UNIVERSIDAD DEL BÍO-BÍO

FACULTAD DE CIENCIAS EMPRESARIALES

DEPARTAMENTO DE INGENIERÍA CIVIL EN INFORMÁTICA

UNIVERSITAT POLITÈCNICA DE CATALUNYA

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA DE TELECOMUNICACIÓ DE BARCELONA

UNIVERSIDAD DEL BÍO-BÍO

"Implementación en Java de una librería para la creación de firma grupal apta para cloud collaboration"

Trabajo final de grado para la obtención del título

GRADO EN INGENIERÍA DE SISTEMAS DE TELECOMUNICACIÓN

Alumno

Jordi Arumí Pauls

Profesor guía

Patricio Galdames Sepúlveda

PRESENTACIÓN DEL TRABAJO

En este trabajo se presenta un esquema de firma digital para la firma de documentos por parte de un colectivo de usuarios, con una serie de restricciones a nivel de anonimato y de formación de dicha firma. En primer lugar, se hace una introducción donde se explica la problemática y el contexto en el cual se desarrolla el proyecto. A continuación, se comentan los objetivos del proyecto y los requerimientos que debe cumplir el esquema. Seguidamente, se encuentra un análisis general de distintas técnicas de firma grupal, las más relevantes de las cuales se detallan y analizan teniendo en cuenta las condiciones que debe cumplir el esquema deseado. Posteriormente, se comparan dichos esquemas, con el fin de escoger el más apropiado de ellos para los objetivos requeridos. El algoritmo escogido es explicado en profundidad a continuación para, finalmente, proceder con la implementación de una librería en Java para dicho esquema, detallando las funciones y métodos contenidos en ella y algunas pruebas de test para la demostración del funcionamiento de ésta.

ÍNDICE

1. INTRODUCCIÓN

Cloud collaboration es un servicio en línea que permite compartir y editar archivos digitales a través de una nube computacional. Este servicio busca facilitar la edición colaborativa de archivos a través de usuarios que se encuentran dispersos en la Internet. La forma tradicional en la que se implementa este servicio consiste en configurar uno o varios servidores dedicados que materializan una "nube". Esta nube gestiona el almacenamiento y controla el acceso a los archivos.

Sin embargo, esta estructura centralizada presenta diversas problemáticas. Primero, sin un adecuado número de servidores, algunos de estos servidores podrían sobrecargarse al recibir una alta demanda de atención por parte de un gran número de clientes. De ese modo, estos servidores se convierten en potenciales cuellos de botella. Segundo, los enlaces de acceso a estos servidores pudieran también presentar altos periodos de congestión, afectando a los tiempos de actualización percibidos por los usuarios. Si bien el número de servidores como también el ancho de banda de los servidores críticos pudiera ser incrementado, aumentando el costo del servicio, es posible que haya situaciones donde estos recursos de red y de servidores sean sub-utilizados, y por ende sean desperdiciados. Tercero, los usuarios no tienen control del lugar en la nube en el cual sus documentos son almacenados, por lo que la confidencialidad e integridad de los documentos pudiera ser comprometida.

Para sobrellevar los desafíos de escalabilidad se ha propuesto en la literatura académica que la implementación del servicio de cloud collaboration sea soportada por una red P2P. En este contexto, se asume que los mismos usuarios que desean colaborar en la edición de un documento conforman la red P2P. Diversas arquitecturas P2P se han propuesto para organizar a los usuarios. La que se asume en este trabajo es del tipo anillo. Para preservar la confidencialidad e integridad de los documentos, se asume la existencia de algún algoritmo de criptografía asimétrica cuya selección se pretende abordar en este trabajo.

2. DEFINICIÓN DEL PROYECTO

2.1. Objetivos del proyecto

Objetivo general:

Implementar una librería en Java que permita generar una llave grupal con un subgrupo de usuarios dentro de la red, atendiendo a los requerimientos demandados por un servicio de cloud collaboration, y respondiendo a unos requisitos específicos que son comentados más adelante.

Objetivos específicos:

- Búsqueda bibliográfica de distintas técnicas y algoritmos de encriptación para la realización de firmas grupales.
- Análisis de los algoritmos encontrados y valoración de cada uno de ellos respecto a los requerimientos demandados.
- Comparación de los esquemas más cercanos al deseado, atendiendo a los requerimientos demandados.
- Decisión del esquema más cercano a los requerimientos demandados.
- Implementación de dicho esquema en Java.

2.2. Requerimientos del esquema

Supongamos que tenemos una red P2P formada por *n* usuarios conectados a la Internet, los cuales editan colaborativamente archivos que comparten y los hacen públicos tan pronto sean estos finalizados. Para este último propósito, hay un servidor externo independiente a los miembros del grupo. Este servidor hace públicos los documentos y sus respectivas firmas digitales con el fin de probar la

autoría de los documentos. También certifica el origen de las claves publicas asociadas a un grupo.

El algoritmo de firma grupal de un documento debe cumplir con los siguientes requisitos:

1. Un número “t” cualquiera de los “n” miembros del grupo actúa en representación de todos los miembros del grupo y colaborativamente crea una firma grupal para un archivo dado. Si un número menor a “t” usuarios firma el documento, esta firma no tendrá validez.

2. Cualquier usuario puede verificar la validez de la firma grupal incluida en un documento. Del proceso de verificación no debe inferirse la identidad de quienes firmaron digitalmente el documento.

3. La firma grupal incluida en un documento debe poseer un “trap-door” que solo es conocido por un ente externo y debe permitirle conocer la identidad de los miembros que firmaron el documento. Se asume que tal ente externo mantiene en secreto la información de trap-door y entrega la identidad de los creadores de una firma bajo circunstancias especiales. Por ejemplo, si la información es conflictiva (por ejemplo, constituye un delito) se puede liberar su origen bajo una orden judicial.

3. ANÁLISIS DE DISTINTOS ESQUEMAS DE FIRMA GRUPAL

Varias técnicas de criptografía se usan para generar firmas grupales sin la necesidad de que todos los miembros del grupo firmen. En las "group signatures" [1], un usuario del grupo puede firmar en representación del mismo, habiendo un mánager encargado de generar dicha firma y con la capacidad de identificar al firmante en caso de necesidad. En las "multi-signatures" [2][3], por su parte, es un subgrupo "t" del total de "n" usuarios o bien un solo usuario el que crea la firma, conociéndose sus identidades. A partir de las group signatures, pero ampliando el número de usuarios necesarios para generar la firma, aparecieron las "threshold signatures" [4][5][6]. Existen threshold signatures basadas en distintas técnicas de encriptación, como RSA [7], Gap Diffie-Hellman (GDH) [8] o ID-based [9]. La mayoría de estos trabajos no hablan de la posibilidad de revocación de la identidad de los firmantes. Sin embargo, en [10], se presenta una técnica de revocación para algunos esquemas de ID-based threshold signatures. En las "proxy signatures" [9][11], por su lado, un grupo externo al de los firmantes se encarga de crear la firma, sin tener la potestad de firmar nada. En el caso de las "ring signatures" [12], un usuario del grupo puede firmar en representación de los demás, sin la existencia de ningún mánager o autoridad que cree la firma, y sin la posibilidad de revocación. En las "linkable ring signatures" [13], se puede saber si dos mensajes distintos han sido firmados por el mismo usuario. Basadas en las ring signatures, las "threshold ring signatures" [14] requieren un mínimo de "t" usuarios para generar la firma. Como ocurre con las threshold signatures, algunos de estos esquemas son ID-based [15][16][17]. La mayoría de ring signatures y threshold ring signatures no permiten o no contemplan la existencia de un ente externo que pueda revocar el anonimato de los firmantes. En [17], sin embargo, se explica una técnica para añadir una autoridad de confianza que pueda hacerlo.

A continuación se analizan algunos de los esquemas citados anteriormente, teniendo en cuenta qué requerimientos cumplen y cuáles no de los que se necesitan para este proyecto. En concreto, se analizan los tres requisitos mencionados con anterioridad, eso es, el número de firmantes, el anonimato de los

firmantes (tanto en el proceso de firma como en el de verificación de ésta) y la existencia de trapp-door.

3.1. Group signatures

En las group signatures, cada miembro "i" del grupo crea su llave secreta "S_i". A partir de esta llave, un generador "g" crea "$g^{S_i} \pmod p$" ("p" es un número primo), la cual es entregada a una autoridad de confianza (también llamada mánager del grupo) que no pertenece al conjunto de posibles firmantes. De esta forma, el mánager posee una lista con todas estas llaves, junto con la identidad de cada miembro del grupo. Cada cierto tiempo, el mánager da a cada usuario del grupo un número elegido al azar "r_i". Con este número, se genera la llave pública de cada usuario, de la forma "$(g^{S_i})^{r_i}$". Durante este periodo de tiempo, el miembro "i" del grupo usa como llave privada "$S_i r_i \pmod{p-1}$". Con este mecanismo, el mánager no conoce ni puede conocer las llaves privadas de cada usuario y, por tanto, no puede firmar con su firma, pero, en caso de necesidad, puede saber quién ha sido el firmante de cierto mensaje. Además, el mánager publica la lista completa de las correspondientes llaves públicas (en orden aleatorio) en un directorio público de confianza, de tal manera que el verificador puede saber si el mensaje proviene o no del grupo.

Propiedades:

1. Número de firmantes:
   El mensaje lo firma un miembro cualquiera del grupo.

2. Verificación y anonimato:
   El receptor del mensaje puede verificar que el mensaje proviene del grupo deseado, sin saber quién ha sido el firmante. Tampoco los miembros del grupo pueden saber quién ha firmado el mensaje.

3. Revocación (trapp-door):
   El mánager puede comprobar quién ha sido el firmante de un mensaje.

En resumen, vemos que el primer requisito no se cumple en este tipo de esquemas, ya que solo 1 usuario firma el mensaje en nombre de todo el grupo. El anonimato de dicho firmante se respeta tanto en el proceso de firma como en el de verificación, cosa que cumple con el segundo requisito. Sólo el mánager puede llegar a conocer la identidad del firmante. Eso cumple con el tercer objetivo. Eso sí, ese mánager participa en la formación de las firmas de los miembros, aunque sin saber las llaves secretas de los mismos, cosa que le impide firmar ningún mensaje en representación del grupo.

3.2. Multi-signatures

Todos los miembros del grupo ("n") conocen una llave privada general y una llave pública general. Mediante un protocolo que todos conocen, cada usuario crea su propia llave privada. Tanto pueden estar diseñadas para que pueda firmar un usuario como para que se necesite un mínimo de "t" usuarios para que un mensaje lleve la firma del grupo. Los firmantes firman con su llave privada, su identidad y la llave pública general. El verificador conoce la llave de grupo y las identidades de los miembros, y con ello verifica que el mensaje viene del grupo, sabiendo quién lo ha firmado y comprobando que la firma es válida.

Propiedades:

1. Número de firmantes:
   Añadiendo un umbral, es necesario que por lo menos "t" usuarios firmen.

2. Verificación y anonimato:
   La validez de una firma se verifica sabiendo las identidades de los miembros del grupo, con lo cual no se respeta su pleno anonimato.

