UNIVERSIDAD DEL BÍO-BÍO
DEPARTAMENTO DE INGENIERÍA CIVIL EN INFORMÁTICA

UNIVERSITAT POLITÈCNICA DE CATALUNYA
ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA DE TELECOMUNICACIÓ BARCELONA

"Evaluación e implementación en Java de un algoritmo de firma grupal digital para preservar el anonimato en redes ad-hoc inalámbricas"

Trabajo final de carrera para optar al título profesional de
Ingeniero superior de telecomunicación

Autor
Marc Chamizo

Profesor guía
Patricio Galdames Sepúlveda

ÍNDICE

Introducción..... 2

- Descripción de la propuesta de trabajo.....3
- Objetivos generales.....4
- Objetivos específicos.....4

1 Búsqueda bibliógrafa.....6

1.1 Ad-hoc group.....6
1.2 RSA.....6
1.3 Ring Signature.....7
1.4 Multi-Signature.....9
1.5 Threshold Ring Signature.....10
1.6 Threshold Proxy Threshold Signature.....14
1.7 Comparación de algoritmos.....16

2 Implementación en Java.....18
2.1 Implementación de RSA 19
2.2 Implementación de Ring Signature.....20
2.3 Implementación de simulación de anillo.....28
2.4 Implementación de Threshold Ring Signature.....29

3 Conclusiones.....30

4 Bibliografía.....34

5 Anexo: Implementación completa del algoritmo.....35

5.1 Clase MyRSA.....35
5.2 Clase RingSignature.....36
5.3 Clase Simulation.....40

“Evaluación e implementación en Java de un algoritmo de firma grupal digital para preservar el anonimato en redes ad-hoc inalámbricas”

Alumno: Marc Chamizo
Profesor guía: Patricio Galdames Sepúlveda

Introducción

Debido a la disminución de costos de fabricación de los teléfonos celulares, sumado también a la baja de los costos de integración de dispositivos de ubicación basados en GPS, hoy en día muchas personas puedan adquirir estos aparatos y con ello demandar nuevos tipos de servicios a la Internet. Un ejemplo de ello, son los servicios basados en la ubicación, conocidos por su sigla en inglés como LBS [1] (Location Based Service).

Para solicitar un LBS, un usuario libera su posición exacta a un servidor LBS de la Internet, y este determina los detalles del servicio que le proporcionara al usuario considerando su posición. Por ejemplo, una persona demanda la pronta llegada de una ambulancia debido a un problema de salud que lo aqueja. El servidor, basado en la posición del paciente, despacha con la mayor prontitud posible la ambulancia más cercana a la ubicación.

Sin embargo, la constante liberación de la posición, puede permitir a la red conocer nuestro estilo de vida y más aún nuestra identidad. Se podría pensar que el uso de un seudónimo pudiera proteger nuestra identidad, pero la posición en si misma podría permitir a la Internet concluir con nuestra identidad. La red podría correlacionar nuestra(s) posición(es) con información geográfica obtenida de bases de datos públicas.

Se han propuesto diversas técnicas para despersonalizar la ubicación de un usuario. La más promisoria de ellas consiste en una técnica que construye para el usuario una región geográfica, llamada región de encubrimiento, que es liberada al servidor LBS en reemplazo de su posición exacta. Esta región corresponde a un conjunto de posiciones en las cuales podría estar el usuario. Mientras mayor es el tamaño del conjunto, mayor es la protección al anonimato.

El problema con esta técnica de despersonalización es que no permite que ciertas aplicaciones de autenticación física requeridas para redes ad-hoc inalámbricas puedan funcionar. Estas aplicaciones requieren verificar la posición de un usuario, que puede ser demandada para certificar que el usuario se encuentra efectivamente en un lugar físico. En este contexto, el usuario, denominado nodo a prueba, clama estar en una posición geográfica y envía un mensaje firmado digitalmente a un usuario denominado el nodo verificador, quien actúa como representante de la red para efectos de la comprobación. Este último nodo, en conjunto con otros, podría ejecutar alguna técnica de localización (multilateración o triangulación) para determinar la posición real del usuario y corroborar si el nodo a prueba esta efectivamente localizado en la posición clamada. Sin duda que esta aplicación atenta contra el anonimato, ya que el verificador concluye con posición exacta del nodo a prueba y conoce también su identidad.

Descripción Propuesta de Trabajo

Un grupo de usuarios, que conforman una red ad-hoc inalámbrica, cuyas identidades son conocidas de antemano por un nodo verificador, quien es un agente externo al grupo, claman estar al interior de una región geográfica específica (región de encubrimiento). Cada usuario transmite mensajes al nodo verificador. El verificador basado en los mensajes recibidos infiere la posición exacta de los usuarios y con esto comprueba si los usuarios están efectivamente dentro de la región señalada. Sin embargo, no debe ser capaz de identificar en qué posición exacta dentro de la región se localiza cada usuario.

Para preservar el anonimato, este proyecto de título se iniciara investigando en la literatura académica la existencia de un algoritmo de firma grupal que pueda ser empleado por los miembros de una red ad-hoc inalámbrica y que satisfaga los siguientes requisitos:

1. Cada uno de los miembros del grupo debe participar de la creación de la firma grupal.
2. La parte de la firma construida por un miembro no debe ser relacionada con la identidad de tal miembro.
3. Para comprobar la validez de la firma grupal deben combinarse todas las partes creadas por cada miembro del grupo.
4. El algoritmo de firma grupal no debe de contener ningún trap-door que permita inferir la identidad de un miembro a partir de la parte que genero este.

Segundo, se implementara una librería para Java de aquel algoritmo que cumpla con las condiciones establecidas o con aquel que cumpla con al menos dos de los criterios expuestos.

Pese a las similitudes con el proyecto de mi compañero Jordi Arumí, que intenta resolver un problema de criptografía grupal en el que varios de los miembros del grupo pueden firmar y además incorporar una trap-door para facilitar la revocación de la identidad de los miembros a un verificador externo al grupo, la propuesta presentada aquí difiere en dos aspectos simples pero fundamentales en el ámbito de la criptografía. En esta propuesta de proyecto los miembros firmantes del mensaje deben ser todos los que conformen el grupo, además no debe existir ningún tipo de trap-door para que la identidad de dichos miembros permanezca oculta de manera irrevocable. Estos dos aspectos conducirán de manera obligada a la elección de dos algoritmos distintos de criptografía grupal y por extensión a dos implementaciones en java distintas de estos algoritmos.

Conceptos Generales

El concepto de seguridad grupal fue introducido en 1991, basado en el algoritmo *RSA* [2](Rivest, Shamir y Adleman), y fue llamado *Group Signature* [3]. Aunque lograba su función, *Group Signature* tenía debilidades, entre ellas la existencia obligatoria de una *trap-door* que permitía conocer al verificador la identidad exacta del firmante. Posteriormente, en 2001 surgió el esquema conocido como *firma en anillo de umbral* [4](Threshold Ring Signature). Threshold Ring Signature especifica un grupo de posibles firmantes y la prueba irrevocable para convencer a cualquier verificador de que los autores de la firma pertenecen a este grupo, mientras la identidad permanece segura debido a la inexistencia de un gestor de revocación. Además, este

esquema ha demostrado ser muy eficiente y rápido en términos de capacidad computacional mientras los algoritmos anteriores usaban métodos robustos de computación asimétrica. Threshold Ring Signature utiliza el esquema de encriptación simétrica AES como pieza clave de su funcionamiento.

Un análisis de poca profundidad sobre el Threshold Ring Signature sería el siguiente:

→ Cada usuario resuelve de forma ordenada la parte de su firma digital empezando en un punto del anillo aleatorio

→ Una vez ha sido resuelta, se concatena a la anterior hasta formar completamente el anillo y la firma digital

→ Al verificar, el índice por donde se empieza a descifrar es aleatorio por lo que no se puede asociar una parte de la firma a ningún usuario del subgrupo

El continuo crecimiento de tecnología portable, como smartphones o tablets, ha propiciado el surgimiento de nuevos tipos de interacciones: *ad-hoc groups.* Imaginemos un grupo de usuarios que decide espontáneamente comunicar datos confidenciales, necesitaría una infraestructura de protocolos que no envolviese a ningún verificador o la certificación de claves públicas nuevas. El esquema presentado anteriormente, Threshold Ring Signature, encajaría perfectamente en esta situación ya que no requiere de una configuración de protocolo para usarse.

Objetivos del Proyecto

Objetivo general del proyecto:

"La implementación de una librería en java de un algoritmo de firma digital en anillo de umbral adecuado para su funcionamiento en redes P2P inalámbricas"

Esta librería contará con las funciones de encriptación que el firmante utilizará para realizar su parte de la firma digital, las funciones necesarias para la concatenación de las partes de la firma que formaran el anillo y finalmente, la función que el verificador utilizará para validar la firma.

Objetivos específicos

Con el fin de aportar soporte teórico en la elección del algoritmo indicado para estas necesidades, se proponen como objetivos específicos del proyecto:

1. Búsqueda bibliográfica y análisis de las fortalezas y debilidades de los siguientes algoritmos de firma grupal: Ring Signature, Threshold Ring Signature, Multisignature, Proxy Signature
2. Implementación de técnica de firma digital escogida en java
3. Evaluación de la técnica seleccionada

Carta Gantt

Se adjunta también una Carta Gantt donde se detalla el tiempo dedicado a cada tarea del proyecto.

