldsymbol{G_T}$ será un subgrupo de orden $\boldsymbol{p}$ de un cuerpo finito de orden 2^{1024}. Este esquema se basa en las mismas técnicas de curvas elípticas que el anterior para conseguir acortar la longitud de las firmas. A continuación, se describen las acciones necesarias para crear el grupo con dicho esquema.

Para generar las claves de grupo el administrador ($\boldsymbol{GM}$) realizará lo siguiente:

- Construir la tupla $\boldsymbol{G} = (\boldsymbol{p}, \boldsymbol{e}, \boldsymbol{G_1}, \boldsymbol{G_2}, \boldsymbol{G_T}, \boldsymbol{g_1}, \boldsymbol{g_2})$.
- El administrador del grupo calculará un tres elementos aleatorio de $\boldsymbol{Z_p^*}$ denominados $\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{y_{GM}}$, un elemento aleatorio $\boldsymbol{Q}$ de $\boldsymbol{G_1}$, un elemento aleatorio $\boldsymbol{W}$ de $\boldsymbol{G_2} \setminus \{\boldsymbol{1_{G_2}}\}$ y un elemento aleatorio $\boldsymbol{S}$ de $\boldsymbol{G_1} \setminus \{\boldsymbol{1_{G_1}}\}$. Después, calculará $\boldsymbol{R} = \boldsymbol{y_{GM} g_2}$, $\boldsymbol{X_1} = \boldsymbol{x_1^{-1} S}$ y $\boldsymbol{X_2} = \boldsymbol{x_2^{-1} S}$. La clave pública del grupo será la tupla $(\boldsymbol{G, Q, R, W, S, X_1, X_2})$ y la clave privada del $\boldsymbol{GM}$ será $(\boldsymbol{y_{GM}, x_1, x_2})$.

A la hora de unirse al grupo se realizará un intercambio seguro de la siguiente información entre un usuario $\boldsymbol{U_i}$ y el $\boldsymbol{GM}$:

- $\boldsymbol{U_i}$ generará un número aleatorio $\boldsymbol{r_i}$ y enviará al $\boldsymbol{GM}$ $\boldsymbol{r_i g_1}$.
- El $\boldsymbol{GM}$ calculará $\boldsymbol{t_i}$ aleatoriamente en $\boldsymbol{Z_p^*}$ y enviará a $\boldsymbol{U_i}$ la tupla $(\boldsymbol{i, A_i, t_i})$ con $\boldsymbol{A_i} = \frac{\boldsymbol{1}}{\boldsymbol{y_{GM} + t_i}}(\boldsymbol{r_i g_1 + Q})$. Además guardará $(\boldsymbol{C_i} = \boldsymbol{r_i g_1, A_i, t_i})$.
- $\boldsymbol{U_i}$ comprobará que $\boldsymbol{e(A_i, t_i g_2 + R) = e(r_i g_1 + Q, g_2)}$ y si así guardará su clave privada $\boldsymbol{r_i}$ y el certificado $\boldsymbol{cert_i} = (\boldsymbol{A_i, t_i})$.

Para firmar un mensaje $\boldsymbol{M}$ un usuario $\boldsymbol{U_i}$ realizará lo siguiente:

- Calcular $\boldsymbol{T_1 = u_1 X_1}$, $\boldsymbol{T_2 = u_2 X_2}$, $\boldsymbol{T_3 = A_i + (u_1 + u_2) S}$, , $\boldsymbol{T_4 = u_3 W}$, $\boldsymbol{T_5 = e(g_1, T_4)^{r_i}}$ para $\boldsymbol{u_1}$, $\boldsymbol{u_2}$ y $\boldsymbol{u_3}$ aleatorios en $\boldsymbol{Z_p}$. De este modo, $\boldsymbol{T_1}$, $\boldsymbol{T_2}$ y $\boldsymbol{T_3}$ forman una encriptación lineal de $\boldsymbol{A_i}$. Además, calcula dos valores auxiliares $\boldsymbol{d_1 = r_i u_1}, \boldsymbol{d_2 = r_i u_2}$.
- $\boldsymbol{U_i}$ genera ahora los valores $\boldsymbol{b_{u_1}}$, $\boldsymbol{b_{u_2}}$, $\boldsymbol{b_{d_1}}$, $\boldsymbol{b_{d_2}}$, $\boldsymbol{b_{r_i}}$ y $\boldsymbol{b_{t_i}}$ de forma aleatoria entre los elementos de $\boldsymbol{Z_p}$ y con ellos calcula:
  $\boldsymbol{B_1 = b_{u_1} X_1}$, $\boldsymbol{B_2 = b_{u_2} X_2}$, $\boldsymbol{B_3 = b_{t_i} T_1 - b_{d_1} X_1}$, $\boldsymbol{B_4 = b_{t_i} T_2 - b_{d_2} X_2}$, $\boldsymbol{B_5 = e(g_1, T_4)^{b_{r_i}}}$ y $\boldsymbol{B_6 = e(T_3, g_2)^{b_{t_i}} * e(h, w)^{-b_{d_1} - b_{d_2}} * e(S, R)^{-b_{u_1} - b_{u_2}} e(g_1, g_2)^{-b_{r_i}}}$.

- Una vez calculados estos valores los enviará al administrador que calculará un número aleatorio $\boldsymbol{c = H(M, T_1, T_2, T_3, T_4, T_5, B_1, B_2, B_3, B_4, B_5, B_6)}$ y se lo enviará a $\boldsymbol{U_i}$. Entonces el usuario calculará:
  $\boldsymbol{s_{u_1} = b_{u_1} + cu_1}$, $\boldsymbol{s_{u_2} = b_{u_2} + cu_2}$, $\boldsymbol{s_{d_1} = b_{d_1} + cd_1}$, $\boldsymbol{s_{d_2} = b_{d_2} + cd_2}$, $\boldsymbol{s_{r_i} = b_{r_i} + cr_i}$ y $\boldsymbol{s_{t_i} = b_{t_i} + ct_i}$. La firma será la tupla formada por los valores $\boldsymbol{sign_i^M = (T_1, T_2, T_3, T_4, T_5, c, s_{u_1}, s_{u_2}, s_{d_1}, s_{d_2}, s_{r_i}, s_{t_i})}$.

Para comprobar la validez de una firma un mensaje $\boldsymbol{sign_i^M}$ el $\boldsymbol{GM}$ comprobará que se satisfacen las siguientes ecuaciones:

- $\boldsymbol{s_{u_1} X_1 = cT_1 + B_1}$.
- $\boldsymbol{s_{u_2} X_2 = cT_2 + B_2}$.
- $\boldsymbol{s_{t_i} T_1 - s_{d_1} X_1 = B_3}$.
- $\boldsymbol{s_{t_i} T_2 - s_{d_2} X_2 = B_4}$.
- $\boldsymbol{e(g_1, T_4)^{s_{r_i}} = T_5^c B_5}$.
- $\boldsymbol{e(T_3, g_2)^{s_{t_i}} * e(h, w)^{-s_{d_1} - s_{d_2}} * e(S, R)^{-s_{u_1} - s_{u_2}} e(g_1, g_2)^{-s_{r_i}} = (e(Q, g_2) / e(T_3, R))^c B_6}$.

Vemos que, siendo $\boldsymbol{T_1 \ldots T_4}$ $\boldsymbol{s_{u_1} \ldots s_{t_i}}$ de 170 bits cada uno, tenemos 11*170 = 1870 y sumando 1024 bits de $\boldsymbol{T_5}$ tenemos que las firmas son de 2894 bits que son 362 bytes aproximadamente.

Para ver qué usuario firmo cierto mensaje, el $\boldsymbol{GM}$ parseará los datos de la firma y calculará $\boldsymbol{A = T_3 - (x_1 T_2 + x_2 T_2)}$ usando los valores de su clave secreta y comprobará qué certificado se corresponde con $\boldsymbol{A}$ en su tabla de transcripciones. A partir de esta comprobación podrá obtener $\boldsymbol{C_i}$. Una vez obtenido este valor, se pueden trazar los mensajes firmados por el usuario $\boldsymbol{U_i}$ comprobando si $\boldsymbol{T_5 = e(C_i, T_4)}$ para $\boldsymbol{T_4}$ y $\boldsymbol{T_5}$ parámetros de cada firma de los mensajes que se comprueban.

C. Operador bilineal

Sean $\boldsymbol{G1}$ y $\boldsymbol{G2}$ dos grupos multiplicativos (por lo tanto cíclicos) de orden primo $\boldsymbol{p}$ con generadores $\boldsymbol{g1}$ y $\boldsymbol{g2}$ respectivamente. Sea un isomorfismo computable de $\boldsymbol{G2}$ a $\boldsymbol{G1}$ con $f(\boldsymbol{g2}) = \boldsymbol{g1}$ y $\boldsymbol{e}$ un operador computable $\boldsymbol{e}$: $\boldsymbol{G1\ x\ G2 \rightarrow G3}$ que cumple las siguientes propiedades:

- Bilinealidad: para todo $\boldsymbol{x}$ perteneciente a $\boldsymbol{G1}$, $\boldsymbol{y}$ perteneciente a $\boldsymbol{G2}$ y $\boldsymbol{a, b}$ números enteros se cumple $\boldsymbol{e(x^a, y^b) = e(x, y)^{ab}}$. Además, fijado $\boldsymbol{x}$, $\boldsymbol{e(x, y)}$ es un operador lineal (cumple las propiedades aditiva y homogénea) de $\boldsymbol{G2}$ a $\boldsymbol{G3}$ y, fijado $\boldsymbol{y}$, $\boldsymbol{e(x, y)}$ es un operador lineal de $\boldsymbol{G1}$ a $\boldsymbol{G3}$
- No degeneración: $\boldsymbol{e(g1, g2) \neq 1}$.

Entonces $\boldsymbol{e}$ es un operador bilineal entre $\boldsymbol{G1}$ y $\boldsymbol{G2}$.

D. Funciones unidireccionales

Una función $\boldsymbol{F}$ se considera unidireccional o de un solo sentido si existe un algoritmo que pueda calcular el resultado de $\boldsymbol{F(x)}$ en tiempo polinómico en función de la longitud de $\boldsymbol{x}$ y que, además, no exista un algoritmo probabilístico que pueda calcular la pre-imagen de $\boldsymbol{F(x)}$ en tiempo polinómico para $\boldsymbol{x}$ elegido aleatoriamente.

E. Autoridad certificadora

Una autoridad certificadora es una entidad confiable encargada de firmar certificados digitales (que contendrá información sobre el emisor del certificado y su clave pública). Antes de firmar un certificado, la autoridad certificadora comprobará la identidad de la empresa o particular que ha solicitado validar su certificado. A la hora de comunicarse con un servidor, éste enviará la firma de su certificado al cliente y dicho cliente intentará verificar dicha firma contra una lista de autoridades en las que se confía (los navegadores actuales suelen venir con una lista de autoridades por defecto). En caso de que no se pueda encontrar la autoridad que firmó el certificado en la lista, se avisará al usuario de que no se reconoce la autoridad emisora del certificado (por ejemplo, puede haber una página que firme sus propios certificados) y se le preguntará si desea continuar con la comunicación.

Dentro de las entidades certificadoras existen diferentes niveles. El certificado de una autoridad certificadora puede estar firmado por una autoridad de nivel superior o por ella misma si no existiese (autoridad raíz). A partir de un certificado raíz existe normalmente una jerarquía de autoridades certificadoras cuyo certificado dependerá del raíz, y que permitirán finalmente firmar certificados de entidad final (para identificar páginas, servicios web y particulares). En caso de querer incluir un certificado dentro del ordenador, un usuario puede importar desde su navegador el certificado raíz de la jerarquía en la que desee confiar.

F. Logaritmo discreto

El logaritmo discreto de $\boldsymbol{y}$ en la base $\boldsymbol{g}$ es la solución $\boldsymbol{x}$ de la ecuación $\boldsymbol{g^x mod\ n = y} \Leftrightarrow \boldsymbol{log_g(y)\ mod\ n = x}$ siendo $\boldsymbol{y}$ y $\boldsymbol{g}$ elementos de un grupo cíclico finito de orden $\boldsymbol{n}$.

Esta ecuación es de especial interés en criptografía gracias al teorema de Euler-Fermat. En primer lugar, consideremos $\boldsymbol{\Phi(n)}$ función que dado entero $\boldsymbol{n}$ devuelve el número de enteros positivos menores o iguales que $\boldsymbol{n}$ y coprimos con él (es decir que su máximo común divisor sea 1). El teorema de Euler-Fermat dice que:
Sean $\boldsymbol{g}$ y $\boldsymbol{n}$ enteros coprimos, entonces $\boldsymbol{g^{\Phi(n)} = 1\ mod\ n}$.

Además, si $\boldsymbol{g^c = 1 \Leftrightarrow c = 0\ mod\ \Phi(n)}$ entonces se dice que $\boldsymbol{g}$ es una raíz primitiva módulo $\boldsymbol{n}$. Con esto, dado un número primo $\boldsymbol{n}$ y una raíz primitiva $\boldsymbol{g}$ podremos generar un grupo de $\boldsymbol{n - 1}$ elementos a partir de la base $\boldsymbol{g}$ en los que cada elemento depende de las condiciones anteriores y, por ello, es difícil de averiguar sin conocerlas. En el caso concreto de nuestro problema, si mantenemos $\boldsymbol{x}$ en secreto es computacionalmente inviable para un oponente calcular la solución de $\boldsymbol{g^x mod\ n = y}$ en tiempo polinómico. Para este problema existe un algoritmo cuántico desarrollado por Peter Shor que lo resuelve en tiempo polinómico, pero hasta el desarrollo de ordenadores con esta capacidad de computación los criptosistemas basados en el logaritmo discreto siguen siendo seguros en este punto.

Podemos observar a continuación, en la **Tabla F.1.,** que las únicas raíces primitivas módulo 7 son el 3 y el 5.

g	$g^1 \bmod 7$	$g^2 \bmod 7$	$g^3 \bmod 7$	$g^4 \bmod 7$	$g^5 \bmod 7$	$g^6 \bmod 7$
1	1	1	1	1	1	1
2	2	4	1	2	4	1
3	3	2	6	4	5	1
4	4	2	1	4	2	1
5	5	4	6	2	3	1
6	6	1	6	1	6	1

Tabla F.1. Subgrupos multiplicativos de Z_7.

G.Computational Diffie-Hellman (CDH)

Computational Diffie-Hellman es una suposición de complejidad computacional dentro de un grupo cíclico $\boldsymbol{G}$. Bajo esta suposición, dado un grupo cíclico (multiplicativo) $\boldsymbol{G}$ de orden $\boldsymbol{n}$, un generador aleatorio $\boldsymbol{g}$, dos elementos aleatorios elegidos de forma independiente $\boldsymbol{a}$, $\boldsymbol{b}$ de $\boldsymbol{Z_n}$ y dada la tupla $(\boldsymbol{g}, \boldsymbol{g^a}, \boldsymbol{g^b})$ es computacionalmente costoso calcular el valor $\boldsymbol{g^{ab}}$.

H.Decisional Diffie-Hellman (DDH)

Decisional Diffie-Hellman es una suposición de complejidad computacional de un problema que implique logaritmos discretos en cierto grupo cíclico $\boldsymbol{G}$. Bajo esta suposición, dado un grupo cíclico (multiplicativo) $\boldsymbol{G}$ de orden $\boldsymbol{n}$ con generador $\boldsymbol{g}$ y tres elementos aleatorios elegidos de forma independiente $\boldsymbol{a}$, $\boldsymbol{b}$ y $\boldsymbol{c}$ de $\boldsymbol{Z_n}$ las distribuciones de probabilidad de $(\boldsymbol{g^a}, \boldsymbol{g^b}, \boldsymbol{g^c})$ y $(\boldsymbol{g^a}, \boldsymbol{g^b}, \boldsymbol{g^{ab}})$ son computacionalmente indistinguibles. Esta suposición guarda una estrecha relación con la suposición del logaritmo discreto dado que, en un grupo donde se pudiesen calcular logaritmos discretos de forma eficiente, se podría determinar fácilmente si $\boldsymbol{c = ab}$. Este problema también está relacionado con **CDH** ya que hay grupos donde resolver el problema de **DDH** es sencillo pero resolver el problema de **CDH** se piensa que es complejo.

I. Linear Diffie-Hellman (LDH)

Linear Diffie-Hellman es una suposición de complejidad computacional tal que si un algoritmo resuelve el problema de LDH en cierto grupo $\boldsymbol{G}$ entonces dicho algoritmo resolverá el problema de DDH en $\boldsymbol{G}$. En general, esto no tiene por qué ser cierto a la inversa por lo que esta suposición es muy útil al trabajar con grupos bilineales donde DDH no se cumpla. El problema que plantea esta suposición es el siguiente:
Dado un grupo cíclico $\boldsymbol{G}$ y tres generadores aleatorios $\boldsymbol{g_1}$, $\boldsymbol{g_2}$ y $\boldsymbol{g_3}$, el problema consiste en averiguar a partir de $(\boldsymbol{g_1}, \boldsymbol{g_2}, \boldsymbol{g_3}, \boldsymbol{g_1^a}, \boldsymbol{g_2^b}, \boldsymbol{g_3^c})$ si $\boldsymbol{c = a + b}$ para $\boldsymbol{a}$, $\boldsymbol{b}$ y $\boldsymbol{c}$ aleatorios.

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

Con el fin de generar firmas grupales más cortas, se presentó en **[3]** en el año 2004 el problema q-SDH. En este problema se consideran dos grupos cíclicos $\boldsymbol{G_1}$ y $\boldsymbol{G_2}$ de orden primo $\boldsymbol{n}$ con generadores $\boldsymbol{g_1}$ y $\boldsymbol{g_2}$ respectivamente. Con estas condiciones se considera el problema de a partir de la entrada $(\boldsymbol{g_1}, \boldsymbol{g_2}, \boldsymbol{g_2^y}, \boldsymbol{g_2^{y^2}}, \dots, \boldsymbol{g_2^{y^q}})$ obtener la salida $(\boldsymbol{g_1^{1/(y+x)}}, \boldsymbol{x})$ con $\boldsymbol{y + x \neq 0 \; mod \; p}$ para $\boldsymbol{x}$ perteneciente a $\boldsymbol{Z_P^*}$.

K.Strong RSA

Es una suposición de seguridad para protocolos o esquemas que utilicen **RSA**. Se basa en que dado $\boldsymbol{N}$, cuya factorización es desconocida para un atacante, y el texto cifrado $\boldsymbol{c}$ es inviable computacionalmente encontrar un par $(\boldsymbol{m}, \boldsymbol{e})$ que cumpla $\boldsymbol{c \equiv m^e \; mod \; N}$. Resolver este problema es al menos tan sencillo como factorizar un número de grandes dimensiones (ya que si conseguimos factorizar $\boldsymbol{N}$ podríamos resolver este problema para cualquiera que fuera el texto cifrado $\boldsymbol{c}$ pues podríamos calcular la clave privada $\boldsymbol{d}$).

L. Advanced Encryption Standard (AES)

AES [14] es un cifrado simétrico por bloques que surgió con el fin de reemplazar el algoritmo de cifrado DES que era el estándar del momento. Fue publicado por el NIST (National Institute of Standards and Technology) en el año 2001. Los motivos para el cambio de estándar fueron (entre otros):

- DES había sido roto por fuerza bruta, el doble DES era atacable con un "**man in the middle**" y el triple DES era demasiado lento.
- DES había sido diseñado para optimizar la implementación a nivel de hardware y se buscaba un algoritmo eficiente a nivel de software.
- El tamaño de bloque de DES era demasiado pequeño (64 bits).