3. Revocación (trapp-door):
   Dado que tanto el verificador como los miembros del grupo pueden conocer las identidades de los firmantes, es innecesario un ente externo para tal fin.

Así pues, las multi-signatures pueden llegar a cumplir el primero de los requisitos si se les añade la propiedad de umbral, pero como vemos, no se cumplen ni el segundo ni el tercer requisito, dado que el anonimato de los firmantes queda totalmente de lado, ya que tanto los participantes en la firma como el resto de usuarios del grupo y también el verificador pueden conocer la identidad de los mismos. Este hecho provoca que no sea necesaria una trapp-door.

3.3. Threshold signatures

Basadas en las group signatures, las threshold signatures requieren de un mínimo de "t" usuarios para generar la firma. A pesar de que en muchos de los trabajos sobre este tipo de esquemas no se habla de la posibilidad de revocación, la mayoría de ellos tienen la posibilidad (como pasa con las group signatures) de ser rastreadas por el mánager con el fin de conocer la identidad de los firmantes de un mensaje.

Propiedades:

1. Número de firmantes
   Se requieren "t" de los "n" miembros del grupo para generar la firma.

2. Verificación y anonimato
   De la misma forma que en las group signatures, se preserva la identidad de los firmantes, tanto de cara a los miembros del grupo (firmantes o no) como de cara al verificador.

3. Revocación (trapp-door)

   El mánager puede conocer la identidad de los firmantes con el esquema adecuado.

Por tanto, los esquemas de threshold signatures se acercan bastante a cumplir los requerimientos. Se requiere un umbral para generar la firma, el anonimato de los firmantes esta a salvo tanto en el proceso de firma como en el de verificación, y el esquema puede ser abierto en muchos casos por el mánager para conocer las identidades de los firmantes.

3.4. Ring signatures

Se trata de una red de posibles firmantes “n” con 1 firmante. Dicho firmante firma con las llaves públicas “P_1”, “P_2”, ..., “P_n” y con su privada “S_s”, siendo “s” la posición aleatoria del firmante dentro del anillo. El algoritmo produce la firma de anillo "σ" a partir de estas llaves introducidas por el firmante, el mensaje y la posición del firmante dentro del anillo (“m”, “P_1”, “P_2”, ..., “P_n”, “S_s”, “s”, "σ"). El verificador conoce "σ". Al recibir el mensaje, introduce "σ". El algoritmo, a partir del mensaje y la llave privada del firmante (“m”, “S_s”), le devuelve al verificador "verdadero" o "falso", en función de si la firma de anillo es o no es válida (es decir, si el mensaje viene del grupo de usuarios deseado o no). De esta manera, el verificador sabe si el mensaje proviene del grupo de usuarios que le interesa sin saber quién ha sido el firmante (el verificador no conoce en ningún momento “S_s”). De hecho, ni los propios miembros del grupo pueden saber quién ha sido el firmante (si introducen "σ" recibirán, al igual que el verificador, únicamente "verdadero" o "falso").

Este tipo de esquemas fueron diseñados para que un usuario del grupo pueda firmar un mensaje en representación del grupo, sin la presencia de un mánager que genere las firmas ni de una autoridad de confianza que pueda revocar el anonimato del firmante.

Propiedades:

1. Número de firmantes:
   Se requiere 1 único usuario de los “n” miembros del grupo para generar la firma.

2. Verificación y anonimato:
   Un verificador que conozca "σ" siempre recibirá de forma correcta "verdadero" o "falso", nunca recibirá una respuesta equivocada y, por tanto, nunca creerá que un mensaje proviene del grupo deseado si no es así ni viceversa, pero no podrá conocer en ningún caso la identidad de los firmantes. Tampoco los miembros del grupo podrán saber dicha información.

3. Revocación (trapp-door):
   No existe mánager en las ring signatures, ni tampoco un miembro externo que pueda romper el anonimato de los firmantes.

Como se puede ver, es un esquema muy fuerte a nivel de anonimato, cosa que cumple con el segundo requisito. Eso sí, el hecho de que solamente firme un miembro del grupo y la inexistencia de trapp-door incumple el primer y tercer requisitos.

3.5. Threshold ring signatures

Son la versión con umbral de las ring signatures. A la hora de generar la firma, al menos “t” usuarios deben firmar con su clave privada para que dicha firma se genere. Tampoco cuentan con mánager del grupo. Existen threshold ring signatures (también ocurre con las ring signatures) basadas en distintas técnicas. De la misma forma que con las ring signatures, estos esquemas sirven para preservar al máximo la identidad de los firmantes, no dando la posibilidad en la

mayoría de esquemas de revocar el anonimato de los mismos. Sin embargo, en las ID-based threshold ring signatures, existen algunas técnicas para incluir "identity escrow", eso es, se da a un ente externo al grupo (llamado "autoridad de confianza") la potestad de romper ese anonimato y conocer la identidad de los firmantes, solamente en casos específicos. Ningún miembro del grupo ni verificador puede hacer tal cosa, solamente la autoridad de confianza.

Propiedades:

1. Número de firmantes:
   Son necesarios al menos "t" miembros del grupo para generar la firma.

2. Verificación y anonimato:
   De la misma forma que en las ring signatures, el verificador recibe "verdadero" o "falso", pero en ningún caso sabe qué miembros del grupo han firmado el mensaje. Los miembros del grupo, sean o no firmantes, tampoco pueden saberlo.

3. Revocación (trapp-door):
   A pesar de que la mayoría de esquemas no lo permiten, alguno sí que da la posibilidad de trapp-door.

Resumiendo, con algunos esquemas (como los mencionados ID-based threshold ring signature) una autoridad de confianza externa al grupo de posibles firmantes puede revocar el anonimato para conocer las identidades de los firmantes de un mensaje en concreto. En estos casos, se cumplen los tres requisitos demandados, dado que el primero y segundo se cumplen para todos los esquemas de threshold ring signature.

3.6. Comparación de los esquemas

En esta tabla se comparan los cinco esquemas explicados con anterioridad, atendiendo a si cumplen o no los requisitos de este proyecto:

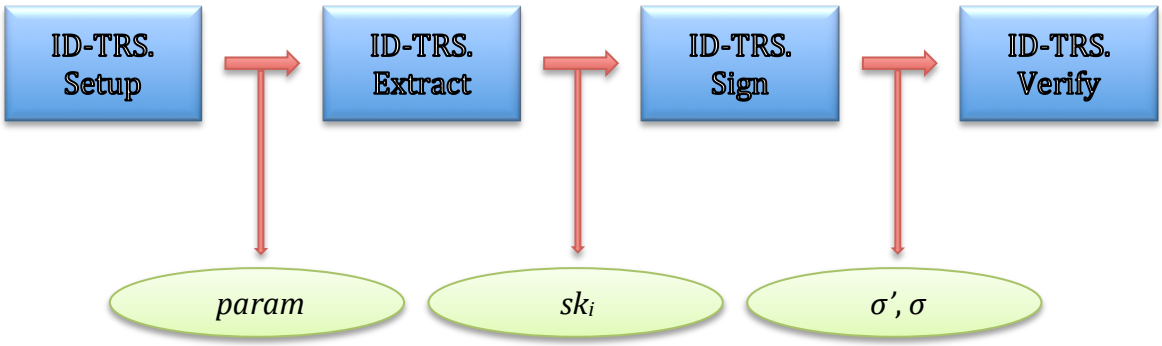
Esquema	Número mínimo de firmantes "t"	Anonimato	Posibilidad de revocación
Group signatures [1]	No	Sí	Sí
Multi-signatures [2][3]	Sí	No	No
Threshold signatures [4]-[11]	Sí	Sí	Sí
Ring signatures [12][13]	No	Sí	No
Threshold ring signatures [14]-[17]	Sí	Sí	Sí

Como se puede ver, las threshold signatures y las threshold ring signatures pueden cumplir con los requisitos de este proyecto. Atendiendo a la robustez de las ring signatures y teniendo en cuenta la buena preservación del anonimato que en estos esquemas se consigue, el esquema escogido es un threshold ring signature. Concretamente, por la capacidad de revocar el anonimato que presenta, se ha escogido como algoritmo a implementar el "ID-based threshold ring signature" que se encuentra en [17].

4. ID-BASED THRESHOLD RING SIGNATURE (ID-TRS)

Como su propio nombre indica, se trata de un esquema de threshold ring signature basado en las identidades de los usuarios que conforman el anillo. A partir de una identidad que cada usuario tiene, se obtiene una llave secreta individual, que será usada posteriormente para firmar un mensaje. Este algoritmo permite que se necesiten para generar la firma grupal tantos usuarios como se desee, desde 1 hasta "n" (en el esquema presentado, se usará un umbral "t").

El proceso de firma grupal consta de cuatro etapas, empezando por la generación de los parámetros y estructuras del algoritmo (ID-TRS. Setup), la obtención de la llave secreta para cada usuario (ID-TRS. Extract), siguiendo con la formación de la firma grupal (ID-TRS. Sign) y terminando con la verificación de la firma de un mensaje recibido (ID-TRS. Verify).



A continuación se desarrolla cada una de las etapas con detalle, prosiguiendo con la explicación de cómo se puede añadir la posibilidad de revocación al esquema y terminando con una tabla clasificatoria de los parámetros y funciones existentes en el algoritmo, en función de por quien pueden ser conocidos.

4.1. Preparación (ID-TRS. Setup)

En esta primera etapa, con unos parámetros fijados previamente por el conjunto de usuarios que forman el anillo, el algoritmo selecciona y computa los parámetros y funciones de hash necesarias para la realización del resto del proceso.

En primer lugar, a partir de un parámetro de entrada λ, que es fijado previamente, el algoritmo selecciona de forma aleatoria dos números primos p' y q' de tamaño en bits igual a λ, y calcula un valor $N=p\cdot q=(2p'+1)\cdot(2q'+1)$. También de forma aleatoria, el algoritmo escoge g_1, g_2, g_3 y g, generadores de $QR(N)$ (conjunto de residuos cuadráticos de N). Seguidamente, genera dos funciones de hash, para igualar todas las identidades, tanto en tamaño como en formato. Una primera función de hash h iguala el tamaño de las identidades, transformándolas a un tamaño de $2\lambda+\theta$ (θ es fijada de forma previa, típicamente su valor es 8). La segunda función de hash, H_{id}, realiza la operación $g^{h(\mathrm{ID})}$ mod N, quedando todas las nuevas identidades dentro de $QR(N)$. Otra función de hash, H_{sig}, es creada para la firma, de tal forma que transforma cualquier cadena de bits introducida a una cadena de k bits. Otros parámetros son introducidos previamente (k, Υ_1, Υ_2, ε, $\Gamma'=[\lfloor 2^{\Upsilon_1}\rfloor-\lfloor 2^{\Upsilon_2}\rfloor+1, \lfloor 2^{\Upsilon_1}\rfloor+\lfloor 2^{\Upsilon_2}\rfloor -1]$ y $\Gamma=[\lfloor 2^{\Upsilon_1}\rfloor-\lfloor 2^{\varepsilon(\Upsilon_2+k)}\rfloor+1, \lfloor 2^{\Upsilon_1}\rfloor+\lfloor 2^{\varepsilon(\Upsilon_2+k)}\rfloor-1]$). Con todo ello, se genera una lista de parámetros, $param=(\lambda, k, \varepsilon, N, H_{id}, H_{sig}, g_1, g_2, g_3, \Gamma', \Gamma)$.
Para asegurar un nivel de seguridad equivalente a un esquema RSA de 1024 bits, se deben seleccionar los siguientes valores para los parámetros: λ=512, k=160, ε=1,1, Υ_1=1080 y Υ_2=800.