	Actividades	Mayo				Junio				Julio			
Nº	Meses-semanas	1	2	3	4	1	2	3	4	1	2	3	4
1	Busqueda bibliografica												
2	Comparación algoritmos												
3	Librería en java												
4	Análisis del resultado												
5	Redacción del proyecto												

La estructura que el trabajo sigue a continuación es la siguiente:

-Primera parte: Búsqueda bibliográfica de algoritmos de firma digital grupal

-Segunda parte: Implementación de una librería en Java del algoritmo elegido tras la realización del estudio bibliográfico.

-Tercera parte: Conclusiones de la implementación.

1 Búsqueda bibliográfica y análisis de algoritmos de firma grupal

En esta sección se realizará un estudio matemático detallado de los algoritmos de firma grupal más conocidos o usados actualmente, para posteriormente compararlos entre ellos tomando como criterios indispensables de esta comparación los expuestos anteriormente en la descripción de la propuesta de trabajo. En esta comparación se tendrá en especial consideración la irrevocabilidad del anonimato, es decir, el algoritmo escogido debería ser anónimo en el sentido que ninguna información será filtrada sobre el grupo de firmantes excepto que dicho grupo forma un conjunto aceptado por el esquema en sí mismo. Además se tendrá en consideración la inexistencia de infraestructuras previas necesarias para usar el algoritmo de modo que su funcionamiento vaya acorde con el espíritu de las redes inalámbricas Ad-hoc.

La estructura de la primera parte es la siguiente: 1.1 Definición de grupo Ad-hoc, 1.2 Definición de RSA, 1.3 Definición de Ring Signature, 1.4 Definición de Multi-Signature, 1.5 Definición de Threshold Ring Signature, 1.6 Definición de Threshold Proxy Threshold Signature, 1.7 Comparación de algoritmos.

1.1 Ad-hoc group

Definiríamos un grupo Ad-hoc, visto desde el propósito criptográfico, del siguiente modo.

$\sum$ es un grupo de n usuarios, incluyendo sus *n* claves públicas certificadas, donde existe un subgrupo S_j de estos usuarios, llamados el subconjunto aceptado. Generalmente, las firmas digitales Ad-hoc constan de los siguientes algoritmos.

-**Algoritmo de firma Ad-hoc.** Con parámetros de entrada un mensaje *m* y la especificación de un grupo Ad-hoc $\sum(n \text{ usuarios con el subconjunto } S_j)$, devuelve una firma *σ* para el mensaje *m* que incluye todas las claves públicas de los *n* miembros.

-**Algoritmo de verificación Ad-hoc.** Con parámetros de entrada un mensaje *m* y la firma *σ*, retorna "verdadero" si *σ* es una firma valida sobre *m* y el subconjunto S_j especificado en $\sum$. Retorna "falso" en caso contrario.

Debido a que los algoritmos estudiados a continuación se encuentran en mayor o menor medida basados en RSA [2], se encuentra oportuno hacer una breve descripción de este algoritmo de encriptación básico.

1.2 RSA

En 1977, Ron Rivest, Adi Shamir, y Leonard Adleman introdujeron un algoritmo criptográfico de clave pública, el primero de este tipo, capaz tanto de cifrar mensajes como de realizar firmas digitales. Los mensajes se cifran convirtiéndolos en números, y el funcionamiento del algoritmo se basa en el producto conocido, *n,* de dos números primos secretos, *p* y *q.* La seguridad del

sistema se basa en parte en la dificultad de realizar la factorización del divisor público *n.* Cuando se quiere enviar un mensaje, el emisor busca la clave pública del destinatario, cifra el mensaje con esa clave, y cuando el mensaje cifrado llega al receptor, este se ocupa de descifrarlo con su clave privada. El esquema consta de los siguientes pasos.

-El mensaje *M* se transforma en un número *m* menor que otro número *n*, a continuación se genera el cifrado *c* mediante:

$c = m^e \bmod n$ donde *e* es la clave pública del receptor. Ahora para descifrar se realiza:

$m = c^d \bmod n$ donde *d* es la clave privada del receptor.

-**Generación de claves.** Se eligen dos números primos *p* y *q,* de forma aleatoria y con un tamaño parecido en bits. Posteriormente se calcula $n = p \cdot q$, *n* se usará como el módulo para calcular la clave pública y la privada. Usando la función Ψ de Euler , $\Psi(n) = (q-1) \cdot (p-1)$, se elige un número *e* entero positivo, que se dará a conocer como clave pública, menor que Ψ(*n*) y a la vez coprimo con Ψ(*n*). Se determina *d* como multiplicador modular inverso de *e,* $d \cdot e = \bmod n$, y se guarda como clave privada. Así pues un ejemplo del cifrado y descifrado con RSA sería :

$c = m^e \bmod n$, el procedimiento anterior funciona ya que $c^d = (m^{ed}) \bmod n$. Esto es directamente comprobable ya que $ed = 1 + k\,\Psi(n)$, entonces se cumple

$m^{ed} = m^{1 + k\,\Psi(n)} = m \bmod n$

1.3 Ring Signature

El concepto de Ring Signature [5] permite especificar un conjunto de posibles firmantes sin revelar cuál de los miembros realmente produjo la firma digital. A diferencia de los algoritmos de firma grupal [3] presentados en 1991, Ring Signature no contempla la existencia de un manager del grupo, ni agentes de revocación de identidad, permitiendo un anonimato incondicional debido a la inexistencia de una trap-door o cualquier método capaz de rastrear la clave privada de los miembros del conjunto. Ring Signature tampoco requiere de coordinación previa, ni procedimientos previos de configuración, permitiendo espontaneidad en su uso.

A continuación se detalla la propuesta presentada por Rivest, Shamir y Tauman:

Se asume que cada usuario ha recibido, vía PKI (public key infrastructure) o por certificado, una clave pública P_k con su correspondiente clave secreta S_k. El esquema de una firma en anillo consiste en los siguientes algoritmos.

-**Firma de anillo.** Algoritmo probabilístico que con los siguientes parámetros de entrada: un mensaje *m*, las claves públicas $P_1,\ldots,P_r$ de los *r* miembros del anillo, juntamente con la clave secreta S_s de un miembro específico, produce una firma en anillo *σ* para el mensaje *m.*

-**Verificación de anillo.** Algoritmo determinista que con los parámetros de entrada (*m, σ*), donde *σ* incluye las claves públicas de todos los posibles firmantes, retorna "verdadero" o "falso".

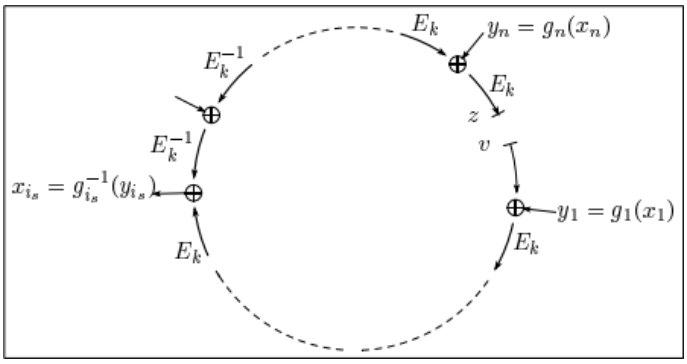
-**Definición: funciones combinadas.** Nombramos l, l_b, l_0 tres parámetros de seguridad y *v* el valor de inicialización del anillo. Consideramos *E* como esquema de encriptación simétrico definido sobre el espacio $\{0,1\}^l$ usando claves de l_0-bits de tamaño y una función de hash *H* que consigue cadenas arbitrarias de l_0-bits en ese espacio. A continuación se asume que cada usuario P_i utiliza un esquema de signatura cuya longitud l_b satisfaga l - l_b≥160.

El esquema propuesto por Rivest, Shamir y Tauman está basado en una función combinada que toma como parámetros de entrada una clave *H(m)*, el valor de inicialización *v* y un conjunto de valores arbitrarios de misma longitud que *v* denotados como y_i = $f_{i(x_i)}$=$x_i^{P_i}$mod *n,* basado en [2]. La función combinada retorna un valor *z* ϵ $\{0,1\}^l$ que para cualquier *H(m)*, *v*, cualquier índice *s* y cualquier valor fijo de $y_{i\neq s}$, $C_{H(m),v}$ es una permutación en el espacio $\{0,1\}^l$ visto como función de y_s. Además está permutación permite ser computada eficientemente del mismo modo que su función inversa.

$$z = C_{H(m),v}(y_{1,\ldots,}y_n) = E_{H(m)}(\, f_{n(x_n)} \oplus E_{H(m)}(\, f_{n-1(x_{n-1})} \oplus E_{H(m)}(\ldots \oplus E_{H(m)}\big(f_{1(x_1)} \oplus \mathrm{v}\big) \ldots)))$$

Para un índice *s*, se puede demostrar que efectivamente $C_{H(m),v}$ es una función combinada reescribiendo la ecuación del siguiente modo:

$$y_s = E_{H(m)}{}^{-1}(\, f_{s+1(x_{s+1})} \oplus \ldots E_{H(m)}{}^{-1}(\, f_{n(x_n)} \oplus E_{H(m)}{}^{-1}(z))) \oplus E_{H(m)}(\, f_{s-1(x_{s-1})} \oplus \ldots E_{H(m)}\big(f_{1(x_1)} \oplus \mathrm{v}\big) \ldots)))$$



La ecuación se verifica si *z* = *v.* Dadas todas las $y_i{}'s$, el verificador recorre el anillo comprobando si $z \stackrel{?}{=} v$. El firmante elige *v* primero, entonces va en sentido horario desde 1 hasta i_s-1 y en sentido anti-horario desde *n* hasta i_s+1. El trap-door inherente en las operaciones modulares de [2] le permite extraer x_{is} mediante la función inversa. Si el exponente en estas operaciones modulares resulta ser 3, el esquema resulta ser muy eficiente, requiriendo solo de una operación modular para firmar y de dos operaciones modulares por cada miembro del anillo para verificar.