Con el fin de obtener un nuevo estándar el NIST sacó un concurso en 1997 buscando un nuevo algoritmo seguro ante los ataques conocidos del momento, con unos costes aceptables (memoria necesaria, hardware y software, licencias, etc.), que fuese multiplataforma y con un diseño flexible. De los cinco finalistas en 1999 finalmente ganó el Rijndael en el 2001. El algoritmo **AES** es ligeramente diferente al Rijndael ya que en este último el tamaño de clave y bloque pueden especificarse de forma independiente (128,192 o 256 bits) mientras que en el **AES** el bloque es siempre de 128 bits y las claves pueden ser de los tres tamaños especificados anteriormente. El algoritmo consiste en una serie de rondas cuyo número depende de la longitud de la clave. Al tratarse de un algoritmo de clave simétrica el proceso de desencriptar es el inverso al proceso de encriptar manteniendo la clave. Es un algoritmo bastante complejo por lo que su seguridad y funcionamiento no se discutirá en este documento. Los modos de operación más comunes del **AES** son:

- ECB (Electronic Codebook) que consiste simplemente en dividir el texto en bloques de longitud fija y encriptar/desencriptar cada bloque por separado (ver figuras **L.1.** y **L.2.**). Este modo tiene la desventaja de que dos textos planos iguales producen el mismo texto cifrado por lo que si el mensaje tiene patrones de bits (como una imagen) estos no quedan ocultos.

- CBC (Cipher Block Chaining) **[32]** que consiste en realizar una operación **XOR** entre el bloque de texto plano y el bloque anterior cifrado antes de encriptar o entre el bloque de texto desencriptado y el bloque anterior sin desencriptar (ver figuras **L3.** y **L.4.**). Para el primer bloque tenemos un vector de inicialización (IV) de la misma longitud que un bloque. La desventaja de este modo es que se debe encriptar de forma secuencial lo que no permite paralelizar este proceso (aunque desencriptar si es paralelizable).

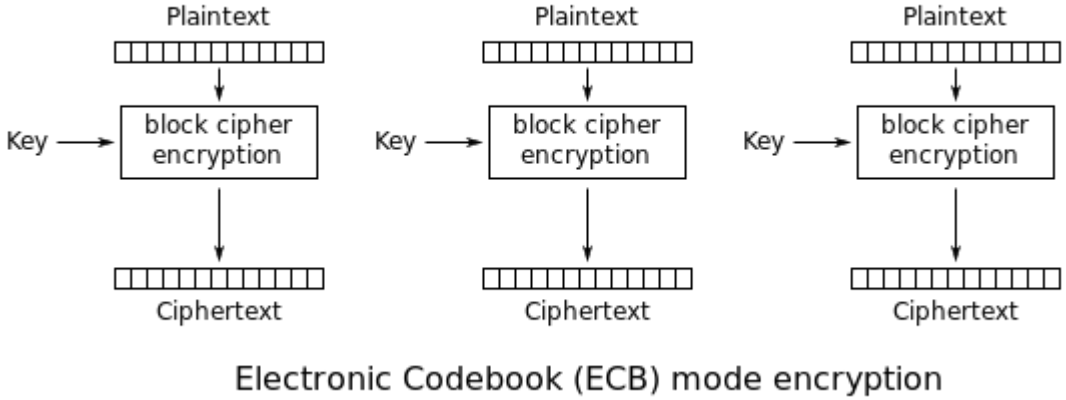

Figura L.1. ECB encriptar.

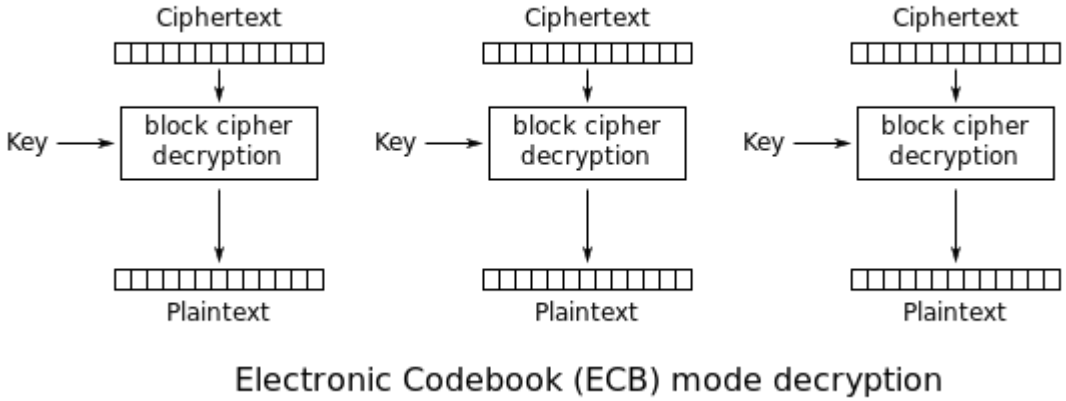

Figura L.2. ECB desencriptar.

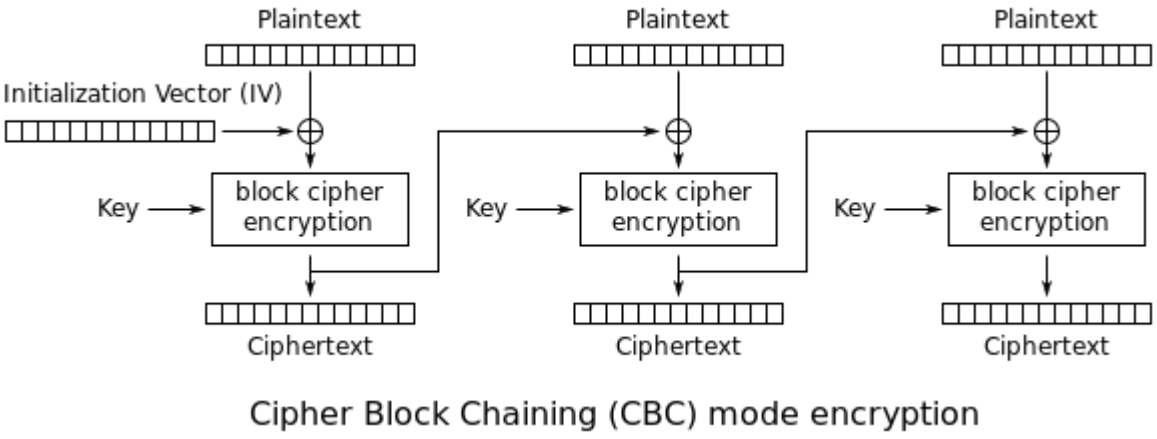

Figura L.3. CBC encriptar.

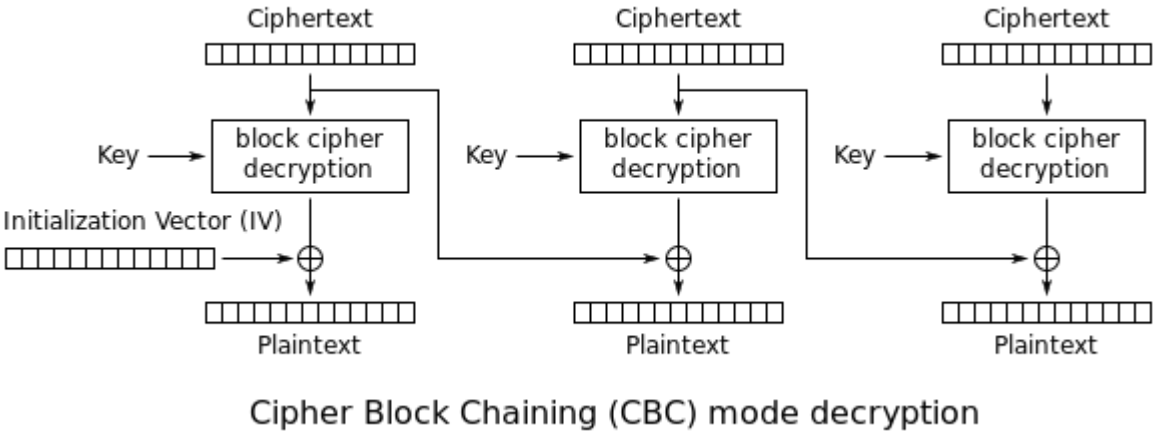

Figura L.4. CBC desencriptar.

M. RSA

El criptosistema **RSA [15]** debe su nombre a sus autores (Ronald **R**ivest, Adi **S**hamir y Leonard **A**dleman) y se trata de un criptosistema de clave pública o asimétrica. Actualmente es uno de los criptosistemas más utilizados tanto para cifrar como para firmar y su robustez se basa en la dificultad del problema de factorizar números enteros grandes. El criptosistema se compone de los siguientes elementos:

- Dos números $\boldsymbol{p}$ y $\boldsymbol{q}$ primos y $\boldsymbol{N = pq}$. Al ser ambos números primos el valor de la función de phi de Euler es $\boldsymbol{\varphi(N) = (p-1)(q-1) = pq - p - q + 1 = N - p - q + 1}$.
- Un número $\boldsymbol{e}$ menor que $\boldsymbol{\varphi(N)}$ y coprimo con él (es decir que el máximo común divisor de $\boldsymbol{\varphi(N)}$ y $\boldsymbol{e}$ sea 1). De esta manera tenemos garantizado que $\boldsymbol{e}$ tenga inverso multiplicativo módulo $\boldsymbol{\varphi(N)}$.
- Un número $\boldsymbol{d}$ tal que $\boldsymbol{de \equiv 1 \; mod \; \varphi(N)}$.

A partir de estos elementos tendremos que la clave pública consistirá en los valores $(\boldsymbol{N}, \boldsymbol{e})$ y la clave privada en los valores $(\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{d})$. Para cifrar un mensaje $\boldsymbol{M}$, se divide en bloques cuya longitud dependerá del tamaño del módulo **RSA** y se convierte ese mensaje en un número entero $\boldsymbol{m}$ que será menor que $\boldsymbol{N}$. A continuación, se cifra este número $\boldsymbol{m}$ con la fórmula $\boldsymbol{c = m^e \bmod N}$. Para descifrar el mensaje utilizamos la clave privada para deshacer el cifrado $\boldsymbol{m = c^d \bmod N}$. Es recomendable por razones de seguridad que los números primos $\boldsymbol{p}$ y $\boldsymbol{q}$ tengan una longitud parecida.

N.APDU

Una APDU (Application Protocol Data Unit) es una unidad de comunicación entre el lector de tarjetas y la smartcard. En función de si se trata de una APDU para transmitir comandos a la tarjeta o una APDU en respuesta a un comando se puede distinguir entre Command APDU (C-APDU) y Response APDU (R-APDU). Si una APDU cumple en estándar ISO/IEC 7816-4 **[19]** será independiente del protocolo de transmisión utilizado para transmitir datos entre el lector y la tarjeta (en particular se aplica a los protocolos estándar **T=0** y **T=1 [18]**). Las C-APDUs se componen de una cabecera y el cuerpo del mensaje (que puede estar ausente en algunos casos) cuya estructura podemos ver en la **Figura N.1.**. La cabecera se compone del byte de clase (**CLA**), el byte de instrucción (**INS**) y los dos bytes para incluir parámetros (**P1** y **P2**). La clase servirá para identificar aplicaciones dentro de la smartcard y su conjunto de comandos permitido. El byte de instrucción indicará el comando a ejecutar y los bytes **P1** y **P2** se utilizarán en caso necesario para controlar variaciones en la ejecución de dicho comando. En el cuerpo del mensaje el byte **Lc** se utiliza para indicar la longitud de los datos del comando en bytes que vienen a continuación en el campo **Data**. El byte **Le** se utiliza para indicar la longitud esperada en bytes de la respuesta (en caso de ser '**00**' se esperaría que la tarjeta devolviese la máxima cantidad de información posible para el comando utilizado). En caso de necesitar más de un byte para indicar la longitud de los datos enviados o que se esperan se pueden utilizar APDUs extendidas donde los campos **Lc** y **Le** son de 3 bytes siendo el primero de ellos siempre '**00**' por lo que se podrían enviar datos de hasta 65536 bytes.

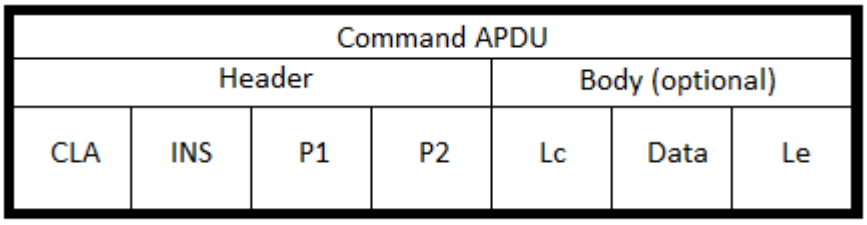

Figura N.1. Estructura C-APDU.

En el caso de las R-APDUs tenemos el campo de los datos que es opcional y los campos finales que devuelven el estado del comando procesado anteriormente. El campo de los datos tendrá la longitud especificada en el campo **Le** de la C-APDU anterior (aunque podría ser 0 en caso de que al procesar dicha C-APDU haya ocurrido un error lo cual se indicaría en los dos bytes **SW1**, **SW2**). Los bytes finales indican el código de retorno que devuelve la tarjeta después de procesar el comando. Si la operación se ha llevado a cabo satisfactoriamente la tarjeta devolverá '**90 00**'.

Figura N.2. Estructura R-APDU.

O.ZKP basada en el problema del coloreado de un grafo

En la referencia **[9]** se propone un protocolo utilizando el problema de coloreado de grafos que consiste en colorear los vértices de un grafo con tres colores sin repetir color entre vértices adyacentes. Este problema es sencillo de resolver para un grafo como el representado en la **Figura O.1.** pero, para grafos más extensos y complicados, está demostrado que el problema del coloreado de grafos de tres colores (incluyendo decidir si puede colorearse con tres colores) es NP-Completo **[10]**. En el protocolo propuesto un probador **P** que tenga una solución escogerá tres colores aleatorios y pintará su solución sobre el grafo cubriendo todos los vértices. Posteriormente **V** elegirá una arista y **P** revelará los dos vértices relacionados con dicha arista demostrando que son de diferente color (si conoce una solución para dicho grafo) o no. Siendo **N** el número de aristas del grafo vemos que, tras comprobar que la primera arista elegida cumple que los colores son distintos, la probabilidad de que hayan engañado a **V** rellenado el grafo sin conocer la solución es de **(N-1)/N**. Repitiendo el proceso desde el principio múltiples veces (**con nuevos colores aleatorios**) **V** puede estar seguro con una alta probabilidad del conocimiento de una solución por parte de **P**. Para asegurar que un protocolo de este tipo es **válido**, el probador puede enviar los colores de los vértices cifrados previamente añadiendo alguna firma que le asocie con dichas soluciones y cuando el verificador le solicite una arista devolverle el valor para descifrar los vértices correspondientes a dicha arista. Es importante destacar el hecho de que este protocolo no aporta conocimiento a **V** sobre la solución. Para ello consideramos que **V** posee un programa que puede extraer información del protocolo al interactuar con **P**. En este caso (suponiendo que **P** tuviera tiempo suficiente para ello), si **P** pintase el grafo de forma aleatoria y reiniciase el protocolo cuando la arista pedida tenga vértices con los mismos colores, siempre se podría dar una solución aparentemente correcta a **V** (este protocolo no cumpliría la propiedad de **validez**). Es importante tener en cuenta que para cada arista pedida el protocolo utiliza **nuevos colores** para pintar el grafo impidiendo que se pueda ir guardando la solución. En este caso, si el programa tuviese éxito al extraer información de una ejecución correcta también lo haría en la misma cantidad con el protocolo que rellena el grafo de forma aleatoria. Dado que claramente en este segundo supuesto no se aporta información sobre una solución del problema (pues no se dispone de solución), no se puede extraer ninguna información.

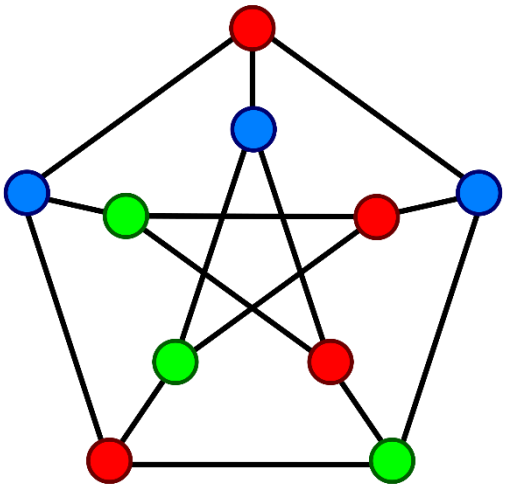

Figura O.1. Una solución del problema de colorear un grafo con 3 colores.

P. Manual del usuario

En este apartado se describirá a partir de ejemplos la forma de uso de los programas implementados para los protocolos propuestos. Estos programas comprenden varios scripts, el servidor que emula al administrador de grupo, los programas para las smartcard y aquellos para intercambiar datos con el servidor y con la smartcard. Salvo que se indique lo contrario, los ficheros necesarios para la ejecución de cada programa deben estar en el mismo nivel de directorio que el programa que se ejecuta y se debe respetar el orden de los argumentos que reciben los programas y el nombre de los ficheros. También será necesario tener instalados los módulos de Python 2.7 necesarios y la librería de firmas **libgroupsig [17]**. El software desarrollado tendrá una arquitectura cliente-servidor donde el administrador del grupo y el ordenador con lector de tarjetas pertenecerán al entorno del servidor y los scripts de Python para realizar las peticiones junto con la applet de la smartcard pertenecerán al entorno del cliente. Toda la parte del servidor y los scripts de Python del cliente se ha ejecutado en máquinas virtuales con **Ubuntu 14.04** y 4GB de RAM y la parte de simulación de la smartcard en Windows 10 con 16 GB de RAM (aunque en este caso la RAM no es demasiado relevante ya que en la simulación se emulan las características de la tarjeta simulada –ver apartado **4.Pruebas y resultados**-) compartiendo los archivos necesarios con las máquinas virtuales para agilizar el proceso de intercambio de datos.