A continuación se muestra un resumen esquemático de esta primera etapa:

- Input: λ

- Algoritmo:
  - p', q' aleatorios tales que $|p'|=|q'|=\lambda$
  - $N=p\cdot q=(2p'+1)\cdot(2q'+1)$
  - g_1, g_2, g_3 aleatorios, generadores de *QR(N)*
  - g aleatorio, generador de *QR(N)*

- h: $\{0,1\}^* \rightarrow \{0,1\}^{2\lambda+\theta}$ (típicamente θ=8)
- H_{id}: $ID \rightarrow g^{h(\mathrm{ID})} \bmod N$
- H_{sig}: $\{0,1\}^* \rightarrow Z_{2k}$

- Otros parámetros:
  - k
  - γ_1
  - γ_2
  - ε
  - $\Gamma' = [\lfloor 2^{\gamma_1} \rfloor - \lfloor 2^{\gamma_2} \rfloor + 1, \lfloor 2^{\gamma_1} \rfloor + \lfloor 2^{\gamma_2} \rfloor - 1]$
  - $\Gamma = [\lfloor 2^{\gamma_1} \rfloor - \lfloor 2^{\varepsilon(\gamma_2+k)} \rfloor + 1, \lfloor 2^{\gamma_1} \rfloor + \lfloor 2^{\varepsilon(\gamma_2+k)} \rfloor - 1]$

- Resultados:
  - $param = (\lambda, k, \varepsilon, N, H_{id}, H_{sig}, g_1, g_2, g_3, \Gamma', \Gamma)$

4.2. Extracción (ID-TRS. Extract)

En esta etapa, cada usuario va a recibir, a partir de su identidad, una llave secreta que servirá para que dicho usuario firme un mensaje.

Para una identidad ID_i, correspondiente al usuario i del grupo, el algoritmo computa $y_i = H_{id}(ID_i)$. A continuación, escoge un número primo aleatorio $x_i \in \Gamma'$, y extrae a_i de tal forma que $a_i^{x_i} = y_i \bmod N$. La llave secreta del usuario i es, entonces, $sk_i = (a_i, x_i)$, la cual es retornada al usuario. Además, el algoritmo se guarda $< ID_i, y_i, a_i, x_i >$, por si el usuario i vuelve a pedir la llave secreta, darle la misma.

Este es el resumen de la segunda etapa:

- Input: ID_i (usuario i)

- Algoritmo:
  - $y_i=H_{id}(ID_i)$
  - $x_i \in \Gamma'$, aleatoriamente
  - a_i tal que $a_i^{x_i}=y_i \bmod N$

- Retorno al usuario:

  $sk_i=(a_i, x_i)$

- El algoritmo se guarda $<ID_i, y_i, a_i, x_i>$

4.3. Firma (ID-TRS. Sign)

Esta es la etapa de la creación de la firma grupal para un determinado mensaje, donde van a intervenir t de los n usuarios del grupo.

Como datos de entrada, se encuentran la lista de parámetros *param*, el número total de usuarios del grupo n, el umbral mínimo de usuarios necesarios para la realización de la firma t, el conjunto las identidades de todos los usuarios $Y=\{ID_1, \ldots, ID_n\}$, el conjunto de las llaves secretas de los fimantes $X=\{sk_{\Pi 1}, \ldots, sk_{\Pi t}\}$ y el mensaje m. Con ello, el algoritmo crea un subgrupo $\Pi=\{\Pi_1, \ldots, \Pi_t\}$, que incluye a los usuarios firmantes, y computa $y_i=H_{id}(ID_i)$ para los n usuarios.

Seguidamente, realiza 6 pasos para la creación de la firma:

1. "Auxiliary commitment"

   - Para los usuarios firmantes ($i \in \Pi$):

     Escoge, de forma aleatoria, $u_i \in \pm\{0,1\}^{2\lambda}$, y calcula $w_i := u_i x_i$. En modulo N, calcula $A_{i,1}:=g_1^{u_i}$, $A_{i,2}:=a_i g_2^{u_i}$ y $A_{i,3}:=g_1^{x_i} g_3^{u_i}$.

- Para los usuarios no firmantes ($i \in [1, n] \backslash \Pi$):
  Escoge aleatoriamente $A_{i,1}$, $A_{i,2}$ y $A_{i,1} \in QR(N)$.

2. "Commitment"

- Para los usuarios firmantes ($i \in \Pi$):
  Escoge, de forma aleatoria, $r_{i,x} \in \pm\{0,1\}^{\varepsilon(\gamma_2+k)}$, $r_{i,u} \in \pm\{0,1\}^{\varepsilon(2\lambda+k)}$ y $r_{i,w} \in \pm\{0,1\}^{\varepsilon(\gamma_1+2\lambda+k+1)}$. En módulo N, calcula $T_{i,1} := g_1^{r_{i,u}}$, $T_{i,2} := g_1^{r_{i,x}} g_3^{r_{i,u}}$, $T_{i,3} := A_{i,1}^{r_{i,x}} g_1^{-r_{i,w}}$ y $T_{i,4} := A_{i,2}^{r_{i,x}} g_2^{-r_{i,w}}$.

- Para los usuarios no firmantes ($i \in [1, n] \backslash \Pi$):
  Escoge, de forma aleatoria, $c_i \in Z_{2^k}$, $s_{i,x} \in \pm\{0,1\}^{\varepsilon(\gamma_2+k)}$, $s_{i,u} \in \pm\{0,1\}^{\varepsilon(2\lambda+k)}$, $s_{i,w} \in \pm\{0,1\}^{\varepsilon(\gamma_1+2\lambda+k+1)}$. En módulo N, calcula $T_{i,1} := g_1^{s_{i,u}} A_{i,1}^{c_i}$, $T_{i,2} := g_1^{s_{i,x}-c_i 2^{\gamma_1}} g_3^{s_{i,u}} A_{i,3}^{c_i}$, $T_{i,3} := A_{i,1}^{s_{i,x}-c_i 2^{\gamma_1}} g_1^{-s_{i,w}}$ y $T_{i,4} := A_{i,2}^{s_{i,x}-c_i 2^{\gamma_1}} g_2^{-s_{i,w}} y_i^{c_i}$.

3. "Challenge"

Con todos estos elementos, el algoritmo computa:

$c_0 := H_{sig}(param, n, d, (y_i, A_{i,1}, A_{i,2}, A_{i,3})^n_{i=1}, (T_{i,1}, \cdots, T_{i,4})^n_{i=1}, m)$.

Como se puede ver, se incluyen en el elemento c_0 los parámetros y_i, A_i y T_i de los n usuarios, pero sólo hay información de las llaves secretas en los parámetros de los firmantes, ya que en el caso de los no firmantes dichos parámetros han sido creados a de forma aleatoria, de tal manera que el tamaño y formato tanto de los A_i como de los T_i son iguales para todos, aunque han estado creados de forma distinta.

4. "Response"

El algoritmo genera un polinomio f sobre el cuerpo $GF(2^k)$ de tal forma que $c_0 = f(0)$ y $c_i = f(i)$ para los usuarios no firmantes ($i \in [1, n] \backslash \Pi$). Si la firma ha sido realizada por un mínimo de t usuarios (en caso contrario esa firma no será

válida), ese polinomio tendrá grado máximo (n-t). Para los usuarios firmantes ($i \in \Pi$), se les asigna $c_i = f(i)$, y se computa para ellos $s_{i,u} := r_{i,u} - c_i u_i$, $s_{i,x} := r_{i,x} - c_i(x_i - 2^{\gamma_1})$, $s_{i,w} := r_{i,w} - c_i w_i$.

5. "Signature"

   El algoritmo crea la firma σ':

   $\sigma' := (f, (s_{i,u}, s_{i,x}, s_{i,w})^n_{i=1})$.

6. "Output"

   Finalmente, se retorna σ:

   $\sigma := ((A_{i,1}, A_{i,2}, A_{i,3})^n_{i=1}, \sigma')$.

Así pues, resumiendo esta tercera etapa, se ve que la firma acaba siendo generada con las identidades (y_i) de todos los usuarios, y con unos parámetros (A_i y T_i), también de todos ellos, aparentemente iguales en cuanto a dimensión y formato. La diferencia radica en cómo estos parámetros han sido formados. En el caso de los firmantes, se crean usando sus llaves secretas (a_i, x_i), mientras que, para los no firmantes, son aleatorios. Esto hace que para el verificador o para cualquier usuario sea imposible saber quien ha firmado, pero sí que puedan saber si han sido por lo menos t usuarios, y si realmente son provenientes del grupo en cuestión, como se puede ver en el siguiente apartado, el de la verificación.

4.4. Verificación (ID-TRS. Verify)

Se trata de la etapa en la que el algoritmo comprueba a un verificador que recibe un mensaje firmado si dicha firma es válida, comprobando primeramente que un mínimo de t usuarios hayan firmado el mensaje y, en segundo lugar, comprobando que esos usuarios efectivamente provengan del grupo deseado.

Los parámetros de entrada para la verificación de un mensaje son la lista de parámetros *param*, n, t, el conjunto de las identidades de todos los usuarios $Y=\{ID_1, \ldots, ID_n\}$, el mensaje m y la firma σ. Con ello, el algoritmo computa $y_i=H_{id}(ID_i)$ para los n usuarios.

Hecho esto, sigue un proceso de 4 pasos para la decisión de si la firma es o no es válida:

1. En primer lugar, comprueba si el polinomio f (incluido en σ) tiene grado mínimo (n-t). De no ser así, eso significa que menos de t usuarios han firmado el mensaje, con lo que, aunque esos usuarios pertenezcan al grupo deseado, la firma no es válida.

2. Para todos los usuarios ($i \in [1, n]$), el algoritmo computa $c_i=f(i)$ y, en módulo N, $T_{i,1}:=g_1^{s_{i,u}}A_{i,1}^{c_i}$, $T_{i,2}:=g_1^{s_{i,x}-c_i2^{\gamma_1}}g_3^{s_{i,u}}A_{i,3}^{c_i}$, $T_{i,3}:=A_{i,1}^{s_{i,x}-c_i2^{\gamma_1}}g_1^{-s_{i,w}}$ y $T_{i,4}:=A_{i,2}^{s_{i,x}-c_i2^{\gamma_1}}g_2^{-s_{i,w}}y_i^{c_i}$.

3. Si la firma ha sido creada correctamente, se deben cumplir las siguientes condiciones: $s_{i,x} \in \pm\{0,1\}^{\varepsilon(\gamma_2+k)+1}$, $s_{i,u} \in \pm\{0,1\}^{\varepsilon(2\lambda+k)+1}$, $s_{i,w} \in \pm\{0,1\}^{\varepsilon(\gamma_1+2\lambda+k+1)+1}$ y $f(0)=H_{sig}(param, n, t, (y_i, A_{i,1}, A_{i,2}, A_{i,3})^n_{i=1}, (T_{i,1}, \cdots, T_{i,4})^n_{i=1}, m)$. El algoritmo comprueba una por una y para todos los usuarios todas esas condiciones. Si una sola de ellas no se cumple, la firma no es válida.