Los autores demostraron anonimato incondicional probando que ningún atacante podría descubrir la identidad del firmante con probabilidad mayor a 1/*r*. Debido a que para cualquier *k = H(m)* y cualquier *z = v,* la ecuación *z* = $C_{H(m),v}(y_{1,\ldots,}y_n)$ tiene exactamente $(2^l)^{r-1}$ soluciones que pueden obtenerse con la misma posibilidad independientemente del firmante. Además, tampoco se puede rastrear a un firmante ya que dados dos mensajes y sus firmas pertinentes no hay forma de saber si han sido realizadas por el mismo miembro del grupo.

1.4 Multi-Signature

El algoritmo Multi-Signature [6] permite que *n* miembros distintos con claves públicas P_{k_1},…, P_{k_n} firmen colectivamente un mensaje *m*, creando una firma digital *σ* de aproximadamente el mismo tamaño en bits que tiene una firma digital estándar, certificando el mensaje simultáneamente con todas las claves P_{k_1},…, P_{k_n}. Transmitiendo *σ* en lugar de *n* firmas digitales individuales, el esquema [6] ayuda de forma eficiente a disminuir costos de comunicación. Sin embargo, se siguen necesitando todas las claves públicas de los firmantes para poder verificar la validez de la firma. En muchas aplicaciones estas claves públicas serán enviadas juntamente con la firma, complicando el objetivo principal de [6], ahorrar en ancho de banda. La inclusión de información que identifique a cada firmante parece inevitable para poder realizar la verificación, pero esta información puede estar representada de forma más eficiente que por medio de claves públicas generadas aleatoriamente. Por ejemplo, las direcciones IP de los miembros podrían satisfacer este propósito. A diferencia de [5], en [6] todos los miembros del grupo deben ponerse de acuerdo previamente para realizar la firma digital. Otra diferencia notable es la falta de anonimato total, la fuente que proporciona las claves públicas y privadas del algoritmo debe ser verificada por todos los miembros requiriendo de cierta configuración previa.

A continuación se detalla la propuesta de Mihir Bellare y Gregory Neven:

Se consideran algoritmos de firma digital interactivos aquellos en los que los firmantes se encuentran simultáneamente online y produciendo de forma interactiva la firma *σ.* Los firmantes interactúan en ciclos, donde al principio de cada ciclo cada firmante recibe un mensaje entrante de cada uno de los demás firmantes, realiza una computación y devuelve un mensaje a todos los otros miembros.

Generalmente el esquema multi-firma basado en identidad IBMS = (Setup, KeyDer, Sign, Vf) identity-based multi-signature por sus siglas en inglés, consta de cuatro algoritmos.

-Setup. Un distribuidor de claves de confianza ejecuta el algoritmo de configuración, Setup, para generar una clave pública maestra mp_k y la correspondiente clave privada ms_k.

-**Key Derivation.** Un algoritmo llamado KeyDer que cada miembro del grupo usará con parámetros de entrada ms_k y ID_i, su identificador dentro del grupo. El algoritmo computara x = H_2(ID) *d* mod *N,* donde H_2 es un oráculo aleatorio y asignará una clave privada x particular a cada firmante.

-Sign. Cada firmante utiliza su clave privada juntamente con el mensaje *m* y el conjunto L ={ID_1, … , ID_n} que contiene las identidades de todos los firmantes participantes en el protocolo. Después de un número de iteraciones, el algoritmo devuelve la multi-firma *σ* o ⊥ en caso de error durante la verificación de los parámetros de entrada. Las operaciones realizadas durante las iteraciones son las siguientes.

Iteración 1.

-El algoritmo toma como parámetros de entrada x_1, L y m.

-Elige un r_1 entero y computa r_1^e mod N = R_1, posteriormente realiza $H_0(R_1)$ = t_1. Donde H_0 es un oráculo aleatorio.

-Manda t_1 a los demás firmantes.

Iteración 2.

-Recibe t_i de cada firmante.

-Manda a cada firmante R_1 computado en la iteración anterior.

Iteración 3

-Recibe R_i de cada firmante.

-Comprueba que t_i = $H_0(R_i)$ para 2 ≤ i ≤ n, parando el algoritmo devolviendo ⊥ si al alguna de las comprobaciones es errónea.

- Por el contrario computa R = $\prod_{i=1}^{n} R_i \ mod\ N$, c = $H_1(R||L||m)$ y s_1= $r_1 x_1^c$ *mod N.*

-Manda s_1 a cada firmante.

Iteración 4

-Recibe s_i de cada firmante

-Computa s = $\prod_{i=1}^{n} s_i \ mod\ N$

-Devuelve σ = (c, s).

-Verificación. El algoritmo de verificación toma como parámetros de entrada la clave maestra pública mp_k, el conjunto L, el mensaje m y la firma σ, dando como resultado 1 si la firma es válida sobre el mensaje m y todas las identidades en L, o por el contrario 0. Para realizar esta operación, el algoritmo computa R = $s^e \ \ (\prod_{i=1}^{n} H_2(ID_i))^{-c} mod\ N$, aceptando la firma si c = $H_1(R||L||m)$.

1.5 Threshold Ring Signature

Threshold Ring Signature [4] es un algoritmo de firma grupal que extiende a [5] permitiendo que t miembros del conjunto n colaboren para realizar una firma digital. Este esquema exige un mínimo criterio de colaboración debido a que el umbral, número mínimo de participantes para que se cree la firma digital, debe ser especificado por el conjunto del grupo. El propósito sin embargo es el mismo que en [5], demostrar a un verificador externo al grupo que t miembros del grupo colaboraron activamente en la creación de una firma digital válida preservando el anonimato de todos los miembros del grupo, firmantes o no.

Se podría pensar que este problema se solucionaría simplemente construyendo *t* firmas en anillo, una de cada firmante, pero no probaría que el mensaje hubiese sido firmado por distintos miembros. Este esquema, del mismo modo que en [5], no contempla la figura de un mánager ni de un agente de revocación de identidad. La única configuración previa necesaria es el establecimiento del umbral ya que las claves públicas y privadas se reciben vía PKI.

Debido a la existencia de distintas variantes de Threshold Ring Signature, dos ejemplos serian: "Identity-Based Threshold Ring Signature" [7] donde la identidad de los miembros es conocida por todo el grupo de un modo parecido a [6] o "Threshold Signature" [8] donde existe la figura de un agente de revocación de identidad, el esquema presentado a continuación ha sido previamente seleccionado para encajar de una mejor forma que sus análogos en cuanto al cumplimiento de las especificaciones que este proyecto propone.

A continuación se expone la propuesta de Emmanuel Bresson, Jaques Stern y Michael Szydlo en "Threshold Ring Signatures and Applications to Ad-hoc groups" [4]. Primeramente se realizará un breve resumen del funcionamiento del algoritmo a modo de facilitar la comprensión matemática desarrollada a posterior:

Consideremos un anillo de *r* miembros, entre los que se encuentran dos usuarios deseando demostrar que han estado colaborando activamente para producir una firma en anillo. La idea sería dividir el grupo en dos sub-grupos desarticulados, y demostrar que cada uno de dichos sub-grupos contiene un firmante. Sin embargo, se podría comprometer el concepto de anonimato perfecto ya que esta división restringe el anonimato de cada miembro en un sub-grupo. La solución consiste en dividir el grupo un número determinado de veces, donde cada partición contenga tantos *sub-anillos* como firmantes (dos en este caso), de forma que siempre exista al menos una partición en que dos usuarios cualesquiera se encuentren en dos *sub-anillos* distintos. Se consideran estas divisiones como nodos de un *súper-anillo*. El algoritmo a continuación demuestra que al menos una de estas divisiones ha sido resuelta, es decir, 2 *sub-anillos* de una de las particiones han sido resueltos de forma individual. Para las particiones restantes, se deberá simular un ring signature, para todo *sub-anillo* no resuelto.

Notación matemática del súper-anillo.

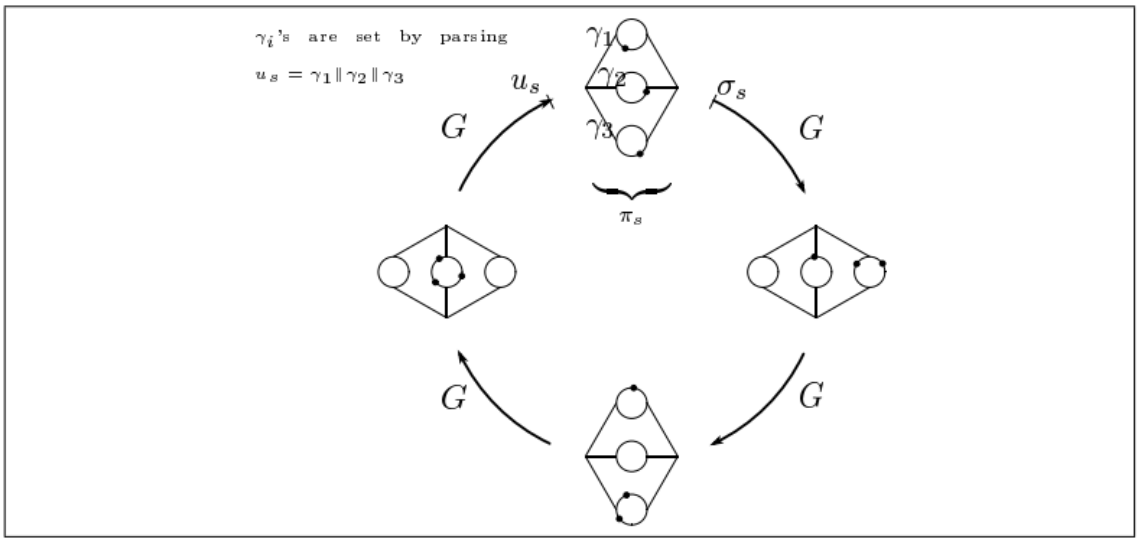
Supongamos t un entero y π= $(\pi^1, \dots \pi^t)$ una partición de [1, n] en *t* conjuntos; π define una partición del anillo R = $P_1, \dots, P_n$ en *t sub-anillos* de R^1 hasta R^t.