P.1. Servidor de los protocolos

Para lanzar el servidor Flask de cualquiera de los protocolos basta con navegar hasta la carpeta **group_manager_flask_N** del servidor, donde **N** indica el protocolo que se desea utilizar, y ejecutar el comando:

```
$ ./run.sh
```

Este comando se encarga de decir a Python Flask que la aplicación que debe ejecutar es **group_manager_flask.py**. Como resultado, tendremos un programa ejecutándose que habrá creado un grupo de firmas con un número base de usuarios y que estará escuchando peticiones

que se le hagan a la dirección que se indica por terminal como se puede apreciar en la **Figura P.1.**.

```
root@coolb:~/PycharmProjects/group_manager_flask_0# ./run.sh
 * Serving Flask app "group_manager_flask"
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Figura P.1. Lanzar el servidor del protocolo.

Para esta implementación se ha considerado identificar a los usuarios reales por el DNI por lo que deben estar dados de alta previamente en la base de datos del servidor. Para ellos basta con incluir los DNI's deseados en el fichero **usuarios.txt** (antes de lanzar el servidor) que contendrá solo una columna con un DNI por fila. Mencionar también que las claves generadas estarán en formato **base 64** para que puedan ser legibles por el usuario.

P.2. Peticiones al servidor

Si nos conectamos a la dirección que se indica en el comando del apartado anterior podremos comprobar que obtenemos un mensaje que dice "**Protocolo N**" siendo **N** el número del protocolo. Ésta es la ruta base de nuestro servidor. Ahora, para realizar peticiones al servidor se utilizan otras dos rutas distintas. La primera ruta sería **rutabase/cmd1/** que se utiliza para solicitar al servidor su clave pública **RSA** a falta de un certificado válido. El resultado se puede observar en la **Figura P.2.** donde obtenemos la clave pública que debemos utilizar para algunas peticiones futuras (podemos copiarla con Crtl + A, Crtl + C).

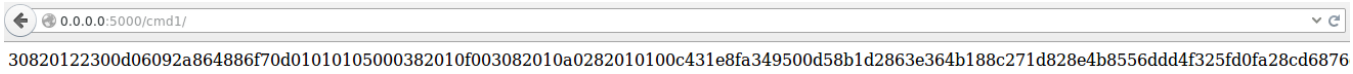

Figura P.2. Respuesta clave RSA servidor.

La segunda ruta sería **rutabase/cmd2/peticioncifrada** y se utiliza para realizar cualquier otra petición al servidor. Dentro de estas peticiones podemos encontrar, por ejemplo, la solicitud de entrada al grupo o la petición de acceso al recurso que controla el servidor. Todas las peticiones irán cifradas con la clave **RSA** pública del servidor que podemos obtener con la primera ruta. El resultado de la petición variará dependiendo de ésta.

P.2.1. Peticiones Protocolo 0, programa cliente y smartcard

A la hora de poder enviar diferentes peticiones cifradas al servidor del **Protocolo 0** se proporciona un programa de Python (**client.py**) que nos devolverá la cadena con la petición que le solicitemos. Primero llamaremos al programa **RSA_keys.py** que generará las claves pública y privada **RSA** de nuestro cliente y las guardará en los archivos **public_key.info** y **private_key.info**.

Para obtener la clave pública del grupo realizaremos lo siguiente:

- Ejecutar el comando:

```
$ python client.py -GRPKEY CLAVEPUBLICARSASERVIDOR
```

Obtendremos entonces una salida con la cadena que representa la petición cifrada para el servidor (también se guarda la petición en el fichero **ASKGROUPKEY.txt** que se borrando el contenido anterior).

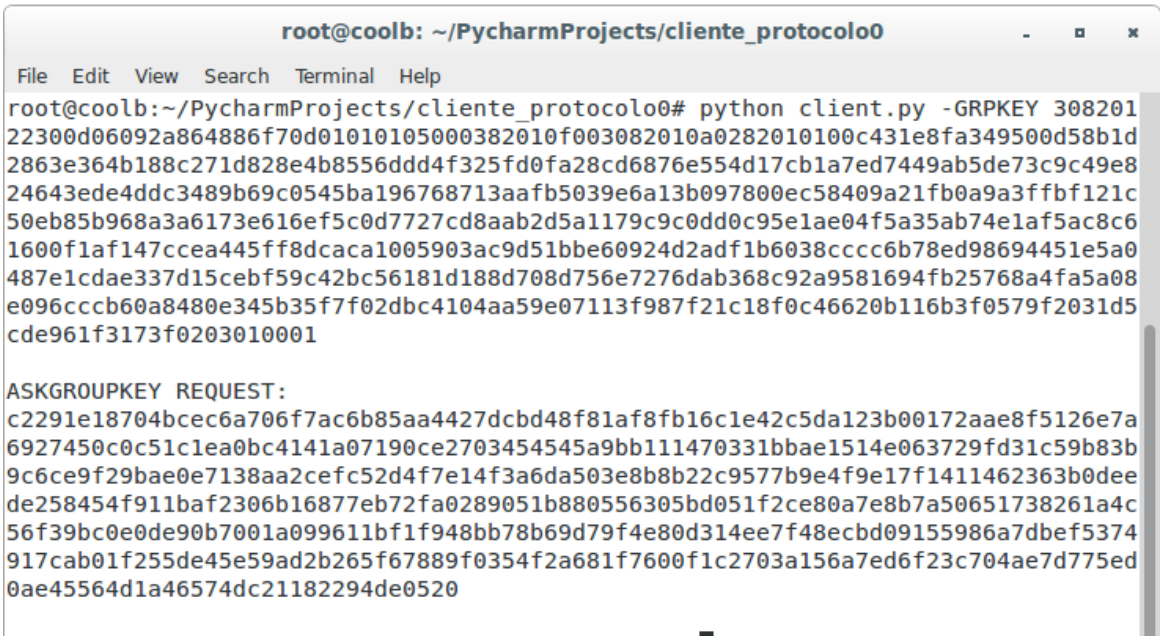

Figura P.3. Petición de clave pública de grupo.

- Accederemos a la ruta **rutabase/cmd2/peticioncifrada** cambiando **peticioncifrada** por el resultado del comando anterior. Obtendremos una respuesta con la clave pública del grupo que debemos copiar en un archivo llamado **grp.key**.

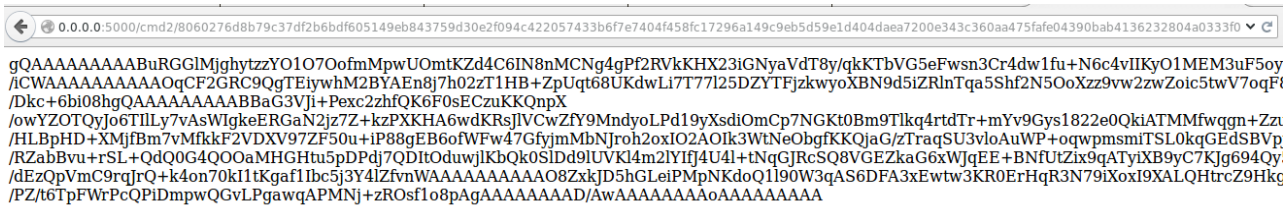

Figura P.4. Respuesta clave pública de grupo.

Para unirnos al grupo realizaremos lo siguiente:

- Ejecutar el comando:

```
$ python client.py -JOIN 1 CLAVEPUBLICARSASERVIDOR
```

Donde **1** indica la fase del proceso de **join**. Obtendremos la petición cifrada junto con una marca temporal llamada **T1** (la petición también se guarda en el fichero **JOIN.txt** borrando el contenido anterior).

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -JOIN 1 308201
22300d06092a864886f70d0101010500038201 0f003082010a0282010100b19428ffe30c1938c9ef
eb79f00c56d7f3c7673c683e09f2770fb872fcc226d299b6caeecb9da722a5cb0bfea35c2d2b0a0f
6e7cff2aab1d8433ba1990077a1ea24e9bb5094788e34daf07b8d16551e99d35005d9baf73e65a28
20fc0e426c0a8a1518ea74dab823df4e70ad6e9f99f0334c3c0e74e7c5179082f691c46e058212e2
81cb31fb56937d84aca75725aee46f0e819f174d13dbf6cfbaa113cfbfb7992f8027f475d63e4a58
ce0356ccdc379bbfcb47798399f95a8683a8845e2e9b879405b1cddff170b734621411762311408c
48063168fafdc13df9fd632f756749e9d5bebd7ddde759a8f7965f8f402e2425efc5227050a2cdd5
81d8ae9a84750203010001

JOIN1 REQUEST:
7c280eb0574199ed4405d9d9a1ecf443fb82c40c2dd1c46b1c80807045c8a3ac8a41b911fe08d1c5
50ca1a6f16367a1758739c8a5fc9072034c337d8ce2246c0edc09ebf843eeb2694cb38575a31eab3
dccc311023fe5d02b1d20bb9b6d8b7d9d79af4bb2a58294a60412efff7c7baa721713f97cb223212
0664e52a407aaed47999953d883aa7b317b9a8fa445b5675048abad5df4f63bcfa88b51cc7fdd3ff
56af2986b8a70a29674efd6348f66c81e2cf266dcdb98f404005326c9b33f32f6b05a2dcadd4d35d
6ae5f088205c1e7d9c4e6e67739ac96f8f25e79e43238fdb2a70c6fb3f5f60a75ab0553a25ae55e6
c7a6ef1683113be657528d40d523c4e785ad142c6a464d56797d1acca22b7ec7a4a442c6be64d1e2
9f4baa8f99db3f568ee17cef3d18e46364456facf9389858dd895e73be01f2df850d04964dff0892
3b90e03e788f92c200d13a8fd573c4e9cbb5fb0d70028e91f5d699c9ffee4cdcb9b5ad8fd0aa32e6
3dc9c30cca3159316965a703fc1e53e2690b9e6f69dda8787d22efac84ab2bf4bafe91901d5d98c7
9ab70ad9553f96e4ff8307940137e00d2b584930b3850d483a1d42ed35ccc56d9928f559f5abcb67
845744c981a5bae019202dfdf926a3fa4d17f003f71262111da7afdd14c28564492b11afbfb008e2
dde850cdc32d01394411f6968ed29bd9060f6cbb816eaaf04f0bdd689ccbf98792275737623f6c47
eb2d5c37689fd91cd992d9646e8b51d419d6889cfadfb6d436de1bc68fd1ae6f979569c4892c5cad
36cb46855e5314e538ca12a0edf5c98ae7f88cd59d776d08598ef81e54c88f16f81683d67c110b89
c0fc91e2802aef8e1dde3dfc4fa3c652e33e472c7c4eb5e8a499a9bee64f9282d955976adde152c2
0041b2c30d01f0249fe364eefc05746da389abacaabe5de1b2398cc785b36096198d8a7b00afd691
99f5bfa062989167196c96bb703bb641557b3e14ce6badec0a180f2fdf4c35bc06c160d0f11edfa7
9412f69fc654d3b63c076eea6033799e4945e81ae484e8d3cd840b00dd527e9ea5808b1eb43c44c2
ce23233c0c5ecc2d

T1:
1496181361.96
```

Figura P.5. JOIN 1.

- Accederemos a la ruta **rutabase/cmd2/peticioncifrada** cambiando **peticioncifrada** por el resultado del comando anterior. Obtendremos una respuesta encriptada con nuestra clave pública **RSA** (que hemos enviado en la petición) para evitar que nadie más pueda leerla. Copiamos la respuesta y la desencriptamos con el siguiente comando:

```
$ python client.py -dec RESPUESTADELSERVIDOR
```

En la respuesta, si todo ha ido bien, recibiremos el **T1** del primer mensaje (que indica que el servidor y solo él lo ha podido leer satisfactoriamente) junto con una marca temporal **T2**. Esta respuesta también se guardará en el fichero **decrypted.txt** borrando el contenido previo.

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -dec 742fd6443cde50c0cc544
74bd099652d41d4c59a1736dfee06246258844b38fe03a34c325baa79f5baa81d2d63b56904fb2d4e11ef867aac0
e95a70b61ae53cd74a2b72adb5c76a98b9c5941779d483956b5735f6bb98e2865da6abf764f62eb270423a1c1012
0cba377b569687c928fc93725902a687d9b2959b6e2a86825efc64ea21190b20c7f72c67669fd7521ef00777425e
9c7e1f2809a20fd0bae66ec35552e3782c94f48b6e0e12b689cf865c0a7b6df8fc40e5529f4a9004d6372a17f019
72fca7ded9ce64b1e3275315f0861f30316f8c5598be6f0c4ec3326c5a2e3b4dee5411d1ca8f1aeffd6119d7a14b
a88f0f21f9c274362e70f718dacc075

Dec_msg:
JOIN2: T1=1496181361.96    T2= 1496181549.54
root@coolb:~/PycharmProjects/cliente_protocolo0#
```

Figura P.6. JOIN 2.

- Crearemos última petición de la fase de **join** de la siguiente manera (que obtendremos por terminal y se guardará también en el fichero **JOIN.txt**):

```
$ python client.py –JOIN 3 CLAVEPUBLICARSASERVIDOR T2 IDENTIFICACION
```

Donde **T2** debe ser el valor obtenido anteriormente e **IDENTIFICACION** una identificación válida como cliente real del servicio que controla el grupo de firmas (para esta implementación se ha considerado el DNI como se ha mencionado anteriormente).

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -JOIN 3 30820122300d06092a
864886f70d01010105000382010f003082010a0282010100b19428ffe30c1938c9efeb79f00c56d7f3c7673c683e
09f2770fb872fcc226d299b6caeecb9da722a5cb0bfea35c2d2b0a0f6e7cff2aab1d8433ba1990077a1ea24e9bb5
094788e34daf07b8d16551e99d35005d9baf73e65a2820fc0e426c0a8a1518ea74dab823df4e70ad6e9f99f0334c
3c0e74e7c5179082f691c46e058212e281cb31fb56937d84aca75725aee46f0e819f174d13dbf6cfbaa113cfbfb7
992f8027f475d63e4a58ce0356ccdc379bbfcb47798399f95a8683a8845e2e9b879405b1cddff170b73462141176
2311408c48063168fafdc13df9fd632f756749e9d5bebd7ddde759a8f7965f8f402e2425efc5227050a2cdd581d8
ae9a84750203010001 1496181549.54 05292331

JOIN3 REQUEST:
047f8ea3991d997bd4c8578a3113cf2b6d5926ceed94b2651cbf35300f3c0f26980ca1a1148cb1f025b2d05a5bbf
a413440ed18eeec62e9a4ddd9421543700eb730e2841ef96349e87f8af659dce49349f15ae66b4865537112c4154
5e04c3af2d8ba19f47602965517ba5aef402c79f4d7078cac49f9171925c40e3a64c9eda8b56c094bfd75e77ebda
14a7291d175e12eff34d87262843d57471c0a50b5da72a51e9fe714bf10cd855060d2ef4272f3ad0008c6821a458
6512be05800e50823eae99eb92066ed083af46b05bcf48b1cf421b0bc923cffccb9cf0ac074c4afefac9f0bbf5ca
5a44d9f872a51b1a5088c6154ff28a2c522bbc2e333dfa4f43c5675f7e263b40a7cd981c4310c993d72ff7599519
def46d537c80ee27d611fbf636c6e71dfe075031455564a5cb1b94dc0f70a538269608a912e0d441df2c6ca346cd
842e60ad3c895b97d026daf1fbfceae300b91f829eb6d69a18e899cb7452d9f3905a9cf1cb93c13b45496c7cee7c
6a53e0bb0388d50058ab9c3fc07c0ff1b60c6d665d39e84706b475aab5abb26104d6bc8a2ba88da5cae980f3f885
3fc800c2d9b219d85be7a4c92252048672753876313171 88c9987f75cd60dc5a2a5662305cd2772602947f2f6556
971b91b431c16a281b7c048ee9c398d3d566b3d917fd38f5c8ee5bb75dae1b0671ef78f2cc04d6b2162855e94d68
54830a92c68278e4b1a341efd3d0ca3c0fac00afc0eee7306628467027c3d6ec45d93a93555e06b1421346c584ca
eae71ec4be774d3b0cd769553efb8469df4fe3bf5257c563732ee0297842feccad1b418e988e0774a6a683abe0f3
14a908178b246c36450e00db1e76b8316f79ae3a4b53bc4d600ee94fdacd48feeffcc2476aa9a756e751ecce65e0
0ddfdaa8f579027fd97d43c2ce55d2a7d86871c5c558dcaa53e8715a84df5858c47a2f1d95a1241122c025340935
3ca3b6c8bb2a7b3a7f00ff8626c51863b025d530fbd7245d5bc43273ae699e4e3c75c73a98eae87d84b6836a7500
7ebb115a2ca581df767ac3565d6f20e4794a19bb3ff2dd84db76f8dbe94b5670
```

Figura P.7. JOIN 3.

- De nuevo accederemos a **rutabase/cmd2/peticioncifrada** cambiando **peticioncifrada** por el resultado del comando anterior y recibiremos como resultado (si todo ha ido bien) nuestra clave de miembro de grupo encriptada con nuestra clave pública **RSA**. Por ello debemos desencriptarla como anteriormente guardando el resultado en un fichero que llamaremos **member.key**.

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -dec 611d019b03fdfc5b14e67
a38ddff7dbe4bad9c4282c574150debcb433481b490c41ad6076a346e0c3e2584ea994b5de1ed0024bb3ba1dc62a
3cac36fa716b48ac0cb44053a411be5b4a514ff52dcb0b6bafe9ae374d41afada25dd067163aede3f1e3208856ab
8270d9fa9d79d00d8ce529a237c2ce6a8ea11760caceb042366dbb18f70e45373423ba64748ef63673d5cdd02000
3bdc14be119ccb40ab4a912fda5de2787b9155c714f844d0250cbf06a9cf1ae77753e4885fb6a93f4de119580cd0
9ff933ecdf6d69c727201e2f1336e2886cb61c2c05419977d60b1428b66ee04d6c188831e5879c74df802fe5bd9f
ee82a83d14f76b53e12620b9b045a4b91dad7f038e549089d44cfde207e367d546d46fb1a2dbecc2ea222aa127a6
276c7c3a179e0ffe10ba01cffa8340ca9cea34225c8a12396eb807b40626ed63b40fd5fffb612fe27a0cdb30508e
2dcf52e89fad5d97ec16117df1805e57e67c31c95acdda7dfef3b40f4a68f9341f60ceb3b8b44e372ce8f44bb4ba
9abbdef18e6a0e605a66fa662f1649e2eafea7e6b41f5b6f34ed8c35dcbe0844fc7f288c4a71d0db747790155cae
e37dfa17d8f85da4a46e4ba3092ec596b36d6600fba57895ba5b83141886877d19da42c3ac9b31b2fdfa571fa4c3
c4fd51d2de87fe2faf84c99ab95e2156bd829531a9a80f6422fac3c0dd4db2fc31b864488bd7820452e346aedb94
3c3c6aac10e6b638cce927d09b04a4fd749d3f4fb805dd0c09e20da8c803ba291a96d44ae21ab43464f1140d37d3
8cd6cc5692ee02663657cdbc004c9289051ef31953e55f7e218fb51bb657dd890732b7a9d70ae682b0f879db0c11
87a36e312f9c7fe2f4976edce35b249fa562e96e71cb5206e1c83801e785c75917b86155600013ce03b4dd1a8521
783a0f7ce1989b8396efc12dd845a6c29faa2ef5a3d5bdb87fb551dcefdb0ce4ab385466e6b6eb0148a512e55d34
eb63a566b8750c82352ff0307222e0a190b04003bf3060153a5429a4c5f546313e22ffb235bf697558a726c309f0
199e289c950d493498b65a68b9412f4026c0f8adbfd

Dec_msg:
gQAAAAAAAAABhvetJ8t/tS+FGteJRU+XYbgTy6gUaf5U7cQUtyCxKn0sWCWRaDWD2tuxLfILiHAxzn21fxFm4JUYZUzc
m+E4kqJxCxmpaXsr6KfZhmng6dX3eF8lmMXViCxUIWReXszhWFRvObQKCfnXw7HOijvTH9fcQwO5F9o2UKssdFafTJmA
AAAAAAAAAEz8xDFp1yy7ydIeHNKd3j5qioppg/xYS0wpTJFdCBk9eCrWnFAO97V5I5/g+ugQBDu9AQDg4a5PxdAjdi/O
iL+VQoJrszwEKoW7TPKA8rxtgv7h8hDOCxN5kYLNjFTIlLgc30dULrFehAHtcoS33ZzOsgzBBvNpQgkEanz4lAe/IAAA
AAAAAABAAAAAAAAAAAAAAAAAAAAAAAAAlNEQYKZl29yf6O6B2A2yAAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAL4kUI2ZN+
uiloA9wXgcZgAAAAAAAAAIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAABAAAAAAAAAAAAAAAAAAAAAAgdMiw38NV+YWq12g8nIw==
```

Figura P.8. Desencriptando la clave de miembro.

Para solicitar una clave **AES** de forma anónima realizaremos las siguientes acciones:

- Ejecutar el comando:

```
$ python client.py –SIMK 1 CLAVEPUBLICARSASERVIDOR
```

De nuevo **1** indica la fase y recibiremos la petición cifrada junto con una marca temporal **T1**. La salida del comando se mostrará por terminal y se guardará en el fichero **ASKSIMK.txt** eliminando el contenido anterior.

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -SIMK 1 30820122300d06092a
864886f70d01010105000382010f003082010a0282010100b19428ffe30c1938c9efeb79f00c56d7f3c7673c683e
09f2770fb872fcc226d299b6caeecb9da722a5cb0bfea35c2d2b0a0f6e7cff2aab1d8433ba1990077a1ea24e9bb5
094788e34daf07b8d16551e99d35005d9baf73e65a2820fc0e426c0a8a1518ea74dab823df4e70ad6e9f99f0334c
3c0e74e7c5179082f691c46e058212e281cb31fb56937d84aca75725aee46f0e819f174d13dbf6cfbaa113cfbfb7
992f8027f475d63e4a58ce0356ccdc379bbfcb47798399f95a8683a8845e2e9b879405b1cddff170b73462141176
2311408c48063168fafdc13df9fd632f756749e9d5bebd7ddde759a8f7965f8f402e2425efc5227050a2cdd581d8
ae9a84750203010001

ASKSIMK1 REQUEST:
759a4506fba37837d1d17cda2130a2e43423d578047291081cd881bbbdd6bdd94aa4eb1522e5d09f13fe42b4cd75
67e9014e9393a124ab04dabb7405e9c8af77f716a8fccf18e92b41288f4b9c7da56e23cecaba0c1deb8d32a92398
bde9194080cde65ff3820a1b3e2f10470479ac42b4af1671ed01ba742c6acb78f590b86a07cf28350a0afbf66217
663deeb5b0c068b4697f1148d68566fffabaa35053fafaf6eb5bd285aa5a30eef2bd12e936e5b602b6491f901a62
abcdf80148f09ccb91f1c166f4c6f08b6c8b18c0b1ef8ee8c172fde0ad8b4d911312084b408cba0fbf3310b9efd0
99283b4fa800ac40443ad061b7a3820834345f9546e67dff2a5129695506ffc4233e25e482f3c4e5f0ea10d7c784
99b9b28ced1bd7c2d6d3df16e68df0ca5b546ce8588809bad1aca705c7ae5709e970017761afec18f565aa48f996
bdcedf71872d5abefbf87eea00ff35fe0606336ba0aa7c8c46d6e5866cea7ba3e3094208096e57be6e8690d5953a
dd878b77c3dbe64ee1698291a03d651c392dbf946dee4a52836ef8f829a1387b9d8ceb858c88d644acc49b1dcd99
94cb0a1725d7eb80014f5bd990269c43d01b2fde9e381a9c4c80d584853d34d945b6a76d1fc1b7e7d1f1abb1c70c
25028eca673756fe95870df1e2346ac7fd12248ed9993df0d03f5e2e78cd5bfb77fb2817a889b27b2b8e27033300
1233b65acf2f30ba42484b02e93f7ca33627165a41ad41d50fbfca8e8bc05c30ad4d640dc2efe0fee3d2a8d79e9f
2d6c4c27c243e77847258f20453ecfb12055962a2ae648395aa7dd266e2cf6db521390cf53e4fa40452bcde2e492
d77a3e3ee7adbbee56dc487d3f9e690d46981c85f04f7785f8cc27e400e1c8d37ed25b40612ec75217c65b74f3a6
3c4b482fb9b5935df3102cc455bd7c18a54b53a5fe1fc8fc0d17aa4e569f62cea4593d5026f17486340b965879d4
82471c16ab91d8ea0b35753fc178af4c8685d79036b7b01acf00ce140ee647ac4742bee2b56ee5a45bcb735182ee
1b8ad0166d8cbe204e86c9ee9059b3c44b3ff023f38f4d76be3bc99af4f96dcb

T1:
1496183837.62
```

Figura P.9. ASKSIMK 1.

- Accederemos a la ruta **rutabase/cmd2/peticioncifrada** cambiando **peticioncifrada** por el resultado del comando anterior. Obtendremos una respuesta encriptada con nuestra clave pública **RSA**. Copiamos la respuesta y la desencriptamos como anteriormente. Deberíamos obtener el **T1** enviado anteriormente y un **T2** que debemos enviar al servidor.

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -dec 8db662250ed733288a658
ef0058106a647cf40c7c957aa19d33b08f5f917ade6236c109d3395b6da114766636ab928ba5a1b175207f42762a
e61d780dcc6f61f4a953d71ba565e588619da535533d322710fe22988583b313dbac098aef9aa40e463a8578e496
68d0b1951c3d85e5956133bd8c8336a676174ef84151eee59cba7e79f900ac22a195fec70ded443db312536707c8
f5502a63a2b5fecb63c19cb6345c92463524c691a3b624899a202a7966d99e44b3c26bfc5321292adea1be99d487
1e382596651a274a80a8d517166b098cd51971a6ad41c79573135a8629bc9d536b523814f6829140dd58ced2ef84
2a3e8a2c006f802c29036b10dfd9801

Dec_msg:
ASKSIMKEY2: T1=1496183837.62    T2= 1496184191.77
```

Figura P.10. ASKSIMK 2.

En la respuesta, si todo ha ido bien, recibiremos el **T1** del primer mensaje (que indica que el servidor y solo él lo ha podido leer satisfactoriamente) junto con una marca temporal **T2**.

- Crearemos la última petición para obtener la clave pública donde firmaremos el mensaje con la clave de miembro obtenida en la fase de **join**. La petición será de bastante longitud debido a que se incluye dentro de ella el mensaje y su firma (de nuevo podemos obtener la petición por terminal o del fichero **ASKSIMK.txt**):

```
$ python client.py -SIMK 3 CLAVEPUBLICARSASERVIDOR T2
```

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -SIMK 3  30820122300d06092
a864886f70d01010105000382010f003082010a0282010100b19428ffe30c1938c9efeb79f00c56d7f3c7673c683
e09f2770fb872fcc226d299b6caeecb9da722a5cb0bfea35c2d2b0a0f6e7cff2aab1d8433ba1990077a1ea24e9bb
5094788e34daf07b8d16551e99d35005d9baf73e65a2820fc0e426c0a8a1518ea74dab823df4e70ad6e9f99f0334
c3c0e74e7c5179082f691c46e058212e281cb31fb56937d84aca75725aee46f0e819f174d13dbf6cfbaa113cfbfb
7992f8027f475d63e4a58ce0356ccdc379bbfcb47798399f95a8683a8845e2e9b879405b1cddff170b7346214117
62311408c48063168fafdc13df9fd632f756749e9d5bebd7ddde759a8f7965f8f402e2425efc5227050a2cdd581d
8ae9a84750203010001 1496184191.77

ASKSIMK3 REQUEST:
2c5a9858c644a8c1642267618acdc58afa412d9109e76e9078db50fd401153b05a6066cdb79eaed5c84d4fe70bc0
9d9fc23aed77ff52e0f797af0cd91a0e9cb7a56318cc2da021832b17a403ffc2153001cf1162db03cfaccc822d5e
bc003f9fb202011437ff270b970c979d7159c80ee25ad59049beb7eaa204e7924a69137aef5a783bb0d24d25ffd3
e1941cd1411385c9b553ce127cbec69434c6ff4e98598452761b24efe97305fa97da439f9d950a64712df49b390b
88ef4b91805fba43f7511c3a8982a514763472fa255a03af30d6874a7fdf7264e4eb13aa81165f477cb97bd3b0be
10cffcd73bfb52eb202cbcde70ae5da6c389681abdc76a4520428112f9526331d783d7616898d7044aa2e5776055
9cd4f14512eab62f28ba353cd3e61bfde7ca815b5375f31331aad8792d8117ac72543cfcd56cedcbf8e48928cbdf
0071d8ac4b08c4b6469d2d61d986090baa4f103623d13cb7cb43848d2326e56f26772d5e64b05465f9715b576cbf
564bc6d1bdc0fae535c058dbf82d67f3491704f920207d7526065e07d68dce6a4e38e6a2a27618160a4216ca24d6
49a4e314cc63b4878f2957a13f8f8c3305c8ca34946809e853c0300972fc8224ea07d90f072c5660608c0d7c6c35
8d9ba08420dab2625a3440ad7fb0c6a38b48d68e5c8532a29e4f47028ed4141e5cdfa986483e3066665b30af4bd8
809c0a3d3f204c79a1f77ad50153065f753164f50bbb5c9b7b36ea14ce5440627b4fca1498637fd5e3b84ce48a9f
cb8822536af13edc09694740b2e01a9f5929920b197d0496709459f2ff1f6623af4e4363ebcce115aeaad4925426
803d2c45147102f898f37a67d874511d76286c76f329c50d8d10c465237273146e79f145aefed73a76d9e55f8b5a
9d1ffde3989bb6bcdb06c7b13e8f017326b726aae2ec57fbdcfe792e5051e4bfb24462339945a6a5238a4597d306
d6ed45266e757aa9328fb274cb893944850b6d5102a0549f873519a7ffb4f60a57e34d03d6bcadc74ce6248842ec
2a19392e50271cc4768ca3cdb51dc6ef42d9ac0c4ed44b130f6be4e4daa81726a4b92314740d3e3d34397748427f
8efff60e41f78088f7f143da8917d9b74843cfe384fabb070453a07e09fef49ff075646aa0c429d9267af403d67d
3346271f7d16b670966c66dcbc4f494a7e9e38f08f313d895f7e9668411dfa651a14dedd5516ce805fd5bee8e699
160cd98ac18f2a8bbc4f1f6823ce7e5fac8fee095ac19a14006737d5922fd581ee76c52d3960ef079956accc0bef
9578bcc38825120450757fbba73b5baddf3333a7b558dd65efc97032276a4b872c8bfa0846dbb6ed6c24cdb215a9
e4fb2933ab5384cdf6aa5014464e95f44db30b9bf9ea4afcc77dd489b06e433ef55addcfe21d1472e980b6462505
18ed5dd3ab72ee1b88e1f2f4347e694dfa66b6b4056095cc950ca6270addfe746ebd83cdc7ffb95f485569782567
c116e3fc8cc759d573948e7f21937d112e0678e27e81a35feecb5a5ac217b58a9e58bad76e2409844d45504d3572
32569a2038771558cb8289028833a5886490f2aa0b5343a07e939240f6aa99bbf1a3d3f3a39ec26bb4e6932c8c57
23b9e467e0743c821d79e8df896f0aceb9eb035004cd23984ef68a3b653d05ff5807b854c3ae213789ee6ebea298
71e52397ace9b40a95ce10c7ba499972137dc1685d2a326cb298c0c1e6600c327f667d46d7f3876b56da2b9e2c4b
813e5c296ea824da0abdc00182441d94a1acaa6e6561fcb5f3dfee6cfa718fc149c0c77d9600680d5f26056f23d5
b9d48e265944cc2dd1d74194cd94e1dd1e6bb027eac25654f879df190027ac94705484bd387397e818d32d73852c
1ba693dcbbef46a20437222148b08260354918ea1f911e8c96b6a461aced8d0582e710a5ca6c4aadfca17beea79f
4edb746f2e6a3e83040b5128965b1a120d1e92bd488fbd09229139b471de7378ee74ac6f9621b8a87eef8120ff90
0b1dad86a1f096caea9dbdb6559c1d459211d3b5f70da29726a4a3aff5bcf274a0206d06e85fe9ee2c7b5a4cff73
ec5c7708ae8b1c8f8e88c8eb02bf3ca76c2bf1343b8441ae9da71ba3237925f86aa0c3f36ab9871d1e1026bc1d15
7454573fbd003931ebfb3e8db2f7e2a9c5db3936724d22ca4a596d0434708538e3458f43bbaa6fdacc2f01580a46
0e93367b546c3867c04e9bc546a37cc3c095f8d19352abfab3f15310b264a41637826251a33251746a72579dba42
024f5d395e5cd48d7dfc9689715b99189bc8cc838e657e6d5a7ab2b3dc2c30ae3c786a6890439a11a10c9ad0fd6f
```

Figura P.11. ASKSIMK 3.

- Accediendo a la ruta **rutabase/cmd2/peticioncifrada** cambiando **peticioncifrada** por el resultado del comando anterior recibiremos como resultado nuestra clave **AES** y nuestro vector de inicialización para guardarlos en la smartcard. También recibiremos nuestro **ID** de miembro que deberemos introducir cuando queramos mandar mensajes con la smartcard al servidor. Primero deberemos desencriptar la respuesta como anteriormente.

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -dec 8bfc40d00791831bbbb82ca166ee7bed9a481954c631873c0e1012ceface5
373fafec114272f174d7f25809464f84b52339a639cea171ce6297d2f149b1d61c9366ba325752d45561a8779d7d648c423563cb70baea123d1d7d7b7bf04ce26dd
c2e3c2a8c2b50ee7d5c31367ad91b56e2ddb11802e2b73c9996e6b29af1a94db3d1c029c6fffa93850f6e478a566808369e13ba4c10adad3e8f3c95ba6ca8f71a67

Dec_msg:
ASKSIMKEY4: ID=0 AES_KEY=6771c887fd9507ce1fce19ea7a5a5776fd469753ddbffb99b04124d4e274bce4 AES_ICV=4b14aaac0c35cac032ce7597b36ddb64
```

Figura P.12. ASKSIMK 4.

A la hora comunicarse e introducir datos en la tarjeta, se ha utilizado actualmente el simulador JCIDE como se menciona en el apartado **3.5.2.1**. La salida de comandos utilizados para interactuar con la tarjeta se mostrará por terminal y también se guardará dentro del fichero **APDU.src** sin borrar el contenido previo de forma que se pueda ejecutar este archivo con el programa **PyApduTool** como se indica en el apartado **3.5.2.2.**. Para simular la tarjeta bastará con crearse un nuevo proyecto en JCIDE:

File->New->Project->Dar nombre al proyecto y elegir la versión de Java Card 2.2.2. o superior ->Ok.

En el nombre del paquete poner **protocolo_0_sc** y **AID** 11 22 33 44 55 66 77 88 99 00 (el AID puede ser otro pero por utilizar el mismo que se ha incluido en los comentarios del código del proyecto). A continuación pulsamos **Finish**. Ahora bastará con copiar el código del archivo **protocolo_0_sc.java** con la implementación dentro del que ha creado el IDE**.** Para simularlo pulsar el botón de **run** o Crtl+F8 y se nos abrirá una línea de comandos para comunicarnos con la tarjeta. Por defecto el AID que le debe haber asignado a nuestra applet será el del paquete en el que se encuentre añadiendo al final **00** (en este caso 11 22 33 44 55 66 77 88 99 00 00). Este AID se puede cambiar antes de la simulación desde la vista **User Package View** haciendo click derecho sobre el applet y pulsando **Change AID**. El primer comando de todos es el necesario para seleccionar nuestra applet dentro de la tarjeta que será el siguiente:

```
/send 00A404000B11223344556677889900 00
```

Donde vemos que los últimos caracteres corresponden con el AID de nuestra applet. Si el comando es correcto recibiremos **90 00** como respuesta**.**

```
>> install 1122334455667788990 01122334455667788990000 0112233445566778899000
>> 80 E6 0C 00 29 0A 11 22 33 44 55 66 77 88 99 00 0B 11 22 33 44 55 66 77 88 99 00 00 0B 11 22 33 44 55 66 77 88 99 00 00 0
<< 00 90 00

>> cardinfo
>> 80 F2 80 00 02 4F 00 00
<< 08 A0 00 00 00 03 00 00 00 01 9E 90 00

>> 80 F2 40 00 02 4F 00 00
<< 0B 11 22 33 44 55 66 77 88 99 00 00 07 00 90 00

>> 80 F2 10 00 02 4F 00 00
<< 0A 11 22 33 44 55 66 77 88 99 00 01 00 01 0B 11 22 33 44 55 66 77 88 99 00 00 90 00

Card Manager AID   :  A000000003000000
Card Manager state :  OP_READY

    Application:  SELECTABLE (--------) 0112233445566778899000
    Load File  :      LOADED (--------) 011223344556677889900
     Module    :                       0112233445566778899000

>> /send 00A404000B11223344556677889900 00
>> 00 A4 04 00 0B 11 22 33 44 55 66 77 88 99 00 00
<< 90 00
```

Figura P.13. Seleccionar Applet.

Con el programa **client.py** se podrán generar los comandos necesarios para comunicarse con la tarjeta que se explican a continuación.

Para obtener la APDU para validar el PIN de nuestra tarjeta realizamos lo siguiente:

```
$ python client.py –PIN NUMPIN
```

Donde **NUMPIN** será un número PIN de 4 dígitos.

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -PIN 1234

APDU PIN: /send 80100000021234
```

Figura P.14. APDU PIN.

Si enviamos el comando a nuestra tarjeta y el PIN es correcto (por defecto en la implementación realizada es 1234) recibiremos **90 00** como respuesta o '**6A 80 Incorrect parameters in the command data field**' si el número PIN es incorrecto. Por razones de seguridad, el estado de validación del PIN se restaura después de una petición que no sea validar el PIN. Por ello es importante tener en cuenta que al enviar cada APDU al programa de la smartcard es necesario enviar primero una APDU validando el PIN. De lo contrario recibiremos la respuesta '**69 82 Security status not satisfied**'.

Para obtener la APDU para cambiar el PIN de nuestra tarjeta realizamos lo siguiente:

```
$ python client.py –SETPIN NUEVOPIN
```

Donde **NUEVOPIN** será un número PIN de 4 dígitos.

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -SETPIN 1893

APDU SETPIN: /send 80110000021893
```

Figura P.15. APDU SETPIN.

Para obtener la APDU para cambiar la clave **AES** de 256 bits de nuestra tarjeta realizamos lo siguiente:

```
$ python client.py –SETAESKEY CLAVEAES
```

Donde **CLAVEAES** será nuestra clave **AES** en formato hexadecimal. Dado que debe ser de 256 bits = 32 bytes deben ser 64 caracteres hexadecimales. Si la longitud es incorrecta o los caracteres no son hexadecimales se informará del error.

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -SETAESKEY 6771c887fd9507ce1fce19ea7a5a5776fd469753ddbffb99b04124d4e274bce4

APDU SETAESKEY: /send 80120000206771C887FD9507CE1FCE19EA7A5A5776FD469753DDBFFB99B04124D4E274BCE4
```

Figura P.16. APDU SETAESKEY.

Para obtener la APDU para cambiar el vector de inicialización (ICV) de nuestra tarjeta realizamos lo siguiente:

```
$ python client.py –SETICV ICV
```

Donde **ICV** será nuestro vector de inicialización en formato hexadecimal. Dado que la longitud de bloque del método **AES** son 128 bits =16 bytes, ésta debe ser la longitud de nuestro ICV y por tanto son necesarios 32 caracteres hexadecimales. Si la longitud es incorrecta o los caracteres no son hexadecimales se informará del error.

```
root@coolb:~/PycharmProjects/cliente_protocolo0# python client.py -SETICV 4b14aaac0c35cac032ce7597b36ddb64

APDU SETICV: /send 80130000104B14AAAC0C35CAC032CE7597B36DDB64
```

Figura P.17. APDU SETICV.