4. Finalmente, si todas las condiciones anteriores se cumplen satisfactoriamente, el algoritmo acepta como válida la firma y entrega el mensaje al verificador. En caso contrario, lo rechaza.

Así pues, como se puede ver, en caso de que menos de t usuarios hayan firmado un mensaje, el algoritmo lo rechaza de inmediato (paso 1). Una vez comprobado que se ha alcanzado el umbral mínimo de usuarios firmantes, el algoritmo procede a la comprobación de que la firma haya sido creada por miembros del grupo, mediante las relaciones del paso 3 (que deben cumplirse para $i \in [1, n]$). Si todas ellas cuadran, se puede afirmar que la firma es válida, y aceptar el mensaje recibido.

4.5. Revocación del anonimato

Como se comentó anteriormente, una de las condiciones de funcionamiento de este esquema era la existencia de un ente externo que tuviese la posibilidad de revocar el anonimato de los firmantes, esto es, que pudiese llegar a saber qué usuarios firmaron un mensaje en concreto. Se asume que este ente (llamado autoridad de confianza) no abusa de esta capacidad, sino que solamente actúa en caso de fuerza mayor, como puede ser por exigencia de una orden judicial que obligue a destapar la identidad de los firmantes de un mensaje.

Tal y como se ha explicado en el esquema, no existe nadie (ni interno ni externo al grupo) que, dada una firma, pueda saber qué usuarios son los firmantes, gracias a la aleatoriedad con la que son escogidos los generadores g_1, g_2, g_3 y g.

Si se quiere añadir la posibilidad de revocación del anonimato, el algoritmo se debe construir de forma idéntica a la explicada en los apartados anteriores, con la excepción de que en la etapa ID-TRS. Setup, g_2 no es escogido aleatoriamente, sino de tal forma que $g_2 \equiv g_1^{\,S} \pmod{N}$. Solamente la autoridad de confianza conoce el parámetro s. Con el conocimiento de este parámetro, dicha autoridad puede conocer la parte a_i de cada usuario, realizando la operación $A_{i,2}/A_{i,1}^{s} \pmod{N}$, puesto que $A_{i,1} := g_1^{u_i}$ y $A_{i,2} := a_i g_2^{u_i}$. Solamente los usuarios firmantes tendrán la pareja $A_{i,1}$, $A_{i,2}$ formada correctamente (para el resto de usuarios estos parámetros han sido creados de forma aleatoria, con lo que no mantienen la relación $A_{i,2}/A_{i,1}^{s}$

(mod N)= a_i). Los n valores a_i son pasados por el emisor de llaves, en la base de datos del cual se comprueban cuales de esos valores corresponden a un a_i real, siendo relacionados con una identidad y descubriendo así cuales fueron los usuarios firmantes del mensaje.

4.6. Clasificación de parámetros

En la siguiente tabla se clasifican los distintos parámetros y funciones del algoritmo en según si pueden ser públicos (conocidos tanto por los usuarios del grupo como por usuarios externos que puedan ejercer de verificadores), conocidos solamente por el usuario al cual pertenecen o conocidos por la autoridad de confianza que tiene la potestad de revocar el anonimato de los firmantes:

Parámetro	Público	Sólo usuario *i*	Autoridad de confianza
$param=(\lambda, k, \varepsilon, N, H_{id}, H_{sig}, g_1, g_2, g_3, \Gamma', \Gamma)$	✔		
Y_1, Y_2	✔		
ID_i	✔		
a_i		✔	✔
x_i		✔	
n	✔		

t	✔		
m	✔		
u_i		✔	
$r_{i,x}, r_{i,u}, r_{i,w}$		✔	
c_i		✔	
$T_{i,1}, \cdots, T_{i,5}$		✔	
$\sigma' := (f, (s_{i,u}, s_{i,x}, s_{i,w})_{i=1}^{n})$	✔		
$\sigma := ((A_{i,1}, A_{i,2}, A_{i,3})_{i=1}^{n}, \sigma')$	✔		
s			✔

5. IMPLEMENTACIÓN DEL ALGORITMO EN JAVA

Para implementar el algoritmo, se ha creado una librería que consta de cuatro clases, una por cada etapa del mismo, ID-TRS. Setup, ID-TRS. Extract, ID-TRS. Sign y ID-TRS. Verify. A continuación se desarrolla cada una de las clases, incluyendo algunas pruebas de test para comprobar el correcto funcionamiento de los distintos métodos y funciones presentes en ellas. Estas pruebas han sido realizadas con unos valores sencillos de los parámetros prefijados, con el fin de poder comprobar adecuadamente que las funciones y métodos funcionan bien.

5.1. Clase Setup

En esta clase se crean los parámetros necesarios para su uso en las otras clases, con métodos que serán llamados desde estas.

En primer lugar, esta es la declaración de las variables necesarias para esta clase:

```
import java.math.BigInteger;
import java.util.Random;
import java.util.Vector;
import java.util.Arrays;
import java.security.MessageDigest;
import java.io.IOException;
import java.security.NoSuchAlgorithmException;


public class Setup {
     static int lambda = 2;
     static int theta = 0;
     static int k = 8;
     static Double epsilon = new Double (1.0);
     static BigInteger gamma1 = new BigInteger ("4");
     static BigInteger gamma2 = new BigInteger ("3");
     static BigInteger pp;
     static BigInteger qp;
```

```
static BigInteger p;
static BigInteger q;
static BigInteger N;
static BigInteger g1 = new BigInteger ("3");
static BigInteger g2 = new BigInteger ("4");
static BigInteger g3 = new BigInteger ("5");
static BigInteger g = new BigInteger ("2");
public static BigInteger uno = new BigInteger ("1");
public static BigInteger dos = new BigInteger ("2");
```

Lo primero que se realiza en esta clase es la generación del parámetro *N*, de la siguiente forma:

```
public Setup () {
   Random r = new Random();
   pp = BigInteger.probablePrime (lambda, r);
   qp = BigInteger.probablePrime (lambda, r);
   p = pp.multiply (dos).add (uno);
   q = qp.multiply (dos).add (uno);
   N = p.multiply (q);
}
```

Para el valor λ=3, *p'* y *q'* deben ser números primos de 3 bits, *p'*=(2*p'*+1), *q'*=(2*q'*+1) y $N=p \cdot q$. Esta es la prueba de test del funcionamiento de éste método, en que se puede comprobar, imprimiendo por pantalla los cinco parámetros, que cumplen lo especificado:

```
p'=7
q'=5
p=15
q=11
N=165
```

Dado que muchos de estos parámetros serán utilizados por otras clases, se implementan varios métodos para retornar dichos parámetros, con el fin de que solamente haga falta modificarlos en esta clase:

```
public static int obtener_lambda () {
    return lambda;
}

public static int obtener_k () {
    return k;
}
```

```
public static double obtener_epsilon () {
    return epsilon;
}

public static int obtener_theta () {
    return theta;
}

public static BigInteger obtener_gamma1 () {
    return gamma1;
}

public static BigInteger obtener_gamma2 () {
    return gamma2;
}

public static BigInteger obtener_N () {
    return N;
}

public static BigInteger obtener_g1 () {
    return g1;
}

public static BigInteger obtener_g2 () {
    return g2;
}

public static BigInteger obtener_g3 () {
    return g3;
}

public static BigInteger obtener_g () {
    return g;
}
```

El siguiente parámetro a crear es el conjunto de residuos cuadráticos $QR(N)$. Este grupo consta de todos aquellos números obtenidos al elevar cada entero del conjunto {1, 2, …, N} módulo N. El siguiente método retorna en un "vector" dicho conjunto, asegurando no repetir un mismo elemento más de una vez:

```
public static Vector obtener_QRN () {
    Vector QRN = new Vector<BigInteger> ();
    BigInteger cont = new BigInteger ("0");
    do {
        cont = cont.add (uno);
        if (!QRN.contains (cont.modPow (dos, N))) {
            QRN.add (cont.modPow (dos, N));
       }
    } while (cont.compareTo (N) <= 0);
    return QRN;
}
```

Para comprobar el correcto funcionamiento de este método, se ha forzado N=17, puesto que con un número muy grande $QR(N)$ tiene muchos elementos. De esta forma se ve fácilmente que el método obtiene $QR(N)$. El conjunto $QR(17)$={0, 1, 2, 4, 8, 9, 13, 15, 16}. Este es el resultado de llamar al método anterior e imprimir el "vector" retornado por pantalla:

```
[1, 4, 9, 16, 8, 2, 15, 13, 0]
```

Se comprueba que el método obtiene correctamente el conjunto $QR(N)$. El hecho de que guarde desordenados los elementos del conjunto no es un problema, ya que éste se usa para obtener valores aleatorios.

El siguiente método de la clase ID-TRS. Setup es el encargado de pasar una identidad ID a y, de tal manera que $y=g^{h(ID)} \bmod N$. Se presupone que las identidades introducidas ya cumplen por si mismas la condición de ser valores de $2\lambda+\theta$ bits:

```
public static BigInteger obtener_y (BigInteger ID) {
    BigInteger y;
    y = g.modPow (ID, N);
    return (y);
}
```

Se ha ejecutado el método para un valor de entrada ID=6 y g=2, generador de $QR(N)$ para N=17. Este es el resultado obtenido, imprimiendo por pantalla ID y y:

```
ID=6
y=13
```

El resultado de elevar 2^6 (g^{ID}) es 64, que en módulo 17 es 13. Así pues, el método funciona correctamente.

El último método que tiene esta clase es el encargado de, a partir de un "string" de entrada, correspondiente a $\{param, n, t, (y_i, A_{i,1}, A_{i,2}, A_{i,3})^n_{i=1}, (T_{i,1}, \cdots, T_{i,4})^n_{i=1}, m\}$,

obtener el coeficiente $c_0 := H_{sig}(param, n, d, (y_i, A_{i,1}, A_{i,2}, A_{i,3})^n_{i=1}, (T_{i,1}, \cdots, T_{i,4})^n_{i=1}, m)$. Este método convierte un "string" de cualquier tamaño en bits a un "string" de tamaño máximo k bits, transformando primero a 256 bits mediante una función de hash "SHA-256" y reduciendo o ampliando posteriormente para dejarlo en k o menos bits:

```
public static String Hsig (String lista_string)
              throws IOException, NoSuchAlgorithmException {
    byte [] lista_bytes = lista_string.getBytes ();
    MessageDigest digest = MessageDigest.getInstance ("SHA-256");
    digest.update (lista_bytes);
    byte [] co_256 = digest.digest();
    byte [] co_k = Arrays.copyOf (co_256, k/8);
    String lista_k = Arrays.toString (co_k);
    return lista_k;
}
```

Para probarlo, se ha fijado k=8 bits y se ha llamado al método para un "string" de entrada "[1000, 2000, 3000, 4000]", claramente de un tamaño muy superior a k. Este ha sido el resultado, imprimiendo por pantalla tanto el "string" de entrada como el de salida:

```
String de entrada=[1000, 2000, 3000, 4000]
String de máximo k bits=[125]
```

Se puede comprobar que el "string" de salida tiene menos de k=8 bits (en este caso, 7 bits).