Caso *t* = 2. Sea π = (π^1, π^2) una partición de [1, n], P_a y P_b dos usuarios que quieren producir una firma "2-sobre-*n*" en un mensaje *m*. Si P_a y P_b pertenecen a distintos *sub-anillos* en al menos una de todas las particiones, es decir $P_a \in R^1$ y $P_b \in R^2$, entonces son capaces de producir dos firmas en anillo correctas, relativas a R^1 y a R^2 respectivamente. En este caso π es una partición justa para los índices {a, b}. En caso contrario, si $P_a, P_b \in R^1$ sería imposible producir una firma de anillo valida en el *sub-anillo* R^2.

Caso *t ≤ n.* Consideremos a (*n, t*) un sistema de particiones completo y un sub-grupo de *t* firmantes. Si π es una partición valida dentro de este conjunto, se pueden resolver todos los *sub-anillos* dentro de esta partición. Para todas las demás particiones, los *sub-anillos* serán simulados y se ubicaran a lo largo del *súper-anillo.*

En esta sección se introduce una nueva función *G*, que se comportará como una función de Hash aleatoria devolviendo cadenas de (*t* * *l*) bits. Además denotamos *p* = 2^t log *n*. Se asume que para cualquier entero *n* y *t* ≤ *n*, un sistema de particiones completo (*n, t*) es viable: $\prod_{\mathrm{n}} = \{\pi_{1,\ldots}\pi_p\}$. Por último se introduce la notación $C_{v,m,i_o}(\gamma, y_j$j ϵ *R*) para cualquier *j* ϵ [1, *t*].

Otra variante interesante de [5] que este algoritmo en particular tiene es la siguiente: el esquema original define que una firma en anillo es válida cuando satisface *z* = *v*. Sin embargo esta es una comprobación puramente arbitraria y no necesaria en sí para ofrecer seguridad. Los autores de este esquema reemplazaron esta condición *rellenando* el vacío (gap) que había entre *z y v,* siempre que dicho vacío no pueda ser elegido por un falsificador. De modo que la ecuación se vería validada si *z* = *v* $\oplus$ $E_k(0)$ en lugar de $z = v$.



En el ejemplo de la figura, *t* = 3. La partición π_s es una partición valida ya que cada sub-anillo ha sido resuelto por un firmante. En las demás particiones, siempre hay al menos un sub-anillo que queda sin resolver.

-Algoritmo de firma. El objetivo de este algoritmo es validar un conjunto de *sub-anillos* que correspondan a una partición justa (cada firmante pertenece a un *sub-anillo*), para posteriormente concatenar el resultado y aplicar un mecanismo parecido al de [5] con la finalidad de demostrar que al menos un conjunto de *sub-anillos* ha sido validado completamente. Se denota π_s como una partición válida para I = { i_1, … i_t}. El algoritmo consta de 6 pasos.

1. Elección de puntos de partida aleatorios para cada *sub-anillo* de cada partición.

Para *i* = 1,…, p (número de particiones) {

Para *k* = 1,…, t (número de sub-anillos)

$v_i^k = \xleftarrow{R} (0,1)^l$

}

2. Simular anillos para todas las particiones excepto la partición válida π_s.

Para *i* = 1,…, p; *i* ≠ *s* {

Para *j* = 1,…*n* { $x_i^j = \xleftarrow{R} (0,1)^l$, y luego $y_i^j \leftarrow g_j(x_i^j)$}

(Siendo x_i^j el bit i-esimo del string *j* y g_j la función extendida modular de RSA, f_i(x), para lograr que todas las $f_i's$ estén en el mismo dominio.)

Para *k* = 1,…*t* { $z_i^k \leftarrow C_{v_i^k,1,m}(0, y_i^j, j \in \pi_i^k\ (R))$, y luego $\gamma_i^k \leftarrow v_i^k \oplus z_i^k$

}

3. Calcular el *Súper-anillo* con los *gap's* calculados en el paso anterior.

$\sigma_s \xleftarrow{R} (0,1)^{tl}$, y $u_{s+1} \leftarrow G\ (\sigma_s)$

Para *i = s+2,…, p, 1,…s,* {

$u_i \leftarrow G\ (u_{i-1} \oplus (\gamma_{i-1}^1 || \ldots \gamma_{i-1}^t))$

}

4. Calcular los *gap's* de la partición válida π_s.

$(\gamma_s^1 || \ldots || \gamma_s^t) \leftarrow u_s \oplus \sigma_s$

5. Resolver los *sub-anillos* de la partición válida π_s.

Para *j* ∈ [1, *n*]\I, { $x_s^j = \xleftarrow{R} (0,1)^l$, y luego $y_s^j \leftarrow g_j(x_i^j)$

Para *j* ∈ I, {

$\sigma_k = \xleftarrow{R} (0,1)^l$ para *k* tal que *j* ∈ π_s^k

$y_s^j \leftarrow C_{j,\gamma_s^k,m}^{-1}\ (\sigma_s, y_s^j, j \in \pi_s^k$ (R)) y $x_s^j \leftarrow g^{-1}(y_s^j)$ }

6. Finalizar la firma.

$v \xleftarrow{R} [1, \mathrm{p}]$ y enviar (v, u_v, $\cup_{1 \le i \le p}(x_i^1, \ldots, x_i^n, v_i^1, \ldots, v_i^n)$)

-**Algoritmo de verificación.**

1. Calcular todos los anillos empezando por el índice 1.

Para *i* = 1,…, p {

Para *j* = 1,…*n* { $y_i^j \leftarrow g_j(x_i^j)$}

Para *k* = 1,…*t* { $z_i^k \leftarrow C_{v_i^k,1,m}(0, y_i^j, j \in \pi_i^k\ (R))$, y luego $\gamma_i^k \leftarrow v_i^k \oplus z_i^k$

}

2. Verificar el *súper-anillo* desde el índice *v* y los gaps obtenidos

$$u_v \overset{?}{\leftarrow} G\,(\gamma_{v-1}^1||\ldots||\gamma_{v-1}^t \oplus G\,(\ldots G\,(\gamma_v^1||\ldots||\gamma_v^t \oplus u_v)\ \ldots)).$$

Este esquema utiliza un *súper-anillo* para demostrar que al menos una de las particiones ha sido resuelta correctamente. Podríamos ver un anillo, tanto un *sub-anillo* o un *súper-anillo,* como un operador lógico "OR" mientras cada partición podría ser vista como un operador "AND".

1.6 Threshold Proxy Threshold Signature

El esquema de criptografía grupal (*c, m)* Threshold Proxy *(t, n)* Threshold Signature [9] permite que *t* firmantes originales de un grupo de *n* firmantes originales autoricen a un conjunto llamado *Proxy group* de *m* miembros que *c* miembros o más puedan cooperar para generar una firma digital. Este esquema permite la delegación de la creación de firmas digitales para evitar que los firmantes originales puedan falsificar la firma proxy, mientras los firmantes proxy no pueden falsificar firmas en nombre de los firmantes originales. La identidad de los firmantes originales permanece anónima de forma incondicional además, en la etapa de verificación se comprueba la veracidad del certificado de umbral proxy antes de comprobar la veracidad de la firma proxy de umbral.

Con la finalidad de mantener un alto nivel de seguridad, la autorización *proxy* para delegar la potestad de realizar firmas a un grupo externo al conjunto de firmantes originales debe ser aceptada también por ese grupo externo. En otras palabras, no solamente los *t* firmantes originales colaboran para crear el certificado *proxy* sino que un número de firmantes proxy deberá ser parte en la generación del certificado. En el esquema presentado a continuación se tiene en cuenta dos casos según el número de firmantes proxy que participen de la creación del certificado. En el primer caso, el número de firmantes proxy es el valor del umbral. En el segundo caso, la totalidad de los firmantes proxy participa en la creación del certificado.

El esquema presentado por Hwang Shin-Jia y Chen Chiu-Chin está dividido en cuatro fases: La configuración del sistema, la generación del certificado proxy de umbral, la generación de la firma digital y por último, la fase de verificación.

En la fase de configuración del sistema un agente de confianza al que se llama *T* es el responsable de la construcción de los parámetros del sistema. Entre estos parámetros se encuentran los

números primos *p* y *q*, las identidades pertenecientes a cada miembro del grupo original ID_{ui}, las identidades de los miembros del grupo *proxy* ID_{pj}, y por último la generación de las correspondientes claves públicas y privadas a través de un polinomio secreto de grado *t-1*.