A continuación, se muestra una imagen con toda la secuencia de comandos completa para inicializar los valores de la smartcard y un ejemplo de fallo. El orden es: seleccionar applet, cambiar el PIN, cambiar la clave, cambiar el vector y cambiar el PIN (validando el PIN en todos los casos menos al seleccionar la applet que no es necesario y al final para mostrar el error).

```
>> /send 00A404000B11223344556677889900 00
>> 00 A4 04 00 0B 11 22 33 44 55 66 77 88 99 00 00
<< 90 00

>> /send 80100000021234
>> 80 10 00 00 02 12 34
<< 90 00

>> /send 80110000021893
>> 80 11 00 00 02 18 93
<< 90 00

>> /send 80100000021893
>> 80 10 00 00 02 18 93
<< 90 00

>> /send 80120000206771C887FD9507CE1FCE19EA7A5A5776FD469753DDBFFB99B04124D4E274BCE4
>> 80 12 00 00 20 67 71 C8 87 FD 95 07 CE 1F CE 19 EA 7A 5A 57 76 FD 46 97 53 DD BF
<< 90 00

>> /send 80100000021893
>> 80 10 00 00 02 18 93
<< 90 00

>> /send 80130000104B14AAAC0C35CAC032CE7597B36DDB64
>> 80 13 00 00 10 4B 14 AA AC 0C 35 CA C0 32 CE 75 97 B3 6D DB 64
<< 90 00

>> /send 80110000021122
>> 80 11 00 00 02 11 22
<< 69 82   Security status not satisfied
```

Figura P.18. SECUENCIA APDUs.

Ahora que ya tenemos la clave y el vector en nuestra tarjeta, podemos cifrar mensajes con **AES-256-CBC**. Por simplicidad, se ha decidido que el ordenador que valida el acceso al recurso que controla el administrador del grupo enviará un mensaje de 16 bytes con una marca temporal a la tarjeta en formato '**T:YYYYmmddHHMMSS**' (año, mes, día, hora, minuto y segundos). En este caso, al ser el mensaje de la misma longitud que un bloque, el vector de inicialización sería como

una parte más de la clave. Esta decisión se ha tomado para no tener que utilizar relleno en caso de que la longitud del mensaje no fuese múltiplo de la longitud de bloque y acelerar las operaciones dentro de la smartcard pero con la idea de que sea sencillo implementar un método de relleno y mensajes más largos si fuera necesario. La tarjeta devolverá el mensaje cifrado con **AES** y posteriormente dicho ordenador lo cifrará con la clave pública **RSA** del servidor y se lo enviará. En el servidor se validará que dicha clave no ha enviado un mensaje previo con esa marca temporal (para evitar un ataque por repetición) y en ese caso se mostrará el mensaje '**Access granted**' (**Figura P.22.**). Para simular el comportamiento de este ordenador se ha creado el script de Python **reader.py** que permite validar el PIN, enviar una APDU a la smartcard pidiendo que cifre un mensaje para autenticarse en el servidor y crear un mensaje cifrado para el servidor.

Para obtener la APDU para pedir a la smartcard que cifre el mensaje utilizamos el siguiente comando:

```
$ python reader.py -AUTH
```

```
root@coolb:~/PycharmProjects/pc_sc_reader_0# python reader.py -AUTH

APDU AUTH: /send 8014000010543a32303137303533313233323533039
```

Figura P.19. APDU AUTH.

A continuación, se muestra una imagen con el comando para pedir que se cifre el mensaje enviado en la APDU (se deben incluir antes los comandos para dar valor a la clave, al vector de inicialización y al PIN si se desea ya que en cada ejecución del simulador se considera que la applet se instala desde cero en la tarjeta).

```
>> /send 80100000021893
>> 80 10 00 00 02 18 93
<< 90 00

>> /send 8014000010543a32303137303533313233323533039
>> 80 14 00 00 10 54 3A 32 30 31 37 30 35 33 31 32 33 32 35 30 39
<< 28 B0 B6 FA D3 A0 34 4B 31 E5 EB C1 2E D6 F1 6C 90 00
```

Figura P.20. AUTH smartcard.

Para el ejemplo de la imagen podemos comprobar que si tomamos la cadena devuelta **28 B0 B6 FA D3 A0 34 4B 31 E5 EB C1 2E D6 F1 6C** al desencriptarla con una herramienta online obtendremos **T:20170531232509**.

Finalmente, para acceder al recurso del servidor utilizaremos la salida del siguiente comando y navegaremos a la ruta **rutabase/cmd2/peticioncifrada** cambiando **peticioncifrada** por la salida del comando:

```
$ python reader.py -SENDCMD CMD IDUSUARIO CLAVEPUBLICARSASERVIDOR
```

Donde **CMD** será la respuesta que hayamos recibido de la smartcard sin espacios y quitando el 90 00 del final (en el ejemplo sería **28B0B6FAD3A0344B31E5EBC12ED6F16C**) y **IDUSUARIO** será el identificador que recibimos con el mensaje del servidor que nos daba la clave y el vector de inicialización.

```
root@coolb:~/PycharmProjects/pc_sc_reader_0# python reader.py -SENDCMD 28B0B6FAD3A0344B31E5EBC12ED6F16C 0 30820122300d06092a864
c683e09f2770fb872fcc226d299b6caeecb9da722a5cb0bfea35c2d2b0a0f6e7cff2aab1d8433ba1990077a1ea24e9bb5094788e34daf07b8d16551e99d3500
6e058212e281cb31fb56937d84aca75725aee46f0e819f174d13dbf6cfbaa113cfbfb7992f8027f475d63e4a58ce0356ccdc379bbfcb47798399f95a8683a88
de759a8f7965f8f402e2425efc5227050a2cdd581d8ae9a84750203010001

SENDCMD:1a119962e2827261fd944fdac1348657747d28436fb5de664dd6cd796c627d25919e0f47262da4e834b14b48127c35629815064405035d4e7d6e413
f9589391b38ce0e47a149b84f5b1d2364ec2b60938869b891cdb628fcf2716a0dcac71edbdee290f7560146c36cb19d5bab0ec41d423b1d5f7dbf681aa155a6
7f5bc825ee50f237313e4463e93e9cafaadec6455de16facec395a98f92fc252640b38d15dbe58c739c53284a4ee8ccbf5
```

Figura P.21. SENDCMD.

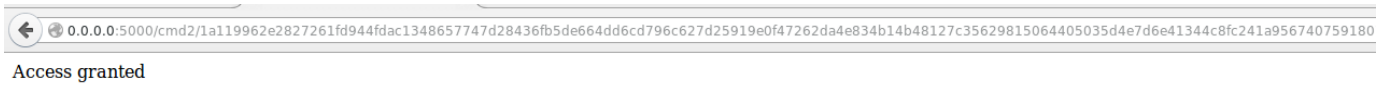

Figura P.22. Validación servidor.

P.2.2. Peticiones Protocolo 1, programa cliente y smartcard

El **Protocolo 1** tiene la misma estructura de programas que el **Protocolo 0**. Las diferencias entre ambos se encuentran en las acciones que pueden llevar a cabo estos programas. A pesar de esto, tiene algunos aspectos en común, a los que nos referiremos con el fin de no duplicar información en este documento.

Tanto el proceso de obtención de la clave pública de grupo como de la clave privada de miembro se realiza de la misma forma que en el **Protocolo 0**. Para más información se puede consultar el apartado anterior.

Una vez disponemos de la clave pública de grupo y de nuestra clave privada de miembro podremos firmar mensajes en nombre del grupo e introducir dichos mensajes en la smartcard. Para ello utilizaremos el siguiente comando:

```
$ python client.py -SETMSG
```

El cual creará un mensaje aleatorio de 100 bytes, lo firmará y nos devolverá la APDU necesaria para introducir la longitud del par (mensaje, firma) concatenado con dicho par dentro de la tarjeta.

```
root@coolb:~/PycharmProjects/cliente_protocolo1# python client.py -SETMSG

APDU SETMSG: /send 801200000008cc08ca7665615765336339713 54b63446b6f4a646551507552486e3171597a3371734f6c354d4d72457
433954714258497533 6c756a3514000000000000000016f23ca23beee01fe55dbaac6602a23723fe831f0d0000000600000005000000800000000
bd36f9a3afad5c3b57717e4412051c84d03a906d7b134d4cb86b9f3fca4e7e152a83169768c62f6350aef53fdcaf00ca2f9440ae6a72058c89
aa520b1cd77ebf9ecb50175b04334bb84550f22ccde832f2fb6b1290df8a84c5cfbdffd68b75fe9efd88bdb00e87dec4c271094b394ccdbd80
1068fdc308171ed211480000000000000000036b6c5d86b7280f412e45b5d34640e19fbef3c046275f83f269d2a46365b908631d9a2261c1c2ce
08621ac4e653802c3f87a638d4298a22f21ab1beab34051e5b14eff36ffe57dca3ff1d3a84ac15d9810000000000000000003159bbed25beab785
2b7d9052bc16f59f7b4ed7eee2142d503a6fd7182dbc88e5705e76a00db361478888a29b1f6a8e9882c2300d7a3a743340f9b94567c469c6e0
a0bdbcdb07e63a366e351be8bfe79a988000190c92052b0ed7549e406c1553ff37fb5797c68b51646ee42a55055365b99b6c63bb50136a000b
00000000000031aa61d222728e3a68f90aa17f61b5ae0368dc96721511a48f13b75b56de5b4ba1d73e0f339afea29f3de6b797c39d14e8792d
61836ea2a52f602768ca6a8e60b60cf7cc4bfdd86933d32328a5467abd810000000000000002a92df659c3d53f0435d22be7c6a99631a7842f
1f2aede177c45613caade2cd4afe9cc377b63493588a832855459bd6d482f1a13db6bd1f31224eec096afb2fc1952a79320bb657989a18c998
4aeaf62db3a19433912835c1f9c22db40137a3eab98bad69f4bb7cc218202b74135abdcc092ab7f8baf024e7db2a49243aeb0318ccaf8b97ec
2a4da4bad2dbdfbb6bb060a7fc4ff1bf5f94bccaf3c9d60f285890d95a3ed46c687bbda18c77b854e1bcf4785550023388514327 5fa2eb9e5d
cbcf855ae19bba5060356350ee19e4a199810000000000000000182e383f514f5512c1fb8dde53f67fee6dde80375e522c830525d6662a5efa6
bc0149cca501040760645b3f9731bea8ac7a909435edc36f68d0e3ea214710315283da1d61a4727c2d8ec309e5fb4ad0781000000000000000
7e95e72887906c1a9becddc0d8bf76ffaf122b20850b0bc245f3e39430f87b388f5da79e2782e4cd7b9cc94ac65b72fb923ab9848405fd5c5c
c13c64d86a23c3c13703f76e2d9a24c2a2c7b945e52a5fcff4d4e7e7065edb28df3ad4099a4747ea67319614db744c2d69815de72a2196b6c6
2dff53850a810000000000000000028cfd0cef4354f461c392ba9bf3e865ecea7e4677cc56a49941bd25b7368172f0d9a7a6fa41b1e18c6cb2c5
2c2dee694a7d6eee6ccb2603e127756f2ede415ab6206df0b6cfcada8457f776884b542aa2d20000000000000000005c9d486198c24928d0f133
6e407db71a0fc25580132555292d2000000000000000008e3e61f2805afd6da26caa3f7d1a0078a47647017b0e9a33df0e7d602ca0bc42d4000
3eec40b69db1c605426cba8513fc5d4635af3f2502a2b8000000000000000136da4a7300ca94979a9b2f3f0c5b3b41d7a4cd2470124167dfc1
bce11589405844b21979df49aae768008c0e71752ea10a1cddf6ca4c208416c30962704d37d53bc720786fc40cd57d2bf07bc2fa4d
```

Figura P.23. APDU SETMSG.

Para simular la interacción con la smartcard en este protocolo se proporciona el programa **protocolo_1_sc.java**. Del mismo modo que en el apartado anterior, se puede utilizar JCIDE como simulador y solamente es necesario crear un proyecto con un paquete llamado **protocolo_1_sc** y tener en cuenta el AID de nuestra applet para poder seleccionarla. Para el siguiente ejemplo se ha simulado con un AID para el applet de 11 22 33 44 55 66 77 88 99 00 01. Las APDUs para seleccionar el applet, validar y cambiar el PIN se construyen del mismo modo que en el protocolo anterior. A continuación, se muestra el proceso de seleccionar la applet, cambiar el PIN e introducir un mensaje en la tarjeta (con las pertinentes validaciones de PIN entre medias).

```
>> /send 00A404000B11223344556677889900 01
>> 00 A4 04 00 0B 11 22 33 44 55 66 77 88 99 00 01
<< 90 00

>> /send 80100000021234
>> 80 10 00 00 02 12 34
<< 90 00

>> /send 80110000021893
>> 80 11 00 00 02 18 93
<< 90 00

>> /send 80100000021893
>> 80 10 00 00 02 18 93
<< 90 00

>> /send 801200000008cc08ca7665615765336339713 54b63446b6f4a646551507552486e3171597a3371734f6c354d4d724578'
>> 80 12 00 00 00 08 CC 08 CA 76 65 61 57 65 33 63 39 71 35 4B 63 44 6B 6F 4A 64 65 51 50 75 52 48 6E 31 '
<< 08 CC 90 00
```

Figura P.24. SETMSG smartcard.

Se puede observar que la respuesta que devuelve es el número de bytes leídos (que coincide con los bytes enviados que se indican en los bytes 6 y 7 de la APDU) y **90 00** que indica que se ha procesado correctamente. Una vez tenemos mensajes con sus firmas dentro de la tarjeta podremos utilizarlos para autenticarnos como usuarios válidos dentro del grupo. Para ello se proporciona una modificación del programa **reader.py** del apartado anterior que permite obtener un mensaje con su firma de la tarjeta, previa validación del PIN. Estos mensajes con sus firmas se sirven de

forma secuencial volviendo al primero al haber utilizado todos. De esta forma se conserva la anonimia siempre que no se utilice el mismo par (mensaje, firma) varias veces.

```
root@coolb:~/PycharmProjects/pc_sc_reader_1# python reader.py -GETMSG

APDU GETMSG: /send 801300000209c4
```

Figura P.25. APDU GETMSG.

```
>> /send 80100000021893
>> 80 10 00 00 02 18 93
<< 90 00

>> /send 801300000209c4
>> 80 13 00 00 02 09 C4
<< 08 CA 76 65 61 57 65 33 63 39 71 35 4B 63 44 6B 6F 4A 64 65 51 50 75 52 48 6E 31 71 59 7A 33 71 73 4F
```

Figura P.26. GETMSG smartcard.

Para acceder al recurso del servidor utilizaremos la salida del siguiente comando y navegaremos a la ruta **rutabase/cmd2/peticioncifrada** cambiando **peticioncifrada** por la salida del comando:

```
$ python reader.py –SENDCMD CMD CLAVEPUBLICARSASERVIDOR
```

Donde **CMD** será la respuesta que hayamos recibido de la smartcard sin espacios (no es necesario quitar el 90 00 del final ya que se indica la longitud del mensaje al comienzo de la respuesta). Este comando se encargará de cifrar la respuesta de la smartcard con la clave pública **RSA** del servidor y añadir al mensaje una marca temporal para evitar que sea utilizado de nuevo en un futuro. En caso de que la firma sea correcta para el mensaje enviado el servidor contestará con el mensaje '**Access granted**' igual que en el **Protocolo 0**.

En este apartado se incluyen los códigos desarrollados para la implementación de los protocolos que se describen en este documento (en caso de duda contactar con alvaro.mes93@gmail.com).

Q.1. Códigos comunes de los protocolos

Q.1.1. Servidor

Q.1.1.1. Fichero BadRequest.py

```
class BadRequest(Exception):
    status_code = 400

    def __init__(self, message, status_code=None, payload=None):
        Exception.__init__(self)
        self.message = message
        if status_code is not None:
            self.status_code = status_code
        self.payload = payload

    def to_dict(self):
        rv = dict(self.payload or ())
        rv['message'] = self.message
        return rv
```

Q.1.1.2. Fichero bin_hex.py

Este fichero también se utiliza en el lado de los scripts de Python del cliente.

```
import binascii

def bin2hex(binStr):
    return binascii.hexlify(binStr)

def hex2bin(hexStr):
    return binascii.unhexlify(hexStr)
```

Q.1.1.3. Fichero run.sh

```
export FLASK_APP=group_manager_flask.py
flask run --host 0.0.0.0
```

Q.1.1.4. Fichero create_group.sh

```
#!/bin/sh
~/jdiazvico-libgroupsig-f8087dedfc5b/tools/group_create CYP06 b64 -d
CYP06_basedir -M manager -g group -m members;
cat /dev/null > CYP06_basedir/manager/crl
```

Q.1.1.5. Fichero join.sh

```
#!/bin/sh
nmemb="$1"
~/jdiazvico-libgroupsig-f8087dedfc5b/tools/join CYP06 b64
CYP06_basedir/group/grp.key CYP06_basedir/manager/mgr.key
CYP06_basedir/manager/gml CYP06_basedir/members/ $nmemb
```

Q.1.1.6. Fichero verify.sh

```
#!/bin/sh
signature_file="$1"
msg_file="$2"
group_k_file="$3"
~/jdiazvico-libgroupsig-f8087dedfc5b/tools/verify CYP06 b64 $signature_file
$msg_file $group_k_file
```

Q.1.1.7. Fichero trace.sh

```
#!/bin/sh
signature_file="$1"
grp_key="$2"
CRL="$3"
mngr_key="$4"
GML="$5"
~/jdiazvico-libgroupsig-f8087dedfc5b/tools/trace CYP06 b64 $signature_file
$grp_key $CRL $mngr_key $GML
```

Q.1.1.8. Fichero same_signer.sh

```
#!/bin/sh
sig1="$1"
sig2="$2"
grpkey="$3"
mngrkey="$4"
GML="$5"
CRL="$6"

~/jdiazvico-libgroupsig-f8087dedfc5b/tools/same_signer CYP06 $sig1 $sig2
$grpkey $mngrkey $GML $CRL
```

Q.1.1.9. Fichero revoke.sh

```
#!/bin/sh
signature_file="$1"
grp_key="$2"
CRL="$3"
mngr_key="$4"
GML="$5"
~/jdiazvico-libgroupsig-f8087dedfc5b/tools/revoke CYP06 b64 $signature_file
$grp_key $mngr_key $GML $CRL
```

Q.1.2. Cliente

Q.1.2.1. Fichero RSA_keys.py

```
from Crypto import Random
from Crypto.PublicKey import RSA
from bin_hex import *


if __name__ == "__main__":
    random_generator = Random.new().read
    key = RSA.generate(2048, random_generator)  # genero las claves
publica y privada RSA
    #Exporto las claves a formato binario
    binPrK=key.exportKey('DER')
    binPuK = key.publickey().exportKey('DER')

    #Las guardo como cadenas hexadecimales
    fPu = open('public_key.info', 'w')
    fPu.write(bin2hex(binPuK))
    fPu.close()

    fPr = open('private_key.info', 'w')
    fPr.write(bin2hex(binPrK))
    fPr.close()
```