Esta es la clase "Setup" entera:

```
import java.math.BigInteger;
import java.util.Random;
import java.util.Vector;
import java.util.Arrays;
import java.security.MessageDigest;
import java.io.IOException;
import java.security.NoSuchAlgorithmException;
```

```
public class Setup {
     static int lambda = 2;
     static int theta = 0;
     static int k = 8;
     static Double epsilon = new Double (1.0);
     static BigInteger gamma1 = new BigInteger ("4");
     static BigInteger gamma2 = new BigInteger ("3");
     static BigInteger pp;
     static BigInteger qp;
     static BigInteger p;
     static BigInteger q;
     static BigInteger N;
     static BigInteger g1 = new BigInteger ("3");
     static BigInteger g2 = new BigInteger ("4");
     static BigInteger g3 = new BigInteger ("5");
     static BigInteger g = new BigInteger ("2");
     public static BigInteger uno = new BigInteger ("1");
     public static BigInteger dos = new BigInteger ("2");

     public Setup () {
        Random r = new Random();
        pp = BigInteger.probablePrime (lambda, r);
        qp = BigInteger.probablePrime (lambda, r);
        p = pp.multiply (dos).add (uno);
        q = qp.multiply (dos).add (uno);
        //N = p.multiply (q);
        N = new BigInteger ("17");
     }

     public static int obtener_lambda () {
         return lambda;
     }

     public static int obtener_k () {
         return k;
     }

     public static Double obtener_epsilon () {
         return epsilon;
     }

     public static int obtener_theta () {
         return theta;
     }

     public static BigInteger obtener_gamma1 () {
         return gamma1;
     }

     public static BigInteger obtener_gamma2 () {
         return gamma2;
     }

     public static BigInteger obtener_N () {
         return N;
     }
```

```
public static BigInteger obtener_g1 () {
    return g1;
}

public static BigInteger obtener_g2 () {
    return g2;
}

public static BigInteger obtener_g3 () {
    return g3;
}

public static BigInteger obtener_g () {
    return g;
}

public static Vector obtener_QRN () {
    Vector QRN = new Vector<BigInteger> ();
    BigInteger cont = new BigInteger ("0");
    do {
        cont = cont.add (uno);
        if (!QRN.contains (cont.modPow (dos, N))) {
            QRN.add (cont.modPow (dos, N));
       }
    } while (cont.compareTo (N) <= 0);
    return QRN;
}

public static BigInteger obtener_y (BigInteger ID) {
    BigInteger y;
    y = g.modPow (ID, N);
    return (y);
}


public static String Hsig (String lista_string)
             throws IOException, NoSuchAlgorithmException {
    byte [] lista_bytes = lista_string.getBytes ();
    MessageDigest digest = MessageDigest.getInstance ("SHA-256");
    digest.update (lista_bytes);
    byte [] co_256 = digest.digest();
    byte [] co_k = Arrays.copyOf (co_256, k/8);
    String lista_k = Arrays.toString (co_k);
    return lista_k;
}
```

5.2. Clase Extract

En esta clase se obtiene la llave secreta para un usuario, es decir, los parámetros x_i y a_i.

En primer lugar, la declaración de variables general para la clase:

```
import java.math.BigInteger;
import java.util.Random;


public class Extract {
    public static BigInteger uno = new BigInteger ("1");
    public static BigInteger dos = new BigInteger ("2");
    int lambda = Setup.obtener_lambda ();
    int theta = Setup.obtener_theta ();
    public static BigInteger gamma1 = Setup.obtener_gamma1 ();
    public static BigInteger gamma2 = Setup.obtener_gamma2 ();
    public static BigInteger N = Setup.obtener_N ();
```

Seguidamente, la clase presenta un método para obtener el valor de x_i para el usuario, que debe ser un número primo aleatorio perteneciente al conjunto $\Gamma'=[\lfloor 2^{\gamma_1}\rfloor-\lfloor 2^{\gamma_2}\rfloor+1, \lfloor 2^{\gamma_1}\rfloor+\lfloor 2^{\gamma_2}\rfloor -1]$ en módulo N:

```
public static BigInteger obtener_x () {
    BigInteger gamma1 = Setup.obtener_gamma1 ();
    BigInteger gamma2 = Setup.obtener_gamma2 ();
    BigInteger N = Setup.obtener_N ();
    BigInteger margen = dos.modPow (gamma2, N).multiply (dos).subtract (uno);
    Random rnd = new Random ();
    int Lmargen = margen.bitLength ();
    BigInteger random;
    BigInteger x;
    do {
        do {
            random = new BigInteger (Lmargen, rnd);
        } while (random.compareTo (margen) > 0);
        BigInteger valmin = dos.modPow (gamma1, N).subtract (dos.modPow (gamma2, N)).add (uno);
        x = random.add (valmin);
    } while (!x.isProbablePrime (10000));
    return x;
}
```

Para comprobar el funcionamiento de este método, se ha escogido γ_1=5 y γ_2=4, por lo que $\Gamma'=[\lfloor 2^{\gamma_1}\rfloor-\lfloor 2^{\gamma_2}\rfloor+1, \lfloor 2^{\gamma_1}\rfloor+\lfloor 2^{\gamma_2}\rfloor -1]=[17, 47]$, y se ha impreso x por pantalla:

```
x=23
```

Efectivamente, 23 es un número primo comprendido entre 17 y 47.

En el siguiente método, se obtiene la otra parte de la llave secreta del usuario, a_i, que debe cumplir $a_i^{x_i}=y_i \bmod N$:

```
public static BigInteger obtener_a (BigInteger x, BigInteger y) {
    BigInteger a = new BigInteger ("0");
    BigInteger b = new BigInteger ("0");
    BigInteger N = Setup.obtener_N ();
    do {
        a = a.add (uno);
        b = a.modPow (x, N);
    } while (b.compareTo (y) != 0);
    return (a);
}
```

Para comprobar su funcionamiento, se ha invocado al método para un valor de x=3 y de y=6, con N=17. Este ha sido el valor de a obtenido:

```
a=5
```

Para los valores indicados anteriormente, $a^x=5^3=125$. En módulo 17, en efecto, 125 es 6, el valor esperado de y.

Así pues, la clase “Extract” queda de la siguiente forma:

```
import java.math.BigInteger;
import java.util.Random;


public class Extract {
    public static BigInteger uno = new BigInteger ("1");
    public static BigInteger dos = new BigInteger ("2");
    int lambda = Setup.obtener_lambda ();
    int theta = Setup.obtener_theta ();
    public static BigInteger gamma1 = Setup.obtener_gamma1 ();
    public static BigInteger gamma2 = Setup.obtener_gamma2 ();
    public static BigInteger N = Setup.obtener_N ();
```

```
public static BigInteger obtener_x () {
    BigInteger gamma1 = Setup.obtener_gamma1 ();
    BigInteger gamma2 = Setup.obtener_gamma2 ();
    BigInteger N = Setup.obtener_N ();
    BigInteger margen = dos.modPow (gamma2, N).multiply (dos).subtract (uno);
    Random rnd = new Random ();
    int Lmargen = margen.bitLength ();
    BigInteger random;
    BigInteger x;
    do {
        do {
            random = new BigInteger (Lmargen, rnd);
        } while (random.compareTo (margen) > 0);
        BigInteger valmin = dos.modPow (gamma1, N).subtract (dos.modPow (gamma2, N)).add (uno);
        x = random.add (valmin);
    } while (!x.isProbablePrime (10000));
    return x;
}

public static BigInteger obtener_a (BigInteger x, BigInteger y) {
    BigInteger a = new BigInteger ("0");
    BigInteger b = new BigInteger ("0");
    BigInteger N = Setup.obtener_N ();
    do {
        a = a.add (uno);
        b = a.modPow (x, N);
    } while (b.compareTo (y) != 0);
    return (a);
}
```

5.3. Clase Sign

En esta clase se generan todos los parámetros de todos los usuarios y generales para la realización de la firma, para, posteriormente proceder con ella. Todas las pruebas de test para esta clase se han realizado con los siguientes valores: λ=2, θ=0, k=1, ε=1, Y_1=4, Y_2=3, N=17, g=2, g_1=3, g_2=4 y g_3=5.

Esta es la declaración de variables de la clase:

```
import java.math.BigInteger;
import java.util.Random;
import java.util.Vector;
import java.util.List;
import java.io.IOException;
import java.security.NoSuchAlgorithmException;
```

```
public class Sign {
    int n = 4;
    int t = 2;
    int [] lista_firmantes = {1, 3};
    int lambda = Setup.obtener_lambda ();
    int theta = Setup.obtener_theta ();
    int k = Setup.obtener_k ();
    int epsilon = Setup.obtener_epsilon ().intValue ();
    int gamma1 = Setup.obtener_gamma1 ().intValue ();
    int gamma2 = Setup.obtener_gamma2 ().intValue ();
    BigInteger N = Setup.obtener_N ();
    BigInteger g1 = Setup.obtener_g1 ();
    BigInteger g2 = Setup.obtener_g2 ();
    BigInteger g3 = Setup.obtener_g3 ();
    BigInteger g = Setup.obtener_g ();
    BigInteger zero = new BigInteger ("0");
    BigInteger uno = new BigInteger ("1");
    BigInteger dos = new BigInteger ("2");
    Random rnd = new Random ();
    BigInteger lista_ID [] = new BigInteger [n];
    BigInteger lista_y_A [] = new BigInteger [n*4];
    BigInteger [] lista_x = new BigInteger [n];
    BigInteger [] lista_a = new BigInteger [n];
    BigInteger lista_T [] = new BigInteger [n*4];
    BigInteger lista_c [] = new BigInteger [n];
    BigInteger m;
    String c0 = "";
```

Como se puede ver, es aquí donde se introducen el número total de usuarios n, el umbral mínimo necesario para que la firma sea válida t y la lista de los t firmantes concretos. En este caso, se ha usado un grupo de n=4 usuarios, con un umbral t=2. Los dos firmantes son el usuario 1 y el 3.

En primer lugar, se crea una lista aleatoria de identidades ID_i (estas identidades pueden ser añadidas manualmente si se desea, anulando esta parte del código y añadiéndolas al "array" dispuesto para ello). Estas identidades son generadas cumpliendo la restricción de que sean de tamaño $2\lambda+\theta$ bits y almacenadas en un "array":

```
BigInteger ID;
int LID = 2 * lambda + theta;
for (int i=1; i<=n; i++) {
    do {
        ID = new BigInteger (LID + 1, rnd);
    } while (ID.bitLength() != LID);
    lista_ID [i-1] = ID;
}
```

Para comprobar esta parte, se han impreso por pantalla las 4 identidades *ID* obtenidas:

```
ID1=11
ID2=12
ID3=10
ID4=14
```

Efectivamente, las cuatro identidades son de $2\lambda+\theta=5$ bits.