En la fase de generación del certificado proxy de umbral, los firmantes proxy también cooperan activamente. En este esquema se supone dos casos según el grado de cooperación: la generación del certificado proxy de umbral con acuerdo parcial de los miembros proxy, o la generación del certificado con acuerdo total de los miembros proxy. El proceso que se realiza en esta fase es el siguiente:

-Cada firmante original U_i elige un número entero al azar k_{0i} y realiza la operación K_{0i} = $g^{k_{0i}}$ mod *n*, enviando K_{0i} a *T*, a los firmantes proxy y a los *t-1* firmantes originales restantes. El operador *g* es un generador público de orden *q* elegido por *T*. Por otra banda, cada firmante proxy P_j elige un número entero al azar k_{pj} y realiza la operación K_{pj} = $g^{k_{pj}}$ mod *n*, enviando K_{pj} a *T*, a los firmantes proxy y a los *m-1* firmantes proxy restantes.

-Todos los firmantes *t*, los firmantes proxy *c* y el grupo *T* realizan la siguiente computación:

K = $\prod_{i=1}^{t} K_{0i} \prod_{j=1}^{c} K_{pj}$ *mod p.*

-Cada firmante original U_i computa v_{0i} = $f_0(x_{0i})\, L_{si} y_0 h(w)$ + K_{0i} *K mod q*, para posteriormente enviárselo a *T.* Cada firmante proxy computa v_{pj} = $s_{pj}' y_{pj}'\, h(w)$ + $k_{pj} K \bmod q$, y lo envía a *T*.

-*T* comprueba la veracidad de v_{0i} y v_{pj} mediante la función *g*. Cuando todas han sido verificadas, computa *V* = $\sum_{i=1}^{t} v_{0i}$ + $\sum_{j=1}^{c} v_{pj}$ *mod q.* Finalmente, los firmantes proxy reciben el certificado proxy de umbral (*K, V*).

En la fase de generación de la firma digital, los *c* miembros del grupo proxy realizan el siguiente proceso.

-Cada firmante P_j elige un entero d_{pj} con el que computara D_{pj} = $g^{d_{pj}}$ mod *p*, para mandárselo a los *c-1* firmantes proxy restantes.

-Posteriormente se calcula *D* = $\prod_{j=1}^{c} D_{pj}$ mod *p*, y cada firmante proxy envía (D_{pj}, s_{pj}) a *T*. El firmante P_1 también envía (*w, (K, V), M*), donde *w* es la garantía proxy que contiene las claves públicas del conjunto *m.*

-*T* verifica cada firma individual proxy usando la función *g*. Computa *S* = $\sum_{j=1}^{c} s_{pj}$ mod *q*. Finalmente la firma proxy de umbral sobre el mensaje *M* es generada como:

(*w, (K, V), M, (D, S))*

En la fase de verificación de la firma digital se verifica el certificado (*w, (K, V), M, (D, S))* en dos pasos. En la primera etapa se verifica *(K, V)*, usando distintas funciones dependiendo de si el certificado *proxy* fue creado por *c* o *m* miembros del grupo proxy. Si la primera verificación resulta errónea se rechaza (*w, (K, V), M, (D, S))*. En la segunda etapa se verifica *(D, S)* usando la función *g*, si resulta verdadera la firma digital proxy de umbral se da por válida.

1.7 Comparación de algoritmos

En esta sección se realizará una comparación de los algoritmos expuestos anteriormente con la finalidad de encontrar cual o cuales se ajustan mejor a los retos que este proyecto desea resolver. Los puntos clave de esta comparación serán: Anonimato incondicional e irrevocable, cada uno de los miembros del grupo debe participar de la creación de la firma grupal, infraestructuras previas casi inexistentes y por último, facilidad de uso en redes Ad-hoc inalámbricas. Se tendrá también en especial consideración la dificultad del cálculo computacional además del tiempo empleado para generar y verificar dichos algoritmos.

Para facilitar la tarea de la comparación, esta sección se dividirá en 3 partes, donde se tratarán temas específicos y como cada esquema presentado anteriormente se comporta en esa situación. Finalmente, se añadirá un gráfico a modo de ayuda con la finalidad de obtener una respuesta visualmente más sencilla.

1.7.1 Anonimato incondicional

La irrevocabilidad del anonimato y la inexistencia de una trap-door en los esquemas, permitirían alcanzar uno de los objetivos principales de este proyecto. De los cuatro esquemas presentados anteriormente, los que aportan anonimato incondicional a sus firmantes son: Ring Signature, Threshold Ring Signature y Threshold Proxy Threshold Signature. En estos esquemas, el verificador conoce únicamente el conjunto de claves públicas de los miembros del grupo, firmantes y no firmantes.

Por el contrario, en el esquema Multi-Signature el verificador requiere del conjunto de las identidades de los firmantes. Como ya se desarrolló durante la explicación de dicho esquema, estas identidades podrían ser las direcciones IP o cuentas de correo electrónico, pero ese dato en sí mismo ya supone un riesgo para el anonimato del firmante.

1.7.2 Participación de la totalidad del conjunto

La cooperación de los miembros para poder realizar una firma digital es imprescindible para el cumplimiento de los objetivos que este proyecto desea solucionar. En los esquemas presentados anteriormente, los algoritmos que permiten que todo el conjunto de firmantes sea participe de la firma son: Threshold Ring Signature, Multi-Signature y Threshold Proxy Threshold Signature.

Ring Signature solo ofrece la posibilidad de que un firmante participe en la creación de la firma digital.

1.7.3 Infraestructuras previas

Con el objetivo de resolver el problema planteado en este proyecto, la espontaneidad de los usuarios de redes Ad-hoc inalámbricas para formar conjuntos debe tenerse en especial consideración a la hora de escoger un algoritmo de firma grupal. Esta espontaneidad solo puede darse en caso de que la infraestructura previa requerida para el correcto funcionamiento del esquema sea casi inexistente. De los esquemas presentados anteriormente, cumplen esta condición: Ring Signature y Threshold Ring Signature. En estos dos esquemas, el conjunto de claves públicas y privadas se obtiene vía PKI o por certificado.

Por el contrario, Multi-Signature requiere la previa coordinación de todos los miembros para validar la obtención de claves y la generación de una clave pública maestra. Del mismo modo, Threshold Proxy Threshold Signature requiere un ente externo de confianza que genere los parámetros necesarios para el uso del algoritmo.

1.7.4 Gráfico

En esta sección se presentara el gráfico que resume el resultado de las comparaciones anteriores.

	Anonimato	Totalidad de firmantes	Imposibilidad de revocación	Infraestructura previa	Cálculo Computacional
Ring Signature	SI	NO	SI	NO	Aceptable
Multi-Signature	NO	SI	SI	SI	Reducido
Threshold Ring Signature	SI	SI	SI	NO	Aceptable
Threshold Proxy Threshold Signature	SI	SI	SI	SI	Elevado

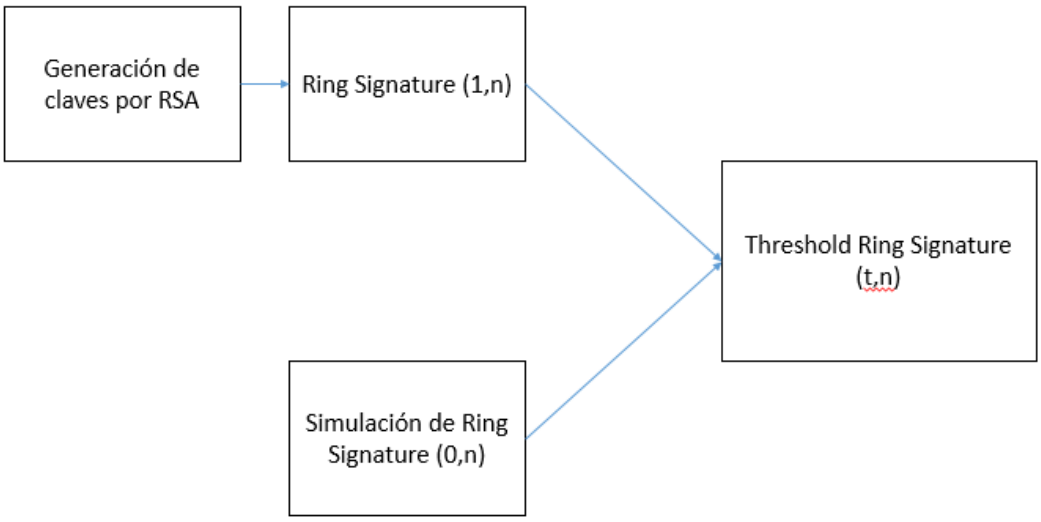
Del gráfico anterior se puede concluir que el esquema de firma grupal que mejor se adapta a las necesidades que este proyecto desea satisfacer es Threshold Ring Signature. Este esquema de firma digital grupal resulta ser muy eficiente en cuanto al tiempo computacional requerido para su funcionamiento, ya que solamente se realiza una operación modular para cada miembro del conjunto durante la generación de la firma digital utilizando llaves de encriptación asimétricas, mientras todas las demás fases de encriptación se realizan mediante un esquema de encriptación simétrico de 256 bits. Además todos los miembros firmantes colaboran activamente en la creación de la firma digital, siendo esta segura aunque partes de la firma sean obtenidas por un atacante. Ofreciendo de este modo un anonimato irrevocable a todos los miembros del conjunto.