Q.1.2.2. Fichero sign.sh

```
#!/bin/sh
signature_file="$1"
msg_file="$2"

~/jdiazvico-libgroupsig-f8087dedfc5b/tools/sign  CYP06  b64  $signature_file
$msg_file member.key grp.key
```

Q.2. Códigos Protocolo 0

Q.2.1. Servidor

Q.2.1.1. Fichero group_manager.py

```
from flask import Flask,jsonify
import subprocess
import os
import random
import string
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP,AES
from Crypto import Random
import time
from bin_hex import *
import BadRequest
import db


def setup():
    if not os.path.isdir("CYP06_basedir"):
        path=os.path.abspath("create_group.sh")
        ret=subprocess.call([path])
    path = os.path.abspath("join.sh")
    ret = subprocess.call([path, '10'])

#Dado que la clave es de 2048 bits = 256 bytes. Ahora teniendo en
cuenta el padding de SHA-1 para randominzar la encriptacion restamos
2*20+2 con lo que tenemos 256-42=214 bytes
def RSA_PKCS_OAEP_enc(key, msg):

    len_msg = len(msg)
    pos = 0
    cipher = PKCS1_OAEP.new(key)
    enc_msg = ''
    while pos<len_msg:
        if (pos + 214 < len_msg):
            enc_msg += cipher.encrypt(msg[pos:pos + 214])
        else :
            enc_msg += cipher.encrypt(msg[pos:len_msg])
        pos += 214

    return enc_msg
```

```
#Dado que la clave es de 2048 bits desencriptamos en bloques de 256
bytes
def RSA_PKCS_OAEP_dec(key, enc_msg):
    len_msg = len(enc_msg)
    pos = 0
    cipher = PKCS1_OAEP.new(key)
    msg = ''
    while pos<len_msg:
        if (pos + 256 < len_msg):
            msg += cipher.decrypt(enc_msg[pos:pos + 256])
        else :
            msg += cipher.decrypt(enc_msg[pos:len_msg])
        pos += 256

    return msg



app = Flask(__name__)
setup()
random_generator = Random.new().read
key = RSA.generate(2048, random_generator) #genero las claves publica
y privada RSA
num_claves = 0
max_claves = 10
db.create_db()
db.insert_users('usuarios.txt')

#Exporto las claves a formato binario
binPrK=key.exportKey('DER')
binPuK = key.publickey().exportKey('DER')
#Objetos con las claves RSA
PrK=RSA.importKey(binPrK)
PuK=RSA.importKey(binPuK)



@app.route('/')
def hello_world():
    return 'Protocol 0.'

@app.route('/cmd1/') #El comando 1 sera para solicitar la clave
publica RSA del servidor
def askRSAPuK():
    return bin2hex(binPuK)

@app.route('/cmd2/<string:enc_cmd>') #El comando 2 sera para el resto
de comandos que vendran encriptados con la clave publica RSA del
servidor
def cmd2(enc_cmd):
    try:
        enc_request ='%s' % enc_cmd
        request = RSA_PKCS_OAEP_dec(PrK, hex2bin(enc_request))
        command = request.split('REQUEST:')[1]
        print request
        command_list = command.split('-')
        func = command_list[0]
```

```
        if (func == 'JOIN'):
            num_claves = db.select_num_users_with_key()
            if num_claves >= max_claves:
                raise BadRequest.BadRequest('Maximum member capacity
has been reached')
            phase = int(command_list[1])
            PuK_cli = RSA.importKey(hex2bin(command_list[2]))
            if (phase == 1):
                t1 = command_list[3]
                t2 = time.time()
                #Mandamos la respuesta con el tiempo que hemos leido y
con t2 que nos tendra que devolver el usuario
                response = 'JOIN2: T1=' + t1 + '    T2= ' + str(t2)
                db.insert_join_phase(RSAPuK=com-
mand_list[2],t2=str(t2))
                return bin2hex(RSA_PKCS_OAEP_enc(PuK_cli, response))

            elif (phase == 3):
                t2 = str(command_list[3])
                #si el usuario nos devuelve nuestro tiempo le mandamos
su clave de miembro (de una de las ya generadas)
                if (db.select_join_phase(command_list[2])["t2"] ==
t2):
                    dni = str(command_list[4])
                    val = db.user_have_key(dni)
                    if val is None: #El usuario no esta en la BD
                        raise Exception
                    if not val: #El usuario esta pero aun no se ha re-
gistrado
                        db.delete_join_phase(command_list[2])
                        num_claves = db.select_num_users_with_key()
                        memk = MemKey(num_claves)
                        db.users_update_ID(dni,num_claves)
                        return bin2hex(RSA_PKCS_OAEP_enc(PuK_cli,
memk))
                    else: #Si ya se ha registrado o el DNI no es va-
lido
                        raise Exception

                else: #Si el tiempo t2 recibido no coincide con el que
guardamos
                    raise Exception

        elif (func == 'ASKGRPKEY'):
            return askGroupKey()


        elif (func == 'ASKSIMKEY'):
            phase = int(command_list[1])
            PuK_cli = RSA.importKey(hex2bin(command_list[2]))

            if (phase == 1):
                t1 = command_list[3]
                t2 = time.time()
                # Mandamos la respuesta con el tiempo que hemos leido
y con t2 que nos tendra que devolver el usuario
                response = 'ASKSIMKEY2: T1=' + t1 + '    T2= ' +
str(t2)
                db.insert_simk_phase(RSAPuK=command_list[2],
t2=str(t2))
                return bin2hex(RSA_PKCS_OAEP_enc(PuK_cli, response))
```

```
            elif (phase == 3):
                t2 = command_list[3]
                # si el usuario nos devuelve nuestro tiempo
comprobamos la firma y si es correcta le devolvemos la clave simetrica
                if (db.select_simk_phase(command_list[2])["t2"] ==
t2):
                    signature = request.split('-SIGNATURE:')[1]
                    msg = 'REQUEST:' +
request.split('REQUEST:')[1].split('-SIGNATURE:')[0]

                    check_sig_val =
check_signature(signature=signature,msg=msg)
                    if check_sig_val == 1:
                            db.delete_simk_phase(command_list[2])
                            AES_key = bin2hex(os.urandom(32))
                            AES_icv = bin2hex(os.urandom(16))
                            ID = db.select_num_registered()

db.insert_registered_users(ID_USER=ID,AES_key=AES_key,AES_icv=AES_icv,
signature=signature)
                            response = 'ASKSIMKEY4: ID=' + str(ID) + '
AES_KEY=' + AES_key + ' AES_ICV=' + AES_icv
                            return bin2hex(RSA_PKCS_OAEP_enc(PuK_cli,
response))
                    elif check_sig_val == 0:
                        #Si la clave esta revocada
                        raise BadRequest.BadRequest('REVOKED signer.')
                    else: #Si la firma no es valida
                        raise BadRequest.BadRequest('INVALID
signature.')

                else:  # Si el tiempo t2 recibido no coincide con el
que guardamos
                    raise Exception

        elif (func == 'CMD'):
            ID = str(command_list[1])
            CMD = str(command_list[2])
            reg_user = db.select_registered_user(ID_USER=ID)
            if reg_user is None:
                raise Exception
            AES_key = hex2bin(reg_user["AES_key"])
            AES_icv = hex2bin(reg_user["AES_icv"])
            cipher = AES.new(AES_key, AES.MODE_CBC, AES_icv)
            ts = cipher.decrypt(hex2bin(CMD))
            ts = ts.split('T:')[1]
            if db.select_user_interactions(ID_USER=ID,timestamp=ts) >
0: #El usuario ya realizo una interaccion previa en ese mismo instante
con esa clave luego es probable que se trate de un ataque de
repeticion
                raise Exception
            db.insert_interactions(ID_USER=ID,timestamp=ts)

            return 'Access granted'

        else:
            raise BadRequest.BadRequest('Unkwon request')
    except Exception:
        raise BadRequest.BadRequest('The request was invalid or
malformed')
```

```
@app.errorhandler(BadRequest.BadRequest)
def badRequest(exception):
    response = jsonify(exception.to_dict())
    response.status_code = exception.status_code
    return response
#Comprueba que una firma es valida y no pertenece a un miembro
revocado. Devuelve -1 si no es valida 0 si esta revocada y 1 si es
valida y no esta revocada
def check_signature(signature,msg):
    sig_file = 'signature' +
''.join(random.choice(string.ascii_uppercase) for _ in range(6)) +
'.txt'
    while os.path.isfile(sig_file):
        sig_file = 'signature' +
''.join(random.choice(string.ascii_uppercase) for _ in range(6)) +
'.txt'
    f = open(sig_file, 'w+')
    f.write(signature)
    f.close()

    msg_file = 'msg' + ''.join(random.choice(string.ascii_uppercase)
for _ in range(6)) + '.txt'
    while os.path.isfile(msg_file):
        msg_file = 'msg' +
''.join(random.choice(string.ascii_uppercase) for _ in range(6)) +
'.txt'

    f = open(msg_file, 'w+')
    f.write(msg)
    f.close()
    path = os.path.abspath("verify.sh")
    # Tomo el resultado del script que verifica si se trata de una
firma valida
    ret = subprocess.check_output([path, sig_file, msg_file,
'CYP06_basedir/group/grp.key'])
    os.remove(msg_file)
    if (ret == 'VALID signature.\n'):
        path = os.path.abspath("trace.sh")
        ret = subprocess.check_output([path, sig_file,
'CYP06_basedir/group/grp.key', 'CYP06_basedir/manager/crl',

'CYP06_basedir/manager/mgr.key', 'CYP06_basedir/manager/gml'])
        os.remove(sig_file)
        if (ret == 'VALID signer.\n'):
            return 1
        else:#Revoked
            return 0
    else:#Invalid
        return -1


def askGroupKey():
    f = open('CYP06_basedir/group/grp.key', 'r')
    grk = f.read()
    return grk


def MemKey(n):
    f = open('CYP06_basedir/members/'+str(n)+'.key', 'r')
    memk = f.read()
    return memk
```

Q.2.1.2. Fichero db.py

```
import dataset
import os

def create_db():
    if os.path.isfile('protocolo0.db'):
        os.remove('protocolo0.db')
    db = dataset.connect('sqlite:///protocolo0.db')
    db.create_table('JOIN_PHASE', primary_id='RSAPuK',
primary_type='String')
    db.create_table('USERS', primary_id='DNI', primary_type='String')
    db.create_table('REGISTERED_USERS', primary_id='ID_USER',
primary_type='Integer')
    db.create_table('SIMK_PHASE', primary_id='RSAPuK',
primary_type='String')
    db.create_table('INTERACTIONS', primary_id='ID_USER',
primary_type='Integer')
    insert_interactions(-1,'00000000000000')

def insert_join_phase(RSAPuK,t2):
    db = dataset.connect('sqlite:///protocolo0.db')
    jp = db['JOIN_PHASE']
    jp.insert(dict(RSAPuK=RSAPuK, t2=t2))

def insert_simk_phase(RSAPuK,t2):
    db = dataset.connect('sqlite:///protocolo0.db')
    jp = db['SIMK_PHASE']
    jp.insert(dict(RSAPuK=RSAPuK, t2=t2))

def insert_interactions(ID_USER,timestamp):
    db = dataset.connect('sqlite:///protocolo0.db')
    interactions = db['INTERACTIONS']
    interactions.insert(dict(ID_USER=ID_USER, timestamp=timestamp))

def delete_join_phase(RSAPuK):
    db = dataset.connect('sqlite:///protocolo0.db')
    jp = db['JOIN_PHASE']
    jp.delete(RSAPuK=RSAPuK)

def delete_simk_phase(RSAPuK):
    db = dataset.connect('sqlite:///protocolo0.db')
    jp = db['SIMK_PHASE']
    jp.delete(RSAPuK=RSAPuK)

def insert_users(filename):
    f=open(filename,'r')
    dnis = f.readlines()
    f.close()
    db = dataset.connect('sqlite:///protocolo0.db')
    users = db['USERS']
    for dni in dnis:
        dni = dni.replace('\n','')
        users.insert(dict(DNI=dni, ID_USER=-1))

def insert_registered_users(ID_USER,AES_key,AES_icv,signature):
    db = dataset.connect('sqlite:///protocolo0.db')
    reg_users = db['REGISTERED_USERS']
    reg_users.insert(dict(ID_USER=ID_USER,
AES_key=AES_key,AES_icv=AES_icv,signature=signature))
```

```
def select_num_users_with_key():
    db = dataset.connect('sqlite:///protocolo0.db')
    count = db.query('SELECT COUNT(*) c FROM USERS WHERE ID_USER!=-1')
    return int(count.next()['c'])

def select_num_registered():
    db = dataset.connect('sqlite:///protocolo0.db')
    count = db.query('SELECT COUNT(*) c FROM REGISTERED_USERS')
    return int(count.next()['c'])

def db_get_users():
    usrs = []
    db = dataset.connect('sqlite:///protocolo0.db')
    users = db['USERS'].all()
    for user in users:
        usrs.append(user)
    return usrs

def users_update_ID(dni,ID):
    db = dataset.connect('sqlite:///protocolo0.db')
    users = db['USERS']
    users.update(dict(DNI=dni, ID_USER=ID), ['DNI'])

def user_have_key(dni):
    db = dataset.connect('sqlite:///protocolo0.db')
    user = db['USERS'].find_one(DNI=dni)
    if user is None:
        return
    elif user['ID_USER']==-1:
        return False
    else:
        return True

def select_join_phase(RSAPuK):
    db = dataset.connect('sqlite:///protocolo0.db')
    jp = db['JOIN_PHASE'].find_one(RSAPuK=RSAPuK)
    return jp

def select_simk_phase(RSAPuK):
    db = dataset.connect('sqlite:///protocolo0.db')
    skp = db['SIMK_PHASE'].find_one(RSAPuK=RSAPuK)
    return skp

def select_registered_user(ID_USER):
    db = dataset.connect('sqlite:///protocolo0.db')
    reg_u = db['REGISTERED_USERS'].find_one(ID_USER=ID_USER)
    return reg_u

def select_user_interactions(ID_USER,timestamp):
    db = dataset.connect('sqlite:///protocolo0.db')
    inter = db.query('SELECT COUNT(*) c FROM INTERACTIONS WHERE
ID_USER='+str(ID_USER)+' AND timestamp=\''+timestamp+'\'')
    return int(inter.next()['c'])
```

Q.2.2. Cliente

Q.2.2.1. Fichero client.py

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import os
import sys
from bin_hex import *
import time
import subprocess

#Dado que la clave es de 2048 bits = 256 bytes. Ahora teniendo en
cuenta el padding de SHA-1 para randominzar la encriptacion restamos
2*20+2 con lo que tenemos 256-42=214 bytes
def RSA_PKCS_OAEP_enc(key, msg):
    len_msg = len(msg)
    pos = 0
    cipher = PKCS1_OAEP.new(key)
    enc_msg = ''
    while pos<len_msg:
        if (pos + 214 < len_msg):
            enc_msg += cipher.encrypt(msg[pos:pos + 214])
        else :
            enc_msg += cipher.encrypt(msg[pos:len_msg])
        pos += 214

    return enc_msg

#Dado que la clave es de 2048 bits desencriptamos en bloques de 256
bytes
def RSA_PKCS_OAEP_dec(key, enc_msg):
    len_msg = len(enc_msg)
    pos = 0
    cipher = PKCS1_OAEP.new(key)
    msg = ''
    while pos<len_msg:
        if (pos + 256 < len_msg):
            msg += cipher.decrypt(enc_msg[pos:pos + 256])
        else :
            msg += cipher.decrypt(enc_msg[pos:len_msg])
        pos += 256

    return msg
```

```
if __name__ == "__main__":
    if len(sys.argv)<2:
        print('\nFaltan argumentos\n')

    elif sys.argv[1]=='-enc' and len(sys.argv)==4:
        key = RSA.importKey(hex2bin(sys.argv[2]))
        func = sys.argv[3]
        output = bin2hex(RSA_PKCS_OAEP_enc(key, func))
        f = open('encrypted.txt', 'w')
        f.write(output)
        f.close()
        print ('\nEnc_msg:\n' + output)

    elif sys.argv[1] == '-dec' and len(sys.argv) == 3:
        enc_msg = sys.argv[2]
        f = open('private_key.info', 'r')
        PrK = RSA.importKey(hex2bin(f.read()))
        f.close()
        output = RSA_PKCS_OAEP_dec(PrK, hex2bin(enc_msg))
        f = open('decrypted.txt', 'w')
        f.write(output)
        f.close()
        print ('\nDec_msg:\n' + output)

    elif sys.argv[1] == '-GRPKEY' and len(sys.argv) == 3:
        key = RSA.importKey(hex2bin(sys.argv[2]))
        output = bin2hex(RSA_PKCS_OAEP_enc(key, 'REQUEST:ASKGRPKEY'))
        f = open('ASKGROUPKEY.txt', 'w')
        f.write(output)
        f.close()
        print ('\nASKGROUPKEY REQUEST:\n' + output + '\n')

    #-JOIN 1 PuK-- Devuelve la primera fase de JOIN y el T1 incluido
en la peticion encriptada
    #-JOIN 3 PuK T2 DNI-- Devuelve la tercera fase de JOIN incluyendo
el T2 contestado por el GM y nuestra identificacion
    elif sys.argv[1] == '-JOIN' and len(sys.argv) in (4,6):
        phase = int(sys.argv[2])
        key = RSA.importKey(hex2bin(sys.argv[3]))
        if phase == 1 and len(sys.argv) == 4 :
            t1=time.time()
            f = open('public_key.info', 'r')
            PuK = f.read()
            f.close()
            output = bin2hex(RSA_PKCS_OAEP_enc(key, 'REQUEST:JOIN-1-'
+ PuK + '-' + str(t1)))
            f = open('JOIN.txt', 'w')
            f.write(output)
            f.close()
            print ('\nJOIN1 REQUEST:\n' + output + '\n\nT1:\n' +
str(t1))

        elif phase == 3 and len(sys.argv) == 6 :
            t2 = sys.argv[4]
            dni = sys.argv[5]
            f = open('public_key.info', 'r')
            PuK = f.read()
            f.close()
            output = bin2hex(RSA_PKCS_OAEP_enc(key, 'REQUEST:JOIN-3-'
+ PuK + '-' + t2 + '-' + dni))
```

```
            f = open('JOIN.txt', 'w')
            f.write(output)
            f.close()
            print ('\nJOIN3 REQUEST:\n' + output)

        else:
            print '\nArgumentos incorrectos\n'