En segundo lugar, se crea una lista de n parámetros y_i, correspondientes a las identidades anteriores. Se almacenan en un "array" que compartirán con los valores A_i, según el orden $(y_i, A_{i,1}, A_{i,2}, A_{i,3})^n_{i=1}$:

```
for (int i=1; i<=n; i++) {
    lista_y_A [i*4-4] = Setup.obtener_y (lista_ID [i-1]);
}
```

Estos son los valores obtenidos:

```
y1=8
y2=16
y3=4
y4=13
```

Los valores y_i obtenidos para cada identidad ID_i son, efectivamente, el resultado de la operación $g^{ID_i} \pmod N$.

En segundo lugar, se genera un "array" de t parámetros x_i, correspondiente a los firmantes:

```
for (int i=1; i<=n; i++) {
    for (int j=0; j<t; j++) {
        if (i == lista_firmantes [j]) {
            lista_x [i-1] = Extract.obtener_x ();
        }
    }
}
```

Esta es la impresión por pantalla de las x_i obtenidas:

```
x1=13
x3=11
```

Como se puede ver, solamente se obtienen x_i para los firmantes, y son números primos pertenecientes al conjunto Γ'=[$\lfloor 2^{\gamma_1} \rfloor$-$\lfloor 2^{\gamma_2} \rfloor$+1, $\lfloor 2^{\gamma_1} \rfloor$+$\lfloor 2^{\gamma_2} \rfloor$ -1]=[17, 47], como corresponde.

Del mismo modo, se genera un "array" para los parámetros a_i de los firmantes:

```
for (int i=1; i<=n; i++) {
    for (int j=0; j<t; j++) {
        if (i == lista_firmantes [j]) {
            lista_a [i-1] = Extract.obtener_a (lista_x [i-1], lista_y_A [i*4-4]);
        }
    }
}
```

La impresión por pantalla de estos parámetros es la siguiente:

```
a1=9
a3=13
```

Nuevamente, se ve que se obtienen a_i sólo para los firmantes. Además, recuperando los valores obtenidos en impresiones anteriores, ambos cumplen $a_i^{x_i}=y_i \bmod N$ ($9^{13} \pmod{17}=8$ y $13^{11} \pmod{17}=4$), por lo que se puede afirmar que su funcionamiento es correcto.

A continuación, se procede con la creación de los parámetros A_i y T_i para los usuarios firmantes, así como de los parámetros necesarios para ello. Como se ha comentado, los parámetros A_i son almacenados en un "array" junto con los y_i, de la forma $(y_i, A_{i,1}, A_{i,2}, A_{i,3})^n_{i=1}$. Los parámetros T_i, a su vez, se almacenan en otro "array", según $(T_{i,1}, \cdots, T_{i,4})^n_{i=1}$:

```
for (int i=1; i<=n; i++) {
    for (int j=0; j<t; j++) {
        if (i == lista_firmantes [j]) {
            BigInteger u;
            BigInteger rx;
            BigInteger ru;
            BigInteger rw;
            do {
                u = new BigInteger (2*lambda + 1, rnd);
            } while (u.bitLength() != 2*lambda);
            BigInteger w = u.multiply (lista_x [i-1]);
            lista_y_A [i*4-3] = g1.modPow (u, N);
            lista_y_A [i*4-2] = (g2.modPow (u, N).multiply (lista_a [i-1])).mod (N);
            lista_y_A [i*4-1] = (g1.modPow (lista_x [i-1], N).multiply (g3.modPow (u, N))).mod (N);
            do {
                rx = new BigInteger (epsilon*(gamma2 + k) + 1, rnd);
            } while (rx.bitLength() != epsilon*(gamma2 + k));
            do {
                ru = new BigInteger (epsilon*(2*lambda + k) + 1, rnd);
            } while (ru.bitLength() != epsilon*(2*lambda + k));
            do {
                rw = new BigInteger (epsilon*(gamma1 + 2*lambda + k + 1) + 1, rnd);
            } while (rw.bitLength() != epsilon*(gamma1 + 2*lambda + k + 1));
            lista_T [i*4-4] = g.modPow (ru, N);
            lista_T [i*4-3] = (g1.modPow (rx, N).multiply (g3.modPow (ru, N))).mod (N);
            lista_T [i*4-2] = (lista_y_A [i*4-3].modPow (rx, N).multiply (g1.modPow (zero.subtract(rw), N))).mod (N);
            lista_T [i*4-1] = (lista_y_A [i*4-2].modPow (rx, N).multiply (g2.modPow (zero.subtract(rw), N))).mod (N);
        }
    }
}
```

Para la comprobación del correcto funcionamiento de esta parte, se han impreso por pantalla no solamente los parámetros A_i y T_i, sino también los que se necesitan para generarlos:

```
u1=8     u3=8
w1=104   w3=88
A11=16   A31=16
A12=9    A32=13
A13=5    A33=10
rx1=12   rx3=11
ru1=27   ru3=16
rw1=883  rw3=892
T11=8    T31=1
T12=10   T32=7
T13=12   T33=4
T13=13   T33=4
```

En primer lugar, se puede ver que, efectivamente, se han creado dichos parámetros para los usuarios 1 y 3, que son los firmantes en este caso. En segundo lugar, los parámetros u_1 y u_3 cumplen la especificación requerida por el algoritmo de ser valores de 2λ=4 bits. Lo mismo ocurre con $r_{1,x}$ y $r_{3,x}$ ($\varepsilon(\gamma_2+k)$ bits), $r_{1,u}$ y $r_{3,u}$ ($\varepsilon(2\lambda+k)$ bits) y $r_{1,w}$ y $r_{3,w}$ ($\varepsilon(\gamma_1+2\lambda+k+1)$ bits). Además, tanto los parámetros A_i como los

T_i cumplen con lo especificado en el algoritmo, es decir, $A_{i,1}$:=$g_1^{u_i}$, $A_{i,2}$:=$a_i g_2^{u_i}$, $A_{i,3}$:=$g_1^{x_i} g_3^{u_i}$, $T_{i,1}$:=$g_1^{r_{i,u}}$, $T_{i,2}$:=$g_1^{r_{i,x}} g_3^{r_{i,u}}$, $T_{i,3}$:=$A_{i,1}^{r_{i,x}} g_1^{-r_{i,w}}$ y $T_{i,4}$:=$A_{i,2}^{r_{i,x}} g_2^{-r_{i,w}}$, en módulo N.

Seguidamente, se generan los mismos parámetros para los usuarios no firmantes:

```
for (int i=1; i<=n; i++) {
    if (lista_y_A [i*4-3] == null) {
        Vector QRN = Setup.obtener_QRN ();
        int num_elem = QRN.size();
        BigInteger indice1;
        BigInteger indice2;
        BigInteger indice3;
        BigInteger numElem = new BigInteger(String.valueOf(num_elem));
        int LnumElem = numElem.subtract (uno).bitLength ();
        do {
            indice1 = new BigInteger (LnumElem, rnd);
        } while (indice1.compareTo (numElem.subtract (uno)) > 0);
        int ind = indice1.intValue ();
        Object elem = QRN.get (ind);
        lista_y_A [i*4-3] = new BigInteger(String.valueOf(elem));
        do {
            do {
                indice2 = new BigInteger (LnumElem, rnd);
            } while (indice2.compareTo (numElem.subtract (uno)) > 0);
        } while (indice2.compareTo (indice1) == 0);
        ind = indice2.intValue ();
        elem = QRN.get (ind);
        lista_y_A [i*4-2] = new BigInteger(String.valueOf(elem));
        do {
            do {
                do {
                    indice3 = new BigInteger (LnumElem, rnd);
                } while (indice3.compareTo (numElem.subtract (uno)) > 0);
            } while (indice3.compareTo (indice1) == 0);
        } while (indice3.compareTo (indice2) == 0);
        ind = indice3.intValue ();
        elem = QRN.get (ind);

        lista_y_A [i*4-1] = new BigInteger(String.valueOf(elem));
        BigInteger su;
        BigInteger sx;
        BigInteger sw;
        BigInteger gamma_1 = new BigInteger(String.valueOf(gamma1));
        lista_c [i-1] = new BigInteger (k + 1, rnd);
        do {
            su = new BigInteger (epsilon*(2*lambda + k) + 1, rnd);
        } while (su.bitLength() != epsilon*(2*lambda + k));
        do {
            sx = new BigInteger (epsilon*(gamma2 + k) +1, rnd);
        } while (sx.bitLength() != epsilon*(gamma2 + k));
        do {
            sw = new BigInteger (epsilon*(gamma1 + 2*lambda + k + 1) + 1, rnd);
        } while (sw.bitLength() != epsilon*(gamma1 + 2*lambda + k + 1));
        lista_T [i*4-4] = (g1.modPow (su, N).multiply (lista_y_A [i*4-3].modPow (lista_c [i-1], N))).mod (N);
        lista_T [i*4-3] = (g1.modPow (sx.subtract (lista_c [i-1].multiply (dos.modPow (gamma_1, N))), N).
                           multiply (g3.modPow (su, N)).multiply (lista_y_A [i*4-1].modPow (lista_c [i-1], N))).
                           mod (N);
        lista_T [i*4-2] = (lista_y_A [i*4-3].modPow (sx.subtract (lista_c [i-1].multiply (dos.modPow (gamma_1, N))), N).
                           multiply (g1.modPow (zero.subtract (sw), N))).mod (N);
        lista_T [i*4-1] = (lista_y_A [i*4-2].modPow (sx.subtract (lista_c [i-1].multiply (dos.modPow (gamma_1, N))), N).
                           multiply (g2.modPow (zero.subtract (sw), N)).multiply (lista_y_A [i*4-4].
                           modPow (lista_c [i-1], N))).mod (N);
    }
}
```

De igual forma que con los firmantes, se han impreso por pantalla todos los parámetros:

```
A21=1        A41=8
A22=0        A42=2
A23=15       A43=13
c2=0         c4=1
su2=16       su4=31
sx2=14       sx4=11
sw2=650      sw4=740
T21=1        T41=14
T22=2        T42=8
T23=15       T43=8
T23=0        T43=2
```

Para empezar, se ve que los parámetros obtenidos corresponden a los usuarios no firmantes (el 2 y el 4, en este caso). Los parámetros $A_{i,1}$, $A_{i,2}$ y $A_{i,3}$ son elementos pertenecientes a $QR(N)$ ($QR(17)$ en este caso), tal y como indica el algoritmo. c_2 y c_4 tienen la dimensión adecuada (k=1 bits). Lo mismo ocurre con $s_{2,u}$ y $s_{4,u}$ ($\varepsilon(2\lambda+k)$ bits), $s_{2,x}$ y $s_{4,x}$ ($\varepsilon(\gamma_2+k)$ bits) y $s_{2,w}$ y $s_{4,w}$ ($\varepsilon(\gamma_1+2\lambda+k+1)$ bits). Finalmente, se puede comprobar que $T_{i,1}:=g_1^{s_{i,u}}A_{i,1}^{c_i}$, $T_{i,2}:=g_1^{s_{i,x}-c_i2^{\gamma_1}}g_3^{s_{i,u}}A_{i,3}^{c_i}$, $T_{i,3}:=A_{i,1}^{s_{i,x}-c_i2^{\gamma_1}}g_1^{-s_{i,w}}$ y $T_{i,4}:=A_{i,2}^{s_{i,x}-c_i2^{\gamma_1}}g_2^{-s_{i,w}}y_i^{c_i}$, todos ellos en módulo N.