2 Desglose del algoritmo e implementación en Java

En esta sección del documento se detallarán las partes del algoritmo Threshold Ring Signature y de los componentes necesarios para su implementación en java. De este modo, se ofrece una vista detallada del mismo a la par que se facilita su entendimiento para la posterior explicación de los métodos desarrollados en la librería.

Tras la previa explicación sobre el funcionamiento de Threshold Ring Signature, este algoritmo puede ser visto como una extensión del algoritmo de firma grupal Ring Signature. Esto es debido a que cada nodo del considerado *súper-anillo* contiene *t sub-anillos* regidos por el algoritmo de Ring Signature. Ofreciendo de este modo seguridad incondicional a las partes que forman el esquema de encriptación por separado. El otro factor trascendental para la creación del algoritmo Threshold Ring Signature es el concepto de simulación de anillo, resumido en la primera parte, permitiendo así que la identidad de los miembros firmantes permanezca oculta dentro del grupo, siempre que la simulación no pueda ser realizada por un atacante.

Tras estas consideraciones, se cree oportuno que la implementación del algoritmo sea dividida en cuatro partes: Generación de claves, Ring Signature, Simulación de anillo y Threshold Ring Signature. Cada parte corresponderá a una clase distinta de la librería, capaz de funcionar de forma independiente o conjunta según el uso requerido. De este modo se puede ofrecer una implementación completa y eficiente al separar el objetivo en distintas fases.



La estructura de la segunda parte es la siguiente: 2.1 Implementación de RSA, 2.2 Implementación de Ring Signature, 2.3 Implementación de la simulación de anillo, 2.4 Implementación de Threshold Ring Signature.

2.1 Implementación de RSA

El algoritmo de encriptación propuesto por Rivest, Shamir y Adleman cuenta con una implementación de Java llamada KeyPairGenerator. Para el propósito de este proyecto se han modificado partes del código fuente, con el objetivo de que los outputs de los métodos sean guardados en un array de BigIntegers, de este modo se facilita su posterior uso.

El método principal de esta clase se llama *Generate*, será el encargado de generar claves públicas y privadas para todos los miembros del anillo. El conjunto de todas las llaves será guardado en una matriz de tamaño n*3 posteriormente.

```
public  BigInteger[] Generate() {

    r = new Random();
    p = BigInteger.probablePrime(bitlength/2, r);
    q = BigInteger.probablePrime(bitlength/2, r);
    N = p.multiply(q);

    phi = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
    e = BigInteger.probablePrime(bitlength, r);

    while (phi.gcd(e).compareTo(BigInteger.ONE) > 0 && e.compareTo(phi) < 0 ) {
        e.add(BigInteger.ONE);
    }
    d = e.modInverse(phi);

    BigInteger[] result = new BigInteger[3];
    result[0] = e;
    result[1] = N;
    result[2] = d;

    return result;
}
```

Juntamente con esta clase se encuentra la clase *GeneratePUB*, idéntica a la anterior con la única diferencia de que en lugar de devolver una clave privada en el último espacio del array devuelve el valor *null*. Este método será el utilizado por los miembros no firmantes del conjunto a modo de no comprometer su llave privada ya que no será usada. El tamaño elegido para el cálculo de las llaves es de 1024 bits, este tamaño ofrece alta seguridad a la vez que no compromete de forma notoria el tiempo computacional empleado en realizar operaciones modulares.

Por último, los métodos restantes de esta clase se llaman *encrypt* y *decrypt*. Serán los encargados de cifrar y descifrar parámetros pertenecientes a cada miembro del conjunto total del grupo para poder construir la firma digital sobre el mensaje. Además, el verificador usará las claves públicas de todos los miembros para crear ciertos parámetros necesarios para comprobar la veracidad de la firma digital.

```java
public BigInteger encrypt(BigInteger message,BigInteger ei,BigInteger Ni){
    return (message).modPow(ei,Ni);
}


public BigInteger decrypt(BigInteger message,BigInteger di,BigInteger Ni) {
    return (message).modPow(di,Ni);
}
```

2.2 Implementación de Ring Signature

Este esquema de encriptación grupal presentado por Rivest, Shamir y Tauman en 2001, permite que un miembro de un grupo pueda realizar una firma digital sobre un mensaje en representación de todo el conjunto. Además, ofrece anonimato incondicional ya que ningún verificador externo conocerá la identidad del miembro firmante, pero tendrá pruebas irrefutables de que dicho miembro pertenece al conjunto conocido.

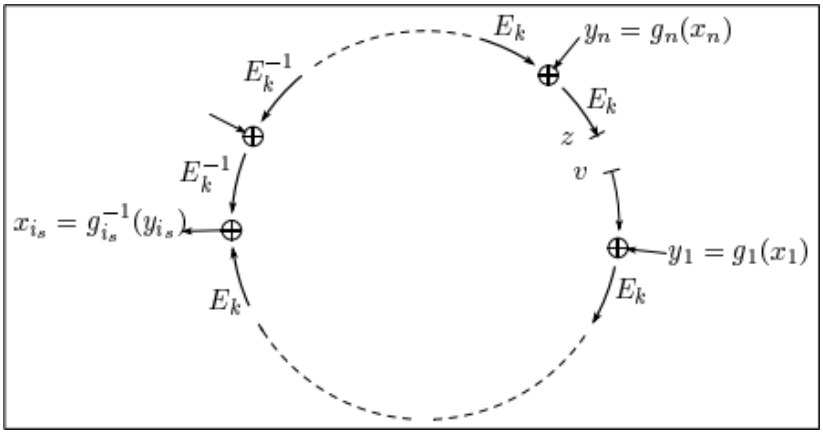
Para la generación de las particiones que formarán cada nodo del *súper-anillo* de Threshold Ring Signature, se construirán tantos Ring Signature como firmantes colaboren de la firma grupal. De este modo, el algoritmo podría verse de una forma simple cómo $p{\cdot}\sum_{i=1}^{t}\pi_i$. Donde p representa el número de particiones que forma el *súper-anillo* y π_i los anillos dentro de las particiones, entre los que se encuentran anillos verídicos y anillos simulados. Sin duda, una implementación eficiente de Ring Signature facilitaría la posterior implementación de Threshold Ring Signature. Por este hecho, en este proyecto se aborda primeramente la implementación de este algoritmo en lugar de la implementación del algoritmo Threshold Ring Signature.

Para la realización de una implementación eficiente de Ring Signature, a continuación se detallarán aspectos intrínsecos de su funcionamiento que no han sido comentados con anterioridad en la primera parte, juntamente con aspectos que ya han sido comentados y se recordarán a modo de facilitar una comprensión completa del funcionamiento al lector. Al final de cada fase del algoritmo se comentarán los métodos que se requerirían para su funcionamiento, finalmente se añadirá un esquema donde se muestre la relación que todos los métodos tienen entre ellos durante la generación y la verificación de la firma digital.

Recordemos las funciones clave en la generación de Ring Signature, la función combinada y su inversa:

$$z = C_{H(m),v}(y_{1,\dots,}y_n) = E_{H(m)}(\, f_{n(x_n)} \oplus E_{H(m)}(\, f_{n-1(x_{n-1})} \oplus E_{H(m)}(\dots \oplus E_{H(m)}\big(f_{1(x_1)} \oplus \mathrm{v}\big) \dots)))$$

$$y_s = E_{H(m)}{}^{-1}(\, f_{s+1(x_{s+1})} \oplus \dots E_{H(m)}{}^{-1}(\, f_{n(x_n)} \oplus E_{H(m)}{}^{-1}(z))) \;\oplus E_{H(m)}(\, f_{s-1(x_{s-1})} \oplus \dots E_{H(m)}\big(f_{1(x_1)} \oplus \mathrm{v}\big) \dots)))$$



Desglose del algoritmo:

1. El firmante elegirá una clave para el esquema de encriptación simétrico, esta clave se computará como el resultado de una función de hash donde el input será el propio mensaje:

   k = H (m)

   Para la realización del primer paso se requiere la implementación de una función de hash, en este caso la librería java.security.MessageDigest consta de los métodos para la creación de esta función. La función de hash elegida es SHA-256, con modificaciones para poder extender el tamaño de la salida al que se desee.

```
public static byte[] HashFunction(byte[] mensaje,int tamaño ) throws NoSuchAlgorithmException{
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
         digest.update(mensaje);
         byte[] digest1 = digest.digest();
         byte[] res = new byte[tamaño/8];
         int dlen = digest1.length;
         for (int i = 0; i < res.length / dlen; i++) {
             System.arraycopy(digest1, 0, res, i * dlen, dlen);
         }
         return res;

    }
```

   Como esquema de encriptación simétrico se utilizará AES, esquema de encriptación simétrica llamado Advanced Encryption Scheme [10], que permite encriptar y desencriptar inputs con la misma clave. Los parámetros encriptados tendrán el mismo tamaño que la llave usada para la encriptación. La librería de java javax.crypto.Cipher, ofrece métodos destinados a encriptación.

```java
public static byte[] encripta(byte[] claveSimetrica, byte[] input)
    SecretKeySpec sks = new SecretKeySpec(claveSimetrica,"AES");
    Cipher cipher  = Cipher.getInstance("AES/ECB/NoPadding");
    cipher.init(Cipher.ENCRYPT_MODE,sks);

    return cipher.doFinal(input);

}

public static byte[] desencripta(byte[] claveSimetrica,byte[] input)
    SecretKeySpec sks = new SecretKeySpec(claveSimetrica,"AES");
    Cipher cipher =  Cipher.getInstance("AES/ECB/NoPadding");
    cipher.init(Cipher.DECRYPT_MODE,sks);
    return cipher.doFinal(input);

}
```

De este modo, los métodos necesarios para la encriptación de los parámetros y_i durante el cómputo del anillo quedan resueltos.