    #-SIMK 1 PuK-- Devuelve la primera fase de ASKSIMK y el T1 in-
cluido en la peticion encriptada
    #-SIMK 3 PuK T2-- Devuelve la tercera fase de ASKSIMK incluyendo
el T2 contestado por el GM. Aqui se incluye el mensaje firmado
    elif sys.argv[1] == '-SIMK' and len(sys.argv) in (4, 5):
        if os.path.isfile('member.key') and os.path.isfile('grp.key'):
            phase = int(sys.argv[2])
            key = RSA.importKey(hex2bin(sys.argv[3]))
            if phase == 1 and len(sys.argv) == 4:
                t1 = time.time()
                f = open('public_key.info', 'r')
                PuK = f.read()
                f.close()
                output = bin2hex(RSA_PKCS_OAEP_enc(key, 'RE-
QUEST:ASKSIMKEY-1-' + PuK + '-' + str(t1)))
                f = open('ASKSIMK.txt', 'w')
                f.write(output)
                f.close()
                print ('\nASKSIMK1 REQUEST:\n' + output + '\n\nT1:\n'
+ str(t1))

            elif phase == 3 and len(sys.argv) == 5:
                t2 = sys.argv[4]
                f = open('public_key.info', 'r')
                PuK = f.read()
                f.close()
                #escribimos el mensaje que queremos firmar
                f = open('mensaje.txt', 'w+')
                msg = 'REQUEST:ASKSIMKEY-3-' + PuK + '-' + t2
                f.write(msg)
                f.close()
                f = open('firma.txt', 'w+')
                f.close()
                #firmamos el mensaje y lo guardamos en firma.txt
                path = os.path.abspath("sign.sh")
                subprocess.call([path,'firma.txt','mensaje.txt'])
                #leemos la firma y encriptamos la respuesta al GM
                f = open('firma.txt', 'r')
                sign = f.read()
                f.close()
                output = bin2hex(RSA_PKCS_OAEP_enc(key, msg + '-SIGNA-
TURE:'+sign))
                f = open('ASKSIMK.txt', 'w')
                f.write(output)
                f.close()
                print (
                '\nASKSIMK3 REQUEST:\n' + output)

            else:
                print '\nArgumentos incorrectos\n'
```

```
        else:
            print('\nFalta el fichero member.key o el fichero
grp.key\n')

    elif sys.argv[1] == '-SETPIN' and len(sys.argv) == 3:
        PIN = str(sys.argv[2])
        numbers = True
        for n in PIN:
            if n not in '1234567890':
                numbers = False
        if len(PIN)!=4 or not numbers:
            print('\nEl PIN debe contener 4 numeros\n')
            exit()
        output = '8011000002' + PIN
        f = open('APDU.src', 'a')
        f.write(output + ';\n')
        f.close()
        print ('\nAPDU SETPIN: /send ' + output + '\n')

    elif sys.argv[1] == '-PIN' and len(sys.argv) == 3:
        PIN = str(sys.argv[2])
        numbers = True
        for n in PIN:
            if n not in '1234567890':
                numbers = False
                break
        if len(PIN) != 4 or not numbers:
            print('\nEl PIN debe contener 4 numeros\n')
            exit()
        output = '8010000002' + PIN
        f = open('APDU.src', 'a')
        f.write(output + ';\n')
        f.close()
        print ('\nAPDU PIN: /send ' + output + '\n')

    elif sys.argv[1] == '-SETICV' and len(sys.argv) == 3:
        ICV = str(sys.argv[2])
        hexa = True
        for h in ICV:
            if h not in '1234567890ABCDEFabcdef':
                hexa = False
                break
        if len(ICV)!= 32 or not hexa:
            print('\nEl ICV deben ser 32 caracteres hexadecimales(16
bytes)\n')
            exit()
        output = '8013000010' + ICV.lower()
        f = open('APDU.src', 'a')
        f.write(output + ';\n')
        f.close()
        print ('\nAPDU SETICV: /send '+ output + '\n')

    elif sys.argv[1] == '-SETAESKEY' and len(sys.argv) == 3:
        AESK = str(sys.argv[2])
        hexa = True
        for h in AESK:
            if h not in '1234567890ABCDEFabcdef':
                hexa = False
                break
```

```
        if len(AESK) != 64 or not hexa:
            print('\nLa clave AES debe ser de 64 caracteres hexadeci-
males (32 bytes)\n')
            exit()
        output = '8012000020' + AESK.lower()
        f = open('APDU.src', 'a')
        f.write(output + ';\n')
        f.close()
        print ('\nAPDU SETAESKEY: /send '+ output + '\n')

    else:
        print('\nFaltan argumentos\n')

    exit()
```

Q.2.2.2. Fichero protocolo_0_sc.java

```
/*
 * @file  protocolo_0_sc.java
 * @version v1.0
 * Package AID: 11 22 33 44 55 66 77 88 99
 * Applet AID:  11 22 33 44 55 66 77 88 99 00
 */


package protocolo_0_sc;

import javacard.framework.*;
import javacardx.crypto.*;
import javacard.security.*;
import javacard.framework.OwnerPIN;

public class protocolo_0_sc extends Applet
{

    private static final byte CLA_PRO0 = (byte)0x80;
    private static final byte INS_VERIFY_PIN = (byte)0x10;
    private static final byte INS_SET_PIN = (byte)0x11;
    private static final byte INS_SET_AES_KEY = (byte)0x12;
    private static final byte INS_SET_AES_ICV = (byte)0x13;
    private static final byte INS_CMD = (byte)0x14;


    private byte [] INIT_PIN = new byte[]{(byte)0x12, (byte)0x34};

    //Creamos un objeto tipo OwnerPIN con 3 intentos y 4 dígitos de longitud
    OwnerPIN pin;
```

```
    private byte AESKeyLen;
    private byte[] AES_Key;
    private byte[] AES_ICV;
    private Cipher AES_CBC;
    private Key AES_Key_builder;

     public protocolo_0_sc()
    {
        AES_Key = new byte[32];
        AES_ICV = new byte[16];
        AESKeyLen = 0;
        pin = new OwnerPIN((byte) 3, (byte) 4);
        pin.update(INIT_PIN, (short) 0, (byte) INIT_PIN.length);
        //Creamos el cifrador de AES CBC.
        AES_CBC = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_CBC_NOPAD, false);
             //Creaamos las claves para AES sin inicializar
        AES_Key_builder =
KeyBuilder.buildKey(KeyBuilder.TYPE_AES_TRANSIENT_DESELECT,
KeyBuilder.LENGTH_AES_256, false);

    }

      public static void install(byte[] bArray, short bOffset, byte bLength)
      {
            new protocolo_0_sc().register(bArray, (short) (bOffset + 1),
bArray[bOffset]);
      }

      public void process(APDU apdu)
      {
            if (selectingApplet()){
                  return;
            }
            if( pin.getTriesRemaining() == 0 ){
                        return;
            }
            byte[] buf = apdu.getBuffer();
            short len = apdu.setIncomingAndReceive();

            //Si la clase no es la del protocolo 0 devolvemos el error
            if(buf[ISO7816.OFFSET_CLA] != CLA_PRO0){
                  ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
            }
            switch (buf[ISO7816.OFFSET_INS]){
                  case INS_VERIFY_PIN:
                        verify(apdu, len);
                        break;
                  case INS_SET_PIN:
                        setPIN(apdu, len);
                        break;
                  case INS_SET_AES_KEY:
                        setAESKey(apdu, len);
                        break;
```

```
                case INS_SET_AES_ICV:
                        setAESICV(apdu, len);
                        break;
                case INS_CMD:
                        CMD(apdu, len);
                        break;
              default:
                    ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
             }
       }

   //Modificar la clave AES
      private void setAESKey(APDU apdu, short len)
      {

            if (!pin.isValidated()) {

      ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
            }

            byte[] buffer = apdu.getBuffer();
            //Si la clave no es de 256 bits devolvemos un error de longitud
            if (len != 32) {
                  ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
            }

            JCSystem.beginTransaction();
            //Guardo el valor de la clave AES
            Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, AES_Key, (short)0,
len);
            AESKeyLen = (byte)32;
            JCSystem.commitTransaction();
            pin.reset();
      }

   //Modifica el vector de inicialización de AES CBC
      private void setAESICV(APDU apdu, short len)
      {
            if (!pin.isValidated()) {

      ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
            }

            if (len != 16)
            {
                  ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
            }
            JCSystem.beginTransaction();
            Util.arrayCopy(apdu.getBuffer(), ISO7816.OFFSET_CDATA, AES_ICV,
(short)0, (short)16);
            JCSystem.commitTransaction();
            pin.reset();
      }

   //Devolvemos un objeto Key con la clave AES.
```

```
	private Key getAESKey()
	{
		Key tempAESKey = AES_Key_builder;

		((AESKey)tempAESKey).setKey(AES_Key, (short)0);
		return tempAESKey;
	}

	private void setPIN(APDU apdu, short len){
		if (!pin.isValidated()) {

	ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
		}
		if (len != 2)
		{
			ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
		}
		byte[] buffer = apdu.getBuffer();
		byte [] PIN = JCSystem.makeTransientByteArray((short) 2,
JCSystem.CLEAR_ON_RESET);
		JCSystem.beginTransaction();
		Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, PIN, (short)0, len);
		pin.update(PIN, (short) 0, (byte) len);
		JCSystem.commitTransaction();
		pin.reset();
	}


   //En el comando vamos a recibir una cadena multiplo de 16 a modo de challenge
que tenemos que encriptar con AES
	private void CMD(APDU apdu, short len)
	{
	   if (!pin.isValidated()) {

	ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
		}

		if (len <= 0 || len % 16 != 0)
		{
			ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
		}

		byte[] buffer = apdu.getBuffer();
		byte mode = buffer[ISO7816.OFFSET_P1] == (byte)0x00 ?
Cipher.MODE_ENCRYPT : Cipher.MODE_DECRYPT;
		AESCipher(apdu, mode, len);
		pin.reset();
	}
```

```
	//AES CBC
	private void AESCipher(APDU apdu, byte mode, short len)
	{
		Key key = getAESKey();
		byte[] buffer = apdu.getBuffer();
		AES_CBC.init(key, mode, AES_ICV, (short)0, (short)16);
		AES_CBC.doFinal(buffer, ISO7816.OFFSET_CDATA, len, buffer,
(short)0);
		apdu.setOutgoingAndSend((short)0, len);
	}

	private void verify(APDU apdu, short len) {
		if (len != 2)
		{
			ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
		}
		byte[] buffer = apdu.getBuffer();
		byte [] PIN = JCSystem.makeTransientByteArray((short) 2,
JCSystem.CLEAR_ON_RESET);
		JCSystem.beginTransaction();
		Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, PIN, (short)0, len);
		boolean check_pin = pin.check(PIN, (short)0, (byte)len);
		JCSystem.commitTransaction();
		if (check_pin == false ){
			ISOException.throwIt(ISO7816.SW_WRONG_DATA);
		}
	}


}
```

Q.2.3. Ordenador de acceso con lector de tarjetas

Q.2.3.1. Fichero reader.py

```
import sys
import datetime
import time
import binascii
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP



def RSA_PKCS_OAEP_enc(key, msg):
    len_msg = len(msg)
    pos = 0
    cipher = PKCS1_OAEP.new(key)
    enc_msg = ''
```

```
    while pos<len_msg:
        if (pos + 214 < len_msg):
            enc_msg += cipher.encrypt(msg[pos:pos + 214])
        else :
            enc_msg += cipher.encrypt(msg[pos:len_msg])
        pos += 214

    return enc_msg


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print('\nFaltan argumentos\n')

    elif sys.argv[1] == '-AUTH' and len(sys.argv) == 2:
        ts = time.time()
        st =
datetime.datetime.fromtimestamp(ts).strftime('%Y%m%d%H%M%S')
        #este mensaje tiene 2 caracteres con lo que mandamos a la SC
un mensaje con el timestamp para que no se pueda repetir el cifrado y
de 16 bytes = 128 bits que es la longitud de un bloque AES
        #Los 16 bytes son 2 del mensaje + 14 del timestamp
(YYYYmmddHHMMSS)
        msg=('T:'+st).encode("hex")
        output = '8014000010' + msg
        f = open('APDU.src', 'a')
        f.write(output + ';\n')
        f.close()
        print ('\nAPDU AUTH: /send ' + output + '\n')

    elif sys.argv[1] == '-SENDCMD' and len(sys.argv) == 5:
        CMD = str(sys.argv[2]).lower()
        ID = str(sys.argv[3])
        key = RSA.importKey(binascii.unhexlify(sys.argv[4]))
        output = binascii.hexlify(RSA_PKCS_OAEP_enc(key,'REQUEST:CMD-
'+str(ID)+'-'+CMD))
        f = open('CMD.txt', 'w')
        f.write(output)
        f.close()
        print ('\nSENDCMD:' + output + '\n')

    elif sys.argv[1] == '-PIN' and len(sys.argv) == 3:
        PIN = str(sys.argv[2])
        numbers = True
        for n in PIN:
            if n not in '1234567890':
                numbers = False
                break
        if len(PIN) != 4 or not numbers:
            print('\nEl PIN debe contener 4 numeros\n')
            exit()
        output = '8010000002' + PIN
        f = open('APDU.src', 'a')
        f.write(output + ';\n')
        f.close()
        print ('\nAPDU PIN: /send ' + output + '\n')
    else:
        print('\nFaltan argumentos\n')
```

Q.3. Códigos Protocolo 1

Q.3.1. Servidor

Q.3.1.1. Fichero group_manager.py

```
from flask import Flask,jsonify
import subprocess
import os
import random
import string
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto import Random
import time
from bin_hex import *
import BadRequest
import db


def setup():
    if not os.path.isdir("CYP06_basedir"):
        path=os.path.abspath("create_group.sh")
        ret=subprocess.call([path])
    path = os.path.abspath("join.sh")
    ret = subprocess.call([path, '10'])



#Dado que la clave es de 2048 bits = 256 bytes. Ahora teniendo en
cuenta el padding de SHA-1 para randominzar la encriptacion restamos
2*20+2 con lo que tenemos 256-42=214 bytes


def RSA_PKCS_OAEP_enc(key, msg):

    len_msg = len(msg)
    pos = 0
    cipher = PKCS1_OAEP.new(key)
    enc_msg = ''
    while pos<len_msg:
        if (pos + 214 < len_msg):
            enc_msg += cipher.encrypt(msg[pos:pos + 214])
        else :
            enc_msg += cipher.encrypt(msg[pos:len_msg])
        pos += 214

    return enc_msg
```

```
#Dado que la clave es de 2048 bits desencriptamos en bloques de 256
bytes
def RSA_PKCS_OAEP_dec(key, enc_msg):
    len_msg = len(enc_msg)
    pos = 0
    cipher = PKCS1_OAEP.new(key)
    msg = ''
    while pos<len_msg:
        if (pos + 256 < len_msg):
            msg += cipher.decrypt(enc_msg[pos:pos + 256])
        else :
            msg += cipher.decrypt(enc_msg[pos:len_msg])
        pos += 256

    return msg



app = Flask(__name__)
setup()
random_generator = Random.new().read
key = RSA.generate(2048, random_generator) #genero las claves publica
y privada RSA
num_claves = 0
max_claves = 10
db.create_db()
db.insert_users('usuarios.txt')

#Exporto las claves a formato binario
binPrK=key.exportKey('DER')
binPuK = key.publickey().exportKey('DER')
#Objetos con las claves RSA
PrK=RSA.importKey(binPrK)
PuK=RSA.importKey(binPuK)



@app.route('/')
def hello_world():
    return 'Protocolo 1.'

@app.route('/cmd1/') #El comando 1 sera para solicitar la clave
publica RSA del servidor
def askRSAPuK():
    return bin2hex(binPuK)

@app.route('/cmd2/<string:enc_cmd>') #El comando 2 sera para el resto
de comandos que vendran encriptados con la clave publica RSA del
servidor
def cmd2(enc_cmd):
    #try:
        enc_request ='%s' % enc_cmd
        request = RSA_PKCS_OAEP_dec(PrK, hex2bin(enc_request))
        command = request.split('REQUEST:')[1]
        print request
        command_list = command.split('-')
        func = command_list[0]
```

```
        if (func == 'JOIN'):
            num_claves = db.select_num_users_with_key()
            if num_claves >= max_claves:
                raise BadRequest.BadRequest('Maximum member capacity
has been reached')
            phase = int(command_list[1])
            PuK_cli = RSA.importKey(hex2bin(command_list[2]))
            if (phase == 1):
                t1 = command_list[3]
                t2 = time.time()
                #Mandamos la respuesta con el tiempo que hemos leido y
con t2 que nos tendra que devolver el usuario
                response = 'JOIN2: T1=' + t1 + '    T2= ' + str(t2)

db.insert_join_phase(RSAPuK=command_list[2],t2=str(t2))
                #join_phase[command_list[2]]=str(t2)
                return bin2hex(RSA_PKCS_OAEP_enc(PuK_cli, response))

            elif (phase == 3):
                t2 = str(command_list[3])
                #si el usuario nos devuelve nuestro tiempo le mandamos
su clave de miembro (de una de las ya generadas)
                if (db.select_join_phase(command_list[2])["t2"] ==
t2):
                    dni = str(command_list[4])
                    val = db.user_have_key(dni)
                    if val is None: #El usuario no esta en la BD
                        raise Exception
                    if not val: #El usuario esta pero aun no se ha
registrado
                        db.delete_join_phase(command_list[2])
                        num_claves = db.select_num_users_with_key()
                        memk = MemKey(num_claves)
                        db.users_update_ID(dni,num_claves)
                        return bin2hex(RSA_PKCS_OAEP_enc(PuK_cli,
memk))
                    else: #Si ya se ha registrado o el DNI no es
valido
                        raise Exception

                else: #Si el tiempo t2 recibido no coincide con el que
guardamos
                    raise Exception

        elif (func == 'ASKGRPKEY'):
            return askGroupKey()

        elif (func == 'CMD'):
            ts = request.split('-T:')[1].split('-MSG:')[0]
            msg = request.split('-MSG:')[1].split('-SIGNATURE:')[0]
            signature = request.split('-SIGNATURE:')[1]
            check_sig_val = check_signature(signature=signature,
msg=msg)
```

```
            if check_sig_val == 1:
                if
db.select_user_interactions(signature=signature,timestamp=ts) > 0:  #
El usuario ya realizo una interaccion previa en ese mismo instante con
esa clave luego es probable que se trate de un ataque de repeticion
                    raise Exception
                db.insert_interactions(signature=signature,
timestamp=ts)
                return 'Access granted'
            elif check_sig_val == 0:
                # Si la clave esta revocada
                raise BadRequest.BadRequest('REVOKED signer.')
            else:  # Si la firma no es valida
                raise BadRequest.BadRequest('INVALID signature.')

        else:
            raise BadRequest.BadRequest('Unkwon request')
    #except Exception:
    #    raise BadRequest.BadRequest('The request was invalid or
malformed')



@app.errorhandler(BadRequest.BadRequest)
def badRequest(exception):
    response = jsonify(exception.to_dict())
    response.status_code = exception.status_code
    return response


#Comprueba que una firma es valida y no pertenece a un miembro
revocado. Devuelve -1 si no es valida 0 si esta revocada y 1 si es
valida y no esta revocada
def check_signature(signature,msg):
    sig_file = 'signature' +
''.join(random.choice(string.ascii_uppercase) for _ in range(6)) +
'.txt'
    while os.path.isfile(sig_file):
        sig_file = 'signature' +
''.join(random.choice(string.ascii_uppercase) for _ in range(6)) +
'.txt'
    f = open(sig_file, 'w+')
    f.write(signature)
    f.close()

    msg_file = 'msg' + ''.join(random.choice(string.ascii_uppercase)
for _ in range(6)) + '.txt'
    while os.path.isfile(msg_file):
        msg_file = 'msg' +
''.join(random.choice(string.ascii_uppercase) for _ in range(6)) +
'.txt'

    f = open(msg_file, 'w+')
    f.write(msg)
    f.close()
    path = os.path.abspath("verify.sh")
    # Tomo el resultado del script que verifica si se trata de una
firma valida
```

```
    ret = subprocess.check_output([path, sig_file, msg_file,
'CYP06_basedir/group/grp.key'])
    os.remove(msg_file)
    if (ret == 'VALID signature.\n'):
        path = os.path.abspath("trace.sh")
        ret = subprocess.check_output([path, sig_file,
'CYP06_basedir/group/grp.key', 'CYP06_basedir/manager/crl',