A continuación, se crea un "vector" que va a comprender todos los parámetros que intervienen en la formación del coeficiente c_0, es decir, $\{param, n, d, (y_i, A_{i,1}, A_{i,2}, A_{i,3})^n_{i=1}, (T_{i,1}, \cdots, T_{i,4})^n_{i=1}, m\}$, y se convierte a "string". En este caso, el mensaje se genera de forma aleatoria, con un tamaño de entre 0 y 10 bits (esto puede ser modificado o incluso introducir el mensaje manualmente):

```
m = new BigInteger (10, rnd);
Vector lista_vector = new Vector ();
lista_vector.add (lambda);
lista_vector.add (k);
lista_vector.add (epsilon);
lista_vector.add (N);
lista_vector.add (g1);
lista_vector.add (g2);
lista_vector.add (g3);
lista_vector.add (n);
lista_vector.add (t);
for (int i=0; i<n*4; i++) {
    lista_vector.add (lista_y_A [i]);
}
for (int i=0; i<n*4; i++) {
    lista_vector.add (lista_T [i]);
}
lista_vector.add (m);
String lista_string = lista_vector.toString ();
```

Como se puede ver, este "vector" debe tener (para el caso de n=4) 42 componentes. Se imprime por pantalla el "vector" para comprobarlo:

```
[2, 8, 1, 17, 3, 4, 5, 4, 2, 16, 4, 16, 10, 9, 13, 16, 1, 4, 16, 13, 6, 9,
16, 8, 13, 8, 3, 14, 4, 12, 3, 7, 16, 4, 11, 9, 4, 12, 14, 6, 13, 218]
```

Efectivamente, tiene las 42 componentes esperadas.

Pasando el "string" por referencia, se llama al método "Setup.Hsig", para obtener el valor de c_0:

```
try {
    c0 = new String(Setup.Hsig (lista_string));
} catch (IOException ex) {
    ex.printStackTrace();
} catch (NoSuchAlgorithmException ex) {
    ex.printStackTrace();
}
```

Como se ha visto en la etapa "ID-TRS. Setup", este método retorna un "string" de máximo k bits, previo paso por la función de hash "SHA-256", que obtiene una cadena de 256 bits. Este es el resultado obtenido para el "string" anterior después de pasar por el método:

```
[111]
```

Se ve que se ha transformado el "string" anterior a un valor igual o menor a k=8 bits (7, en este caso).

Lo que viene a continuación no se ha podido implementar por falta de tiempo, pero se explica como debería hacerse para completar la clase.

El siguiente paso sería la creación de los coeficientes del polinomio $f(x)=k_0+k_1x+k_2x^2+\cdots+k_{n-t}x^{n-t}$. Como se vio en la explicación del algoritmo, dicho polinomio debe cumplir $c_0=f(0)$ y $c_i=f(i)$ para los usuarios no firmantes ($i \in [1,$

$n]\backslash\Pi$). Dichos parámetros se encuentran en el "array" "lista_c". Debería implementarse un método que resolviera el siguiente sistema de ecuaciones lineales, para el caso n=4, t=2, con los usuarios firmantes {1, 3} y los no firmantes {2, 4}:

$$\begin{cases} f(0) = k_0 = c_0 \\ f(2) = k_0 + 2k_1 + 4k_2 = c_2 \\ f(4) = k_0 + 4k_1 + 16k_2 = c_4 \end{cases}$$

Esto se podría resolver con un método que calculase, mediante Gauss, los coeficientes k_i, a partir de las matrices siguientes:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 2 & 4 \\ 1 & 4 & 16 \end{pmatrix} \begin{pmatrix} k_0 \\ k_1 \\ k_2 \end{pmatrix} = \begin{pmatrix} c_0 \\ c_2 \\ c_4 \end{pmatrix}$$

Como se puede ver, es extrapolable a cualquier valor de n y t, y para cualquier conjunto de firmantes. Estos valores de k_i serían almacenados en un "vector".

A continuación, se asignaría a los firmantes un valor c_i, evaluando el polinomio según c_i=$f(i)$. Con ello, se computarían los parámetros $s_{i,u} := r_{i,u} - c_i u_i$, $s_{i,x} := r_{i,x} - c_i(x_i - 2^{\gamma_1})$ y $s_{i,w} := r_{i,w} - c_i w_i$ para los firmantes.

Seguidamente, se crearía un "vector" que contuviese $\sigma' := (f, (s_{i,u}, s_{i,x}, s_{i,w})^n_{i=1})$. Finalmente, un método retornaría otro "vector" que contendría $\sigma := ((A_{i,1}, A_{i,2}, A_{i,3})^n_{i=1}, \sigma')$, llegando así al final de la clase.

Así es como queda la clase Sign entera, a falta de implementar lo que se acaba de comentar:

```
import java.math.BigInteger;
import java.util.Random;
import java.util.Vector;
import java.util.List;
import java.io.IOException;
import java.security.NoSuchAlgorithmException;
```

```
public class Sign {
    int n = 4;
    int t = 2;
    int [] lista_firmantes = {1, 3};
    int lambda = Setup.obtener_lambda ();
    int theta = Setup.obtener_theta ();
    int k = Setup.obtener_k ();
    int epsilon = Setup.obtener_epsilon ().intValue ();
    int gamma1 = Setup.obtener_gamma1 ().intValue ();
    int gamma2 = Setup.obtener_gamma2 ().intValue ();
    BigInteger N = Setup.obtener_N ();
    BigInteger g1 = Setup.obtener_g1 ();
    BigInteger g2 = Setup.obtener_g2 ();
    BigInteger g3 = Setup.obtener_g3 ();
    BigInteger g = Setup.obtener_g ();
    BigInteger zero = new BigInteger ("0");
    BigInteger uno = new BigInteger ("1");
    BigInteger dos = new BigInteger ("2");
    Random rnd = new Random ();
    BigInteger lista_ID [] = new BigInteger [n];
    BigInteger lista_y_A [] = new BigInteger [n*4];
    BigInteger [] lista_x = new BigInteger [n];
    BigInteger [] lista_a = new BigInteger [n];
    BigInteger lista_T [] = new BigInteger [n*4];
    BigInteger lista_c [] = new BigInteger [n];
    BigInteger m;
    String c0 = "";

    public Sign () {

        // Creación lista n identidades "ID":
        BigInteger ID;
        int LID = 2 * lambda + theta;
        for (int i=1; i<=n; i++) {
            do {
                ID = new BigInteger (LID + 1, rnd);
            } while (ID.bitLength() != LID);
            lista_ID [i-1] = ID;
        }

        // Creación lista n identidades "y":
        for (int i=1; i<=n; i++) {
            lista_y_A [i*4-4] = Setup.obtener_y (lista_ID [i-1]);
        }

        // Creación lista t parámetros "x" (firmantes):
        for (int i=1; i<=n; i++) {
            for (int j=0; j<t; j++) {
                if (i == lista_firmantes [j]) {
                    lista_x [i-1] = Extract.obtener_x ();
                }
            }
        }

        // Creación lista t parámetros "a" (firmantes):
        for (int i=1; i<=n; i++) {
            for (int j=0; j<t; j++) {
                if (i == lista_firmantes [j]) {
                    lista_a [i-1] = Extract.obtener_a (lista_x [i-1], lista_y_A [i*4-4]);
                }
            }
        }

     // Firmantes:
     for (int i=1; i<=n; i++) {
         for (int j=0; j<t; j++) {
             if (i == lista_firmantes [j]) {
                 BigInteger u;
                 BigInteger rx;
                 BigInteger ru;
                 BigInteger rw;
                 do {
                     u = new BigInteger (2*lambda + 1, rnd);
                 } while (u.bitLength() != 2*lambda);
                 BigInteger w = u.multiply (lista_x [i-1]);
                 lista_y_A [i*4-3] = g1.modPow (u, N);
                 lista_y_A [i*4-2] = (g2.modPow (u, N).multiply (lista_a [i-1])).mod (N);
                 lista_y_A [i*4-1] = (g1.modPow (lista_x [i-1], N).multiply (g3.modPow (u, N))).mod (N);
                 do {
                     rx = new BigInteger (epsilon*(gamma2 + k) + 1, rnd);
                 } while (rx.bitLength() != epsilon*(gamma2 + k));
                 do {
                     ru = new BigInteger (epsilon*(2*lambda + k) + 1, rnd);
                 } while (ru.bitLength() != epsilon*(2*lambda + k));
                 do {
                     rw = new BigInteger (epsilon*(gamma1 + 2*lambda + k + 1) + 1, rnd);
                 } while (rw.bitLength() != epsilon*(gamma1 + 2*lambda + k + 1));
                 lista_T [i*4-4] = g.modPow (ru, N);
                 lista_T [i*4-3] = (g1.modPow (rx, N).multiply (g3.modPow (ru, N))).mod (N);
                 lista_T [i*4-2] = (lista_y_A [i*4-3].modPow (rx, N).multiply (g1.modPow (zero.subtract(rw), N))).mod (N);
                 lista_T [i*4-1] = (lista_y_A [i*4-2].modPow (rx, N).multiply (g2.modPow (zero.subtract(rw), N))).mod (N);
             }
         }
     }
```

```
// No firmantes:
for (int i=1; i<=n; i++) {
    if (lista_y_A [i*4-3] == null) {
        Vector QRN = Setup.obtener_QRN ();
        int num_elem = QRN.size();
        BigInteger indice1;
        BigInteger indice2;
        BigInteger indice3;
        BigInteger numElem = new BigInteger(String.valueOf(num_elem));
        int LnumElem = numElem.subtract (uno).bitLength ();
        do {
            indice1 = new BigInteger (LnumElem, rnd);
        } while (indice1.compareTo (numElem.subtract (uno)) > 0);
        int ind = indice1.intValue ();
        Object elem = QRN.get (ind);
        lista_y_A [i*4-3] = new BigInteger(String.valueOf(elem));
        do {
            do {
                indice2 = new BigInteger (LnumElem, rnd);
            } while (indice2.compareTo (numElem.subtract (uno)) > 0);
        } while (indice2.compareTo (indice1) == 0);
        ind = indice2.intValue ();
        elem = QRN.get (ind);
        lista_y_A [i*4-2] = new BigInteger(String.valueOf(elem));
        do {
            do {
                do {
                    indice3 = new BigInteger (LnumElem, rnd);
                } while (indice3.compareTo (numElem.subtract (uno)) > 0);
            } while (indice3.compareTo (indice1) == 0);
        } while (indice3.compareTo (indice2) == 0);
        ind = indice3.intValue ();
        elem = QRN.get (ind);
        lista_y_A [i*4-1] = new BigInteger(String.valueOf(elem));
        BigInteger su;
        BigInteger sx;
        BigInteger sw;
        BigInteger gamma_1 = new BigInteger(String.valueOf(gamma1));
        lista_c [i-1] = new BigInteger (k + 1, rnd);
        do {
            su = new BigInteger (epsilon*(2*lambda + k) + 1, rnd);
        } while (su.bitLength() != epsilon*(2*lambda + k));
        do {
            sx = new BigInteger (epsilon*(gamma2 + k) +1, rnd);
        } while (sx.bitLength() != epsilon*(gamma2 + k));
        do {
            sw = new BigInteger (epsilon*(gamma1 + 2*lambda + k + 1) + 1, rnd);
        } while (sw.bitLength() != epsilon*(gamma1 + 2*lambda + k + 1));
        lista_T [i*4-4] = (g1.modPow (su, N).multiply (lista_y_A [i*4-3].modPow (lista_c [i-1], N))).mod (N);
        lista_T [i*4-3] = (g1.modPow (sx.subtract (lista_c [i-1].multiply (dos.modPow (gamma_1, N))), N).
                           multiply (g3.modPow (su, N)).multiply (lista_y_A [i*4-1].modPow (lista_c [i-1], N))).
                           mod (N);
        lista_T [i*4-2] = (lista_y_A [i*4-3].modPow (sx.subtract (lista_c [i-1].multiply (dos.modPow (gamma_1, N))), N).
                           multiply (g1.modPow (zero.subtract (sw), N))).mod (N);
        lista_T [i*4-1] = (lista_y_A [i*4-2].modPow (sx.subtract (lista_c [i-1].multiply (dos.modPow (gamma_1, N))), N).
                           multiply (g2.modPow (zero.subtract (sw), N)).multiply (lista_y_A [i*4-4].
                           modPow (lista_c [i-1], N))).mod (N);
    }
}

// Creación lista parámetros para crear c0:
m = new BigInteger (10, rnd);
Vector lista_vector = new Vector ();
lista_vector.add (lambda);
lista_vector.add (k);
lista_vector.add (epsilon);
lista_vector.add (N);
lista_vector.add (g1);
lista_vector.add (g2);
lista_vector.add (g3);
lista_vector.add (n);
lista_vector.add (t);
for (int i=0; i<n*4; i++) {
    lista_vector.add (lista_y_A [i]);
}
for (int i=0; i<n*4; i++) {
    lista_vector.add (lista_T [i]);
}
lista_vector.add (m);
String lista_string = lista_vector.toString ();

// Creación parámetro c0:
try {
    c0 = new String (Setup.Hsig (lista_string));
} catch (IOException ex) {
    ex.printStackTrace ();
} catch (NoSuchAlgorithmException ex) {
    ex.printStackTrace ();
}
```