2. Elección del parámetro de inicialización aleatorio *v* dentro del conjunto $\{0,1\}^l$, juntamente con la elección de *n-1* parámetros x_i dentro del conjunto $\{0,1\}^{lb}$.

Para la elección de parámetros aleatorios se ha usado la función Random de java, aunque el algoritmo de esta función no puede ser considerado tan aleatorio como los algoritmos que rigen los Secure Random Generator, generadores aleatorios más seguros, la función Random sirve al propósito que este proyecto desea conseguir. Sin duda, el uso de Secure Random Generator beneficiaria en el cálculo de ciertos parámetros pertenecientes a la firma digital pero cabe añadir que dichos parámetros tendrán un tamaño de 1024 bits por lo que un ataque por fuerza bruta requeriría de un alto coste computacional tanto si esos parámetros son calculados por Random o Secure Random Generator.

El problema que se presentó a continuación fue como solucionar los cambios de espacio entre el conjunto l y el conjunto l_b. Las operaciones que se realizarían sobre parámetros pertenecientes a un conjunto debían dar como resultado nuevos parámetros pertenecientes al segundo conjunto. La solución en java a este problema consistió en la creación de dos métodos llamados *trimLeading* y *leadingZero*. Se realizarán todos los cálculos sobre el mismo espacio, entonces se truncara el bit de mayor importancia siempre que este sea un cero o se añadirá un cero según a que espacio se desee transformar el array de bytes. De este modo, una operación matemática que en principio parecía complicada transformar a código informático quedó resuelta de una forma simple y elegante.

Los métodos *trimLeading* y *leadingZero* son los siguientes:

```
public static byte[] trimLeading(byte[] b) {
    if (b.length % 128 == 0) {
        return b;
    }
    if (b[0] != 0) {
        throw new IllegalStateException("Attempting to trim " + b[0]);
    }
    byte[] res = new byte[b.length - 1];
    for (int i = 0; i < res.length; i++) {
        res[i] = b[i + 1];
    }
    return res;
}

public static byte[] leadingZero(byte[] b) {
    byte[] res = new byte[b.length + 1];
    for (int i = 0; i < b.length; i++) {
        res[i + 1] = b[i];
    }
    res[0] = 0;
    return res;
}
```

3. Computo de $f(x_i)$ y función *"xor"* ajustada para recibir inputs de un array de bytes para las funciones combinadas.

La función f(x) se refiere a la conocida *Trapdoor one-way function* de RSA, es decir: $c = m^e \bmod n$. En este caso, todas las llaves públicas y privadas tendrán el mismo tamaño en bits, ya que los métodos de la clase RSA se encargan de ello. De este modo no será necesario extender esta función a *g(x)* para que los parámetros encriptados pertenezcan a un conjunto del mismo tamaño. En caso contrario, si las llaves tuvieran un tamaño distinto, los métodos *encrypt* y *decrypt* de RSA deberían modificarse con la función *g(x):*

$$g_i(x) = \begin{cases} q_i n_i + f_i(r_i) & \text{if } (q_i + 1)n_i \leq 2^\ell \\ x & \text{otherwise} \end{cases}$$

De modo que para el cálculo de $f(x_i)$ y $f^{-1}(x_i)$ se utilizarán los métodos de la clase RSA que han sido expuestos en el apartado 2.1.

El siguiente método a implementar en la clase Ring Signature para poder construir la función combinada y su inversa fue el método *xor.* La función *xor* ya está implementada en java, pero para su uso en la implementación del algoritmo algunos aspectos deben ser modificados. La función modificada de *xor* se asegurará primero de que los dos inputs estén en el mismo espacio, para posteriormente realizar la suma lógica byte a

byte. Finalmente, el resultado será almacenado en un array del mismo tamaño que los inputs.

```
public static byte[] xor(byte[] a,byte[] b){
  int len = a.length;
    if (len != 128) {
            throw new IllegalStateException("not posible");
    }
    while (len < b.length) {
        b = trimLeading(b);

    }
    while (len > b.length) {
        b = leadingZero(b);

    }
    byte[] result = new byte[len];
    for (int i = 0; i < len; i++) {
        result[i] = (byte) (((int) a[i]) ^ ((int) b[i]));
    }
    return result;

}
```

4. Solucionar la inversa de la función combinada para que el firmante obtenga su valor y_s. Por definición, dados unos valores arbitrarios para los demás parámetros de entrada, existe un solo y_s que satisface la ecuación dando como resultado el valor *v*.

La función combinada y su inversa serán las piezas clave de la creación de la firma digital y de la verificación. Para su implementación se requieren los métodos *xor,* los métodos del esquema de encriptación simétrico, el valor de inicialización *v*, el índice en el que se encuentra el firmante *i = s* y finalmente el array de parámetros y'_s. Todos ellos han sido calculados con anterioridad a excepción de la posición del firmante dentro del conjunto, esta posición se calculara en el método principal de la clase.

Para que el firmante pueda obtener su valor y_s debe solucionar la ecuación:

$$y_s = E_{H(m)}{}^{-1}(f_{s+1(x_{s+1})} \oplus \ldots E_{H(m)}{}^{-1}(f_{n(x_n)} \oplus E_{H(m)}{}^{-1}(z))) \oplus E_{H(m)}(f_{s-1(x_{s-1})} \oplus \ldots E_{H(m)}(f_{1(x_1)} \oplus \mathrm{v}) \ldots)))$$

De este modo podrá demostrar que al menos un valor x_s no fue elegido de forma aleatoria. Para poder solucionarla, el firmante recorre el anillo en sentido horario desde la posición 1 hasta la posición s-1 y en sentido anti-horario desde la posición n hasta la posición s+1. La suma lógica *xor* de ambos resultados le permitirá obtener el valor de y_s.

La implementación de esta ecuación lleva el nombre de *Combining_d_b,* de la abreviación *combining direct and backwards,* y se muestra a continuación:

```
public static byte[] Combining_d_b(byte[] key,byte[] v, BigInteger[] y,int s)
    byte[] temp1 = v;
    byte[] temp2 = v;
    int r = y.length;
    temp1=desencripta(key,temp1);
    for(int i = r-1; i > s; i--){
        temp1 = xor(temp1,y[i].toByteArray());
        temp1=desencripta(key,temp1);
    }
    for(int j=0;j<s;j++){
        temp2=xor(temp2,y[j].toByteArray());
        temp2=encripta(key,temp2);
    }
    return xor(temp1,temp2); //value of ys
}
```

Juntamente con la implementación de la inversa de la función combinada necesaria para el cálculo del valor y_s, la pieza clave restante en el algoritmo es la función combinada directa. Esta será la encargada de recorrer el anillo en sentido horario desde el valor de inicialización *v* hasta el resultado final. Este método será utilizado durante el cómputo de la firma digital para comprobar la veracidad de la misma, después de depurar posibles errores futuros en las fases finales de la implementación este método deberá ser eliminado de la generación del anillo para no comprometer la seguridad. Así mismo, el método *Combining_direct* será el método principal de la fase de verificación.

```
public static byte[] Combining_direct(byte[] key,byte[] v,BigInteger[] y)
    byte[] value = v;
    for(int i =0;i<y.length;i++){
        value = xor(value,y[i].toByteArray());
        value = encripta(key,value);
    }
    return value;
}
```

5. Extracción del parámetro x_s y creación de la firma digital Ring Signature.

En este punto, todos los requerimientos necesarios para poder construir la firma digital han sido realizados. El método Ring Signature será el encargado de encontrar la posición del firmante dentro del array donde se almacenan todas las llaves públicas y una sola de privada, computar todos los valores *y* de los miembros no firmantes, ejecutar el método *Combining_d_b* para obtener el valor y_s, obtener el valor x_s mediante la llave privada del firmante y finalmente construir la firma digital. La firma resultante tendrá una longitud de (2**n* +1), donde *n* es el número de miembros del conjunto participe de la firma, en ella se incluirán todas las llaves públicas de los miembros juntamente con los valores *x's* pertenecientes a cada miembro y finalmente el valor de inicialización aleatorio *v.* El método Generate_Ring se muestra a continuación:

```java
public static byte[][] Generate_Ring(BigInteger[][] keys, byte[] message, Random r)
    myRSA rsa = new myRSA();
    byte[] k = HashFunction(message,256);
    byte[] v = new byte[1024/8];
    r.nextBytes(v);
    int s = -1;
    for(int i = 0; i<keys.length;i++){
        if(keys[i][2]!=null){
          s=i;
        }

    }

    if (s == -1) {
        throw new IllegalStateException("1 prib key needed");
    }
   BigInteger[] x = new BigInteger[keys.length];
   BigInteger[] y = new BigInteger[keys.length];
   for(int i = 0; i<keys.length;i++){
        if(i!=s){

            x[i] = new BigInteger(1024,r);
            y[i] = rsa.encrypt(x[i],keys[i][0],keys[i][1]); //(ei,Ni,Null)
        }
   }



    byte[] signer = Combining_d_b(k,v,y,s);
    y[s] = new BigInteger(signer);
    x[s] = rsa.decrypt(y[s],keys[s][2],keys[s][1]);

    //TESTING
    System.out.print("Value of Ys extracted with pribkey:\n"+y[s]);
    System.out.print("\n");
    byte[] check = Combining_direct(k, v, y);
    int d = new BigInteger(check).compareTo(new BigInteger(v));
    if (d != 0) {
        throw new IllegalStateException("invalid signature");
    }
    //end of Test
    byte[][] result = new byte[keys.length + 1][];
    for (int i = 0; i < keys.length; i++) {
        byte[] X = x[i].toByteArray();
        if (X.length == 129) {
            X = trimLeading(X);
        }
        result[i] = X;
    }
    result[keys.length] = v;
    return result;

}
```