'CYP06_basedir/manager/mgr.key', 'CYP06_basedir/manager/gml'])
        os.remove(sig_file)
        if (ret == 'VALID signer.\n'):
            return 1
        else:#Revoked
            return 0
    else:#Invalid
        return -1

def askGroupKey():
    f = open('CYP06_basedir/group/grp.key', 'r')
    grk = f.read()
    return grk


def MemKey(n):
    f = open('CYP06_basedir/members/'+str(n)+'.key', 'r')
    memk = f.read()
    return memk
```

Q.3.1.2. Fichero db.py

```
import dataset
import os


def create_db():
    if os.path.isfile('protocolo1.db'):
        os.remove('protocolo1.db')
    db = dataset.connect('sqlite:///protocolo1.db')
    db.create_table('JOIN_PHASE', primary_id='RSAPuK',
primary_type='String')
    db.create_table('USERS', primary_id='DNI', primary_type='String')
    db.create_table('INTERACTIONS', primary_id='SIGNATURE',
primary_type='String')
    insert_interactions('signature', '00000000000000')
```

```
def insert_join_phase(RSAPuK,t2):
    delete_join_phase(RSAPuK) #En caso de que repita la consulta para
entrar el mismo miembro
    db = dataset.connect('sqlite:///protocolo1.db')
    jp = db['JOIN_PHASE']
    jp.insert(dict(RSAPuK=RSAPuK, t2=t2))

def insert_interactions(signature,timestamp):
    db = dataset.connect('sqlite:///protocolo1.db')
    interactions = db['INTERACTIONS']
    interactions.insert(dict(SIGNATURE=signature,
timestamp=timestamp))

def delete_join_phase(RSAPuK):
    db = dataset.connect('sqlite:///protocolo1.db')
    jp = db['JOIN_PHASE']
    jp.delete(RSAPuK=RSAPuK)

def insert_users(filename):
    f=open(filename,'r')
    dnis = f.readlines()
    f.close()
    db = dataset.connect('sqlite:///protocolo1.db')
    users = db['USERS']
    for dni in dnis:
        dni = dni.replace('\n','')
        users.insert(dict(DNI=dni, ID_USER=-1))


def select_num_users_with_key():
    db = dataset.connect('sqlite:///protocolo1.db')
    count = db.query('SELECT COUNT(*) c FROM USERS WHERE ID_USER!=-1')
    return int(count.next()['c'])


def db_get_users():
    usrs = []
    db = dataset.connect('sqlite:///protocolo1.db')
    users = db['USERS'].all()
    for user in users:
        usrs.append(user)
    return usrs

def users_update_ID(dni,ID):
    db = dataset.connect('sqlite:///protocolo1.db')
    users = db['USERS']
    users.update(dict(DNI=dni, ID_USER=ID), ['DNI'])

def user_have_key(dni):
    db = dataset.connect('sqlite:///protocolo1.db')
    user = db['USERS'].find_one(DNI=dni)
    if user is None:
        return
    elif user['ID_USER']==-1:
        return False
    else:
        return True
```

```
def select_join_phase(RSAPuK):
    db = dataset.connect('sqlite:///protocolo1.db')
    jp = db['JOIN_PHASE'].find_one(RSAPuK=RSAPuK)
    return jp

def select_user_interactions(signature,timestamp):
    db = dataset.connect('sqlite:///protocolo1.db')
    inter = db.query('SELECT COUNT(*) c FROM INTERACTIONS WHERE
SIGNATURE=\''+str(signature)+'\' AND timestamp=\''+timestamp+'\'')
    return int(inter.next()['c'])
```

Q.3.2. Cliente

Q.3.2.1. Fichero client.py

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import os
import sys
from bin_hex import *
import time
import subprocess
import datetime
import random
import string


#Dado que la clave es de 2048 bits = 256 bytes. Ahora teniendo en
cuenta el padding de SHA-1 para randominzar la encriptacion restamos
2*20+2 con lo que tenemos 256-42=214 bytes


def RSA_PKCS_OAEP_enc(key, msg):
    len_msg = len(msg)
    pos = 0
    cipher = PKCS1_OAEP.new(key)
    enc_msg = ''
    while pos<len_msg:
        if (pos + 214 < len_msg):
            enc_msg += cipher.encrypt(msg[pos:pos + 214])
        else :
            enc_msg += cipher.encrypt(msg[pos:len_msg])
        pos += 214

    return enc_msg
```

```
#Dado que la clave es de 2048 bits desencriptamos en bloques de 256
bytes
def RSA_PKCS_OAEP_dec(key, enc_msg):
    len_msg = len(enc_msg)
    pos = 0
    cipher = PKCS1_OAEP.new(key)
    msg = ''
    while pos<len_msg:
        if (pos + 256 < len_msg):
            msg += cipher.decrypt(enc_msg[pos:pos + 256])
        else :
            msg += cipher.decrypt(enc_msg[pos:len_msg])
        pos += 256

    return msg


if __name__ == "__main__":
    if len(sys.argv)<2:
        print('\nFaltan argumentos\n')

    elif sys.argv[1]=='-enc' and len(sys.argv)==4:
        key = RSA.importKey(hex2bin(sys.argv[2]))
        func = sys.argv[3]
        output = bin2hex(RSA_PKCS_OAEP_enc(key, func))
        f = open('encrypted.txt','w')
        f.write(output)
        f.close()
        print ('\nEnc_msg:\n'+output)

    elif sys.argv[1] == '-dec' and len(sys.argv) == 3:
        enc_msg = sys.argv[2]
        f = open('private_key.info', 'r')
        PrK = RSA.importKey(hex2bin(f.read()))
        f.close()
        output = RSA_PKCS_OAEP_dec(PrK, hex2bin(enc_msg))
        f = open('decrypted.txt', 'w')
        f.write(output)
        f.close()
        print ('\nDec_msg:\n' + output)

    elif sys.argv[1] == '-GRPKEY' and len(sys.argv) == 3:
        key = RSA.importKey(hex2bin(sys.argv[2]))
        output = bin2hex(RSA_PKCS_OAEP_enc(key, 'REQUEST:ASKGRPKEY'))
        f = open('ASKGROUPKEY.txt', 'w')
        f.write(output)
        f.close()
        print ('\nASKGROUPKEY REQUEST:\n' + output + '\n')

    #-JOIN 1 PuK-- Devuelve la primera fase de JOIN y el T1 incluido
en la peticion encriptada
    #-JOIN 3 PuK T2 DNI-- Devuelve la tercera fase de JOIN incluyendo
el T2 contestado por el GM y nuestra identificacion
```

```
    elif sys.argv[1] == '-JOIN' and len(sys.argv) in (4,6):
        phase = int(sys.argv[2])
        key = RSA.importKey(hex2bin(sys.argv[3]))
        if phase == 1 and len(sys.argv) == 4 :
            t1=time.time()
            f = open('public_key.info', 'r')
            PuK = f.read()
            f.close()
            output = bin2hex(RSA_PKCS_OAEP_enc(key, 'REQUEST:JOIN-1-'
+ PuK + '-' + str(t1)))
            f = open('JOIN.txt', 'w')
            f.write(output)
            f.close()
            print ('\nJOIN1 REQUEST:\n' + output + '\n\nT1:\n'+
str(t1))

        elif phase == 3 and len(sys.argv) == 6 :
            t2 = sys.argv[4]
            dni = sys.argv[5]
            f = open('public_key.info', 'r')
            PuK = f.read()
            f.close()
            output = bin2hex(RSA_PKCS_OAEP_enc(key, 'REQUEST:JOIN-3-'
+ PuK + '-' +t2 + '-' + dni))
            f = open('JOIN.txt', 'w')
            f.write(output)
            f.close()
            print ('\nJOIN3 REQUEST:\n' + output)

        else:
            print '\nArgumentos incorrectos\n'


    elif sys.argv[1] == '-SETMSG' and len(sys.argv) == 2:
        if os.path.isfile('member.key') and os.path.isfile('grp.key'):
            # escribimos el mensaje que queremos firmar
            f = open('mensaje.txt', 'w+')
            msg =
''.join(random.SystemRandom().choice(string.ascii_letters +
string.digits) for _ in range(100))
            f.write(msg)
            f.close()
            f = open('firma.txt', 'w+')
            f.close()
            # firmamos el mensaje y lo guardamos en firma.txt
            path = os.path.abspath("sign.sh")
            subprocess.call([path, 'firma.txt', 'mensaje.txt'])
            # leemos la firma y encriptamos la respuesta al GM
            f = open('firma.txt', 'r')
            sign = f.read()
            f.close()
            hexamsg = msg.encode('hex')
            lenhexamsg = len(hexamsg)
            hexasign=sign.decode('base64').encode('hex')
            lenhexasign = len(hexasign)
            lenhexa = format(((lenhexamsg+lenhexasign)/2)+2,'04x')#2
bytes hexadecimales con la longitud de 2+mensaje+firma siendo los dos
primeros para la longitud
```

```
            output = '8012000000'+ lenhexa +
format((lenhexamsg+lenhexasign)/2,'04x') + bin2hex(msg.encode('utf-
8'))+bin2hex(sign.decode('base64'))
            f = open('APDU.src', 'a')
            f.write(output+';\n')
            f.close()
            print ('\nAPDU SETMSG: /send ' + output)

        else:
            print('\nFalta el fichero member.key o el fichero
grp.key\n')


    elif sys.argv[1] == '-SETPIN' and len(sys.argv) == 3:
        PIN = str(sys.argv[2])
        numbers = True
        for n in PIN:
            if n not in '1234567890':
                numbers = False
                break
        if len(PIN)!=4 or not numbers:
            print('\nEl PIN debe contener 4 numeros\n')
            exit()
        output = '8011000002' + PIN
        f = open('APDU.src', 'a')
        f.write(output + ';\n')
        f.close()
        print ('\nAPDU SETPIN: /send '+ output + '\n')

    elif sys.argv[1] == '-PIN' and len(sys.argv) == 3:
        PIN = str(sys.argv[2])
        numbers = True
        for n in PIN:
            if n not in '1234567890':
                numbers = False
                break
        if len(PIN) != 4 or not numbers:
            print('\nEl PIN debe contener 4 numeros\n')
            exit()
        output = '8010000002' + PIN
        f = open('APDU.src', 'a')
        f.write(output + ';\n')
        f.close()
        print ('\nAPDU PIN: /send ' + output + '\n')

    else:
        print('\nFaltan argumentos\n')

    exit()
```

Q.3.2.2. Fichero protocolo_1_sc.java

```
/*
 * @file  protocolo_1_sc.java
 * @version v1.0
 * Package AID: 11 22 33 44 55 66 77 88 99 00
 * Applet AID:  11 22 33 44 55 66 77 88 99 00 01
 */


package protocolo_1_sc;

import javacard.framework.*;
import javacardx.crypto.*;
import javacard.security.*;
import javacard.framework.OwnerPIN;
import javacard.framework.Util;
import javacard.framework.APDU;
import javacardx.apdu.ExtendedLength;
import java.lang.Object;
import java.lang.ArrayIndexOutOfBoundsException;

public class protocolo_1_sc extends Applet implements ExtendedLength
{

      private static final byte CLA_PRO1                 = (byte)0x80;
      private static final byte INS_VERIFY_PIN     = (byte)0x10;
      private static final byte INS_SET_PIN  = (byte)0x11;
      private static final byte INS_SET_MSG  = (byte)0x12;
    private static final byte INS_GET_MSG    = (byte)0x13;
    private byte [] INIT_PIN = new byte[]{(byte)0x12, (byte)0x34};

    //Creamos un objeto tipo OwnerPIN con 3 intentos y 4 dígitos de longitud
    OwnerPIN pin;

    private short Num_Signatures;
    private short Max_Signatures;
    private short Num_Dispached;
    private short Index_Update;
    Object[] signatures;


     public protocolo_1_sc()
    {
        Num_Signatures = 0;
        Max_Signatures = 3;
        Num_Dispached = 0;
        Index_Update = 0;
        signatures = new Object[Max_Signatures];
```

```
        for(short i=0;i<Max_Signatures;i++){
              signatures[i] = new byte[2500];
        }
        pin = new OwnerPIN((byte) 3, (byte) 4);
        pin.update(INIT_PIN, (short) 0, (byte) INIT_PIN.length);
    }


      public static void install(byte[] bArray, short bOffset, byte bLength)
      {
            new protocolo_1_sc().register(bArray, (short) (bOffset + 1),
bArray[bOffset]);
      }

      public void process(APDU apdu)
      {

            if (selectingApplet())
        {
            return;
        }

        if( pin.getTriesRemaining() == 0 ){
                  return;
            }
        byte[] buff = apdu.getBuffer();
        byte cla= buff[ISO7816.OFFSET_CLA];
        byte ins = buff[ISO7816.OFFSET_INS];
        byte p1 = buff[ISO7816.OFFSET_P1];
        byte p2 = buff[ISO7816.OFFSET_P2];
        short p1p2 = Util.makeShort(p1, p2);

        if (cla != CLA_PRO1)
        {
            ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
        }

        if (p1p2 != 0x00)
        {
            ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        }

        short recvLen = apdu.setIncomingAndReceive();

        //Tomamos la longitud de los datos entrantes
        short lc = apdu.getIncomingLength();

        if (lc == 0x00)
        {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
```

```
        switch (ins){
            case INS_VERIFY_PIN:
                verify(apdu, recvLen);
                break;
            case INS_SET_PIN:
                setPIN(apdu, recvLen);
                break;
            case INS_SET_MSG:
                addSignature(apdu, recvLen, lc);
                break;
            case INS_GET_MSG:
                getSignature(apdu);
                break;
            default:
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }

  //Guarda un mensaje firmado en la SC
    private void addSignature(APDU apdu, short recvLen, short lc)
    {
          if (!pin.isValidated()) {

    ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
          }
          if (Index_Update == Max_Signatures) {
                Index_Update = (short)0;
          }
          byte [] buffer = apdu.getBuffer();
          short pointer = 0;
          short offData = apdu.getOffsetCdata();

          //Guardo los datos
          while (recvLen > (short) 0)
          {
                Util.arrayCopy(buffer, offData, (byte
[])signatures[Index_Update], pointer, recvLen);
                pointer += recvLen;
                //Tomo todos los bytes posibles del buffer
                recvLen = apdu.receiveBytes(offData);
          }
          if(Num_Signatures<Max_Signatures){
                Num_Signatures+=1;
          }
          Index_Update+=1;

          // send the lc length
          apdu.setOutgoing();
          apdu.setOutgoingLength((short)2);
          Util.setShort(buffer, (short)0, lc);
          apdu.sendBytesLong(buffer, (short) 0, (short)2);
          pin.reset();
    }
```

```
	private void setPIN(APDU apdu, short recvLen){
		if (!pin.isValidated()) {

	ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
		}
		if (recvLen != 2)
		{
			ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
		}
		byte[] buffer = apdu.getBuffer();
		byte [] PIN = JCSystem.makeTransientByteArray((short) 2,
JCSystem.CLEAR_ON_RESET);
		JCSystem.beginTransaction();
		Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, PIN, (short)0,
recvLen);
		pin.update(PIN, (short) 0, (byte) recvLen);
		JCSystem.commitTransaction();
		pin.reset();
	}


   //Devuelve un mensaje firmado aleatorio de la SC
	private void getSignature(APDU apdu)
	{
		if (!pin.isValidated()) {

	ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
		}
		byte [] buffer = apdu.getBuffer();
		if(Num_Signatures == (short)0)
		{
			ISOException.throwIt(ISO7816.SW_FILE_NOT_FOUND);
		}
		if(buffer[ISO7816.OFFSET_LC] != 0x02)
		{
			ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
		}
		if (Num_Dispached == Max_Signatures || Num_Dispached ==
Num_Signatures){
			JCSystem.beginTransaction();
			Num_Dispached = 0;
			JCSystem.commitTransaction();
		}
		short sendLen = Util.getShort(buffer, ISO7816.OFFSET_CDATA);
		apdu.setOutgoing();
		apdu.setOutgoingLength(sendLen);
		apdu.sendBytesLong((byte [])signatures[Num_Dispached], (short)0,
sendLen);
		Num_Dispached+=1;
		pin.reset();
	}
```

```
	private void verify(APDU apdu, short recvLen){
		if (recvLen != 2)
		{
			ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
		}
		byte[] buffer = apdu.getBuffer();
		byte [] PIN = JCSystem.makeTransientByteArray((short) 2,
JCSystem.CLEAR_ON_RESET);
		JCSystem.beginTransaction();
		Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, PIN, (short)0,
recvLen);
		JCSystem.commitTransaction();
		boolean check_pin = pin.check(PIN, (short)0, (byte)recvLen);
		if (check_pin == false ){
			ISOException.throwIt(ISO7816.SW_WRONG_DATA);
		}

	}


}
```

Q.3.3. Ordenador de acceso con lector de tarjetas

Q.3.3.1. Fichero reader.py

```
import sys
import datetime
import time
import binascii
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP


def RSA_PKCS_OAEP_enc(key, msg):
    len_msg = len(msg)
    pos = 0
    cipher = PKCS1_OAEP.new(key)
    enc_msg = ''
    while pos<len_msg:
        if (pos + 214 < len_msg):
            enc_msg += cipher.encrypt(msg[pos:pos + 214])
        else :
            enc_msg += cipher.encrypt(msg[pos:len_msg])
        pos += 214

    return enc_msg
```

```
if __name__ == "__main__":
    if len(sys.argv) < 2:
        print('\nFaltan argumentos\n')

    elif sys.argv[1] == '-GETMSG' and len(sys.argv) == 2:
        output = '801300000209c4'
        f = open('APDU.src', 'a')
        f.write(output + ';\n')
        f.close()
        print ('\nAPDU GETMSG: /send ' + output + '\n')


    elif sys.argv[1] == '-SENDCMD' and len(sys.argv) == 4:
        msg_sig = str(sys.argv[2]).lower()
        msg_len = int(msg_sig[0:4],16)
        msg_sig = msg_sig[4:4+2*msg_len] #La longitud esta en bytes
pero el mensaje en hexa por eso contamos por dos.
        key = RSA.importKey(binascii.unhexlify(sys.argv[3]))
        ts = time.time()
        st =
datetime.datetime.fromtimestamp(ts).strftime('%Y%m%d%H%M%S')
        # este mensaje tiene 2 caracteres con lo que mandamos a la SC
un mensaje con el timestamp para que no se pueda repetir el cifrado y
de 16 bytes = 128 bits que es la longitud de un bloque AES
        # Los 16 bytes son 2 del mensaje + 14 del timestamp
(YYYYmmddHHMMSS)
        msg = msg_sig[:200].decode('hex')
        sign = msg_sig[200:].decode('hex').encode('base64')
        output = binascii.hexlify(RSA_PKCS_OAEP_enc(key,'REQUEST:CMD-
T:'+str(st)+'-MSG:'+msg+'-SIGNATURE:'+sign))
        f = open('CMD.txt', 'w')
        f.write(output)
        f.close()
        print ('\nSENDCMD:' + output + '\n')

    elif sys.argv[1] == '-PIN' and len(sys.argv) == 3:
        PIN = str(sys.argv[2])
        numbers = True
        for n in PIN:
            if n not in '1234567890':
                numbers = False
                break
        if len(PIN) != 4 or not numbers:
            print('\nEl PIN debe contener 4 numeros\n')
            exit()
        output = '8010000002' + PIN
        f = open('APDU.src', 'a')
        f.write(output + ';\n')
        f.close()
        print ('\nAPDU PIN: /send ' + output + '\n')
    else:
        print('\nFaltan argumentos\n')
```