5.4. Clase Verify

En esta clase se comprueba si una firma recibida es válida. Por falta de tiempo no se ha podido implementar esta clase, pero a continuación se explica cómo debería haberse hecho.

En primer lugar, se debería llamar al método de la clase "Sign" que retorna $\sigma:=((A_{i,1}, A_{i,2}, A_{i,3})^{n}_{i=1}, \sigma')$, donde $\sigma':=(f, (s_{i,u}, s_{i,x}, s_{i,w})^{n}_{i=1})$.

El primer paso para comprobar la veracidad de la firma consiste en comprobar que el polinomio f es de grado mínimo (n-t), para asegurar que un mínimo de t usuarios han participado en la formación de la firma. Puesto que los coeficientes de f estarían guardados en σ', se trataría de comprobar que existen n-t+1 coeficientes. En caso contrario, se retornaría "false", y no se seguiría ejecutando el código. Si existen n-t+1 coeficientes, se seguiría con el siguiente paso.

El segundo paso sería computar $c_i=f(i)$, $T_{i,2}:=g_1^{s_{i,x}-c_i 2^{\gamma_1}} g_3^{s_{i,u}} A_{i,3}^{c_i}$, $T_{i,3}:=A_{i,1}^{s_{i,x}-c_i 2^{\gamma_1}} g_1^{-s_{i,w}}$ y $T_{i,4}:=A_{i,2}^{s_{i,x}-c_i 2^{\gamma_1}} g_2^{-s_{i,w}} y_i^{c_i}$ para los n usuarios, y guardarlos en dos "vectores" distintos (uno para los coeficientes c_i y otro para los parámetros T_i).

En tercer lugar, se comprobarían los tamaños en bits de los parámetros s_i (que han sido recibidos en σ), para todos los usuarios. Si alguno de ellos no cumpliese con las condiciones $s_{i,x} \in \pm\{0,1\}^{\varepsilon(\gamma_2+k)+1}$, $s_{i,u} \in \pm\{0,1\}^{\varepsilon(2\lambda+k)+1}$, $s_{i,w} \in \pm\{0,1\}^{\varepsilon(\gamma_1+2\lambda+k+1)+1}$, se retornaría "false", y no se seguiría ejecutando el código. Si todos ellos cumpliesen dichas condiciones, se continuaría con el siguiente y último paso.

La última comprobación sería $f(0)=H_{sig}(param, n, t, (y_i, A_{i,1}, A_{i,2}, A_{i,3})^{n}_{i=1}, (T_{i,1}, \cdots, T_{i,4})^{n}_{i=1}, m)$. Para ello, se introduciría en un "string" $param, n, t, (y_i, A_{i,1}, A_{i,2}, A_{i,3})^{n}_{i=1}, (T_{i,1}, \cdots, T_{i,4})^{n}_{i=1}, m$, y se llamaría al método "Setup.Hsig", pasando por referencia este "string". A continuación, se compararía el valor obtenido con $f(0)$. En caso de ser distintos, se retornaría "false", y no se seguiría ejecutando el código.

Si todas las comprobaciones anteriores fueran positivas, se retornaría “true”, indicando que la firma recibida es válida.

6. CONCLUSIONES

Se puede decir que se han cumplido satisfactoriamente los objetivos marcados en este proyecto. En primer lugar, se han encontrado muchos trabajos de algoritmos distintos para firmas grupales. El trabajo de búsqueda bibliográfica ha sido una parte fundamental en este proyecto, puesto que me ha servido para conocer distintas técnicas de firma grupal, algunas de las cuales desconocía por completo. Asimismo, de cada una de estas técnicas he visto diferentes algoritmos concretos. He logrado seleccionar los más interesantes des del punto de vista de los requerimientos que se me pedían y he analizado a fondo cada uno de ellos. Finalmente, he encontrado un algoritmo que cumplía con las exigencias que tenía, por lo que se puede afirmar que esta parte del proyecto se ha resuelto de forma totalmente satisfactoria.

En este punto, el objetivo era implementar en Java el algoritmo seleccionado. Cabe decir, llegados a este punto, que he tenido que enfrentarme a problemas de programación que escapaban a mis conocimientos. Mis estudios no son de Ingeniería Informática, sino de Ingeniería de Telecomunicaciones, por lo que mis conocimientos de programación son bastante más básicos que los que se ofrecen en Ingeniería Informática. Aun así, gracias a mi formación, he podido ir buscando la manera de resolver la mayoría de problemas a los que me he ido enfrentando. Si bien es cierto que no he podido implementar todo exactamente en la forma en la que funciona el algoritmo, sí que puede decirse que todo lo implementado funciona correctamente en correspondencia con éste, por lo que este objetivo ha quedado resuelto también satisfactoriamente. Hay que tener en cuenta que, debido al cambio de universidad (y de país) para realizar el proyecto, por los distintos calendarios que hay en ambas, no he podido terminar toda la implementación del algoritmo. De las cuatro grandes fases que presenta el algoritmo, solamente la última no he podido realizarla. Aun así, he presentado una explicación de cómo hubiese desarrollado dicha etapa en caso de haber contado con más tiempo.

A nivel personal, debo decir que la criptografía es uno de los ámbitos que más me interesan de mi carrera, a pesar de que no es uno de los que más se trabajan. El hecho de haber realizado este proyecto me ha ayudado a conocer mejor este sector, además de haber aprendido a realizar una búsqueda tan extensa como lo ha sido esta, y a profundizar en trabajos y algoritmos desconocidos para mi. En la parte de la programación, ha sido muy satisfactorio para mi aprender más gracias a este proyecto.

Así pues, para concluir, este ha sido un trabajo en el que he podido trabajar en un ámbito distinto a los que suelen trabajarse en mi carrera, lo cual me ha servido para aprender, tanto en la metodología de trabajo como en la materia concreta que he ido trabajando. Por otro lado, los objetivos que se me fijaron al inicio de este trabajo los he cumplido de forma bien satisfactoria.

7. BIBLIOGRAFIA

[1] D. Chaum and E. van Heyst, “Group signatures”, Eurocrypt 91, 1991.

[2] S. Lal and M. Kumar, “A Directed -Threshold Multi-Signature Scheme”, National Conference on Information Security, New Delhi, 2004.

[3] L. Harn, J. Ren, “Efficient identity-based RSA multisignatures”, Computers & security 27 (2008) 12–15, 2008.

[4] Y. Desmendt and Y. Frankel, “Shared generation of authenticators and signatures”, Advances in Cryptology, Crypto '91, 1991.

[5] N.-Y. Lee, “Threshold signature scheme with multiple signing policies”, IEE Proc.-Comput. Digit. Tech., Vol. 148, No. 2, March 2001.

[6] J. Y. Hwang, H. J. Kim, D. H. Lee, B. Song, “An enhanced (t,n) threshold directed signature scheme”, Information Sciences 275, 2014.

[7] V. Shoup, “Practical threshold signatures”, Eurocrypt 00, 2000.

[8] A. Boldyreva, “Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme”, PKC 2003, ed. by Y. Desmedt. LNCS, vol. 2567 (Springer, New York, 2003), 2003.

[9] Y. Yong, B. Yang, Y. Sun, “Identity-based threshold signature and mediated proxy signature schemes”, The journal of China universities of posts and telecommunications, Volume 14, Issue 2, June 2007.

[10] B. Libert, J. Quisquater, “Efficient Revocation and Threshold Pairing Based Cryptosystems”, Principles of Distributed Computing, PODC, 2003.

[11] H. Shin-Jia, C. Chiu-Chin, “New threshold-proxy threshold-signature schemes”, Computers and Electrical Engineering 31 (2005) 69–80, 2005.

[12] L. L. Wang, G. Y. Zhang and C.G. Ma, “A survey of ring signature”, J. Commun, 28, 2007.

[13] J. K. Liu, D. S. Wong, “Linkable ring signatures: Security models and new schemes”, Gervasi, O., et al. (eds.) Computational Science and Its Applications – ICCSA 2005. LNCS, vol. 3481, pp. 614–623. Springer, Heidelberg, 2005.

[14] E. Bresson, J. Stern, and M. Szydlo, “Threshold ring signatures and applications to ad-hoc groups”, Crypto 2002, LNCS 2442, Springer-Verlag, 2002.

[15] S. S. M. Chow, L. C. K. Hui, Yiu, S. M. Yiu, “Identity based threshold ring signature”, Park, C.-s., Chee, S. (eds.) ICISC 2004. LNCS, vol. 3506, Springer, Heidelberg, 2005.

[16] L. Deng, J. Zeng, “Two new identity-based threshold ring signature schemes”, Theoretical Computer Science 535, 2014.

[17] P. P. Tsang, M. Au, J. K. Liu, W. Susilo, D. S. Wong, “A suite of ID-based threshold ring signature schemes with different levels of anonymity”, Cryptology ePrint Archive, Report 2005/326, 2005.