6. Verificación de la firma digital

Esta es la última fase del algoritmo, en la que un verificador externo al conjunto de posibles firmantes certificará la veracidad de la firma digital. Para poder conseguir este propósito, el verificador requerirá de la firma digital dónde se incluyen las claves públicas de todos los miembros del conjunto, todos los valores *x's* y finalmente el valor de inicialización *v*. Usando la llave pública de cada miembro calculará el valor y_i que le corresponda. A continuación deberá utilizar el método *Combining_direct*, expuesto anteriormente, y comprobar si la salida de dicho método coincide con el valor de inicialización dando como resultado una firma verídica. En caso de que el valor no coincida, el verificador certificará una firma digital falsa.
Es necesario hacer hincapié en que el agente externo que ejecuté el método de verificación no tendrá acceso al código, por lo que no podrá relacionar el valor x_s creado por el firmante con la correspondiente llave pública. De este modo, el agente que realice el rol de verificador, solo tendrá acceso a la firma digital, al mensaje y la certeza de si dicha firma digital es verdadera o falsa.
El método *Verify_Ring* se muestra a continuación:

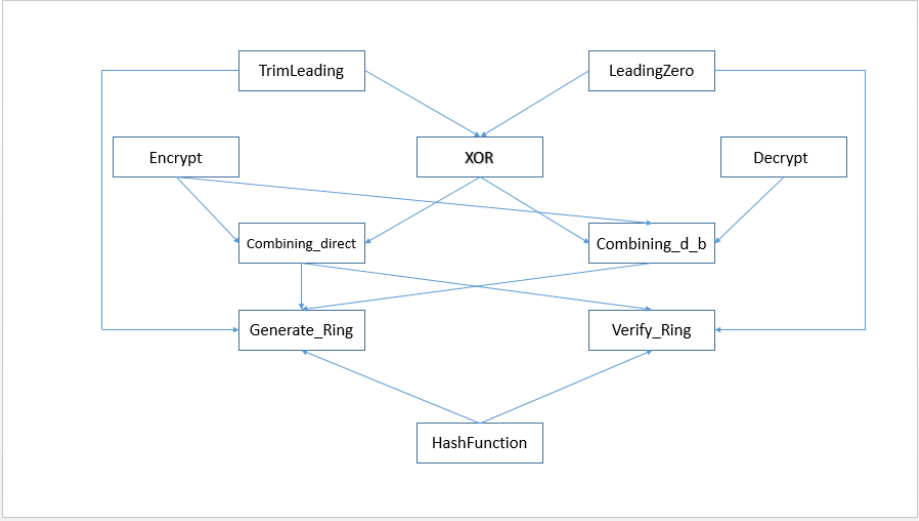
```
public static boolean Verify_Ring(byte[][] signature,BigInteger[][] keys,byte[] message)
byte[] k = HashFunction(message,256);
byte[] v = signature[signature.length-1];
myRSA rsa = new myRSA();
BigInteger[] x = new BigInteger[signature.length-1];
BigInteger[] y = new BigInteger[x.length];
for(int i =0;i<x.length;i++){
    x[i]=new BigInteger(leadingZero(signature[i]));
    y[i]=rsa.encrypt(x[i],keys[i][0],keys[i][1]);
   }

 byte[] z = Combining_direct(k,v,y);
 return new BigInteger(z).equals(new BigInteger(v));
}
```

7. Método principal de la clase Ring Signature e interconexión de todos los métodos.

El método principal de esta clase será el punto de contacto entre el firmante y la implementación del algoritmo Ring Signature. El firmante indicará el número total de miembros que forman el conjunto y su posición, ya sea aleatoria o elegida, dentro del conjunto. A continuación, el firmante introducirá el mensaje sobre el que se desea realizar la firma digital y ejecutará el método *Generate_Ring.*
La existencia de un método *main* dentro de la clase Ring Signature, ofrece la posibilidad de utilizar este algoritmo de forma independiente a Threshold Ring Signature.

Para finalizar el desglose del algoritmo Ring Signature, se adjunta a continuación un diagrama de los métodos que forman esta clase de java, ofreciendo al lector una visión de la interconexión de todos los métodos desarrollados anteriormente.



Por último, se añade a continuación una captura de pantalla en la que se ve lo que muestra la implementación del algoritmo por pantalla en un ejemplo donde el número de integrantes en el conjunto es de *n* = 4. Primeramente, se pueden observar los tres valores aleatorios x_i y sus correspondientes y_i. A continuación se calcula el valor y_s para extraer el valor x_s con la llave privada del firmante *s*. Por último se genera la firma digital de tamaño ((*n*1024)+1).* En este caso, la firma digital es de 5120 bits.

```
Valor aleatorio de x[i]
171396008399615433213683905339200823363883589485128598689425016532862008340681787613424300377947375861073082391504227660971439327036470787756271088163767181731711900000369285008291
Valor aleatorio de y[i] computado a partir de x[i]
121580637372247225352851937028462991001353853969026983160307816784926069844200338601573943786190096642862478342604512440207075821878865826438597402370298855969696780222563341487400
Valor aleatorio de x[i]
394600283738077932715894310356555180920091932350511724800762334944073761425162944994579614510730994016334360758424598572325486489764645365612572800833006002799859703464340953106582
Valor aleatorio de y[i] computado a partir de x[i]
103980324474265169235530938148155845628117484032402459390315912883678593112729025507709742357789349985677177022031916838009340342568105067111026734762065168636143918890722157399057
Valor aleatorio de x[i]
102565266721179811325121683016153418229676966531909580260784611658311271316942191838557177934286009202455792702055524390780894774260384347014972115148299014253066500623198659037011
Valor aleatorio de y[i] computado a partir de x[i]
504529363976158302796319783557648152759478890887290687759368452045998992336215098826058180436225537051930337327679367884769396187791180659325782783534658530679308662651781245695765
Valor y[s] extraido de la inversa de la función combinada
326693631627332985070676356530847417374055417981314253474648880507888437921158754502328358500683258888755685387825748049773657381807791247247190477546549383482795144874167416176438
Valor de x[s] extraido con la llave privada del usuario s:
326693631627332985070676356530847417374055417981314253474648880507888437921158754502328358500683258888755685387825748049773657381807791247247190477546549383482795144874167416176438

This is the Signature
[[52, -61, -17, 70, 6, 121, -69, -107, 81, 76, -73, -107, 55, 126, -6, -73, -100, -93, -68, -69, 71, 3, -121, -14, -108, 63, 64, 33, -34, 95, -23, 21, 6, 69, 117, 57, -83, -25, 92,
The Signature is :
true
BUILD SUCCESSFUL (total time: 1 second)
```

2.3 Implementación de la simulación de anillo

El siguiente paso para poder realizar una implementación en java del algoritmo Threshold Ring Signature es la implementación de una Simulación de anillo. Para un funcionamiento correcto del algoritmo Threshold Ring Signature los anillos pertenecientes a cada partición que no hayan sido resueltos, ningún usuario ha utilizado su llave privada para obtener el valor x_s, deben ser simulados. El concepto de simulación de anillo consiste en modificar la condición de verificación de Ring Signature para poder cerrar el anillo sin la necesidad de extraer el valor y_s a través de la inversa de la función combinada. En lugar de cerrar el anillo cuando la salida de la función combinada directa *z* es igual a *v,* la solución consiste en *rellenar el espacio* que separa *z* de *v* de forma que ese espacio no pueda ser rellenado por un atacante. Este nuevo parámetro referido al espacio entre el valor de inicialización del algoritmo y el valor final recibe el nombre de *gap*. La nueva condición de verificación para los anillos simulados es la siguiente: $\Upsilon = z \oplus v$.

De este modo la simetría se ve rota, pero de una forma crucial, permitiendo la simulación de anillos para los anillos no resueltos de las particiones que forman *súper-anillo*. Este *gap* puede introducirse entre dos posiciones consecutivas cualesquiera del algoritmo. Para facilitar su comprensión e implementación, las posiciones elegidas serán la última y la primera posición de la función combinada.

La implementación de la clase Simulate Ring constará de los mismos métodos existentes en la clase Ring Signature debido a su similitud en ciertos aspectos, sin embargo algunos de estos métodos deberán ser modificados o eliminados en su totalidad. Los métodos *HashFunction, Encripta, Desencripta, trimLeading, leadingZero, xor* y *Combining_direct* no se verán modificados pues su funcionamiento e implementación fue previsto de forma que pudieran ser utilizados en las implementaciones de la simulación del anillo y de Threshold Ring Signature. A diferencia de los métodos *Combining_d_b* y *Verify_Ring* que han sido eliminados de esta clase pues su uso no es necesario. Ningún usuario utilizará su llave privada para extraer el valor x_s de modo que por extensión no es necesario realizar el cálculo de la inversa de la función combinada, pues todos los valores x_i serán elegidos aleatoriamente. De forma similar, una implementación de un método de verificación carece de sentido, pues el objetivo de la simulación de anillo es justamente producir una firma digital falsa de tal modo que ningún atacante pueda producirla del mismo modo. Por último se añade el método *Simulate_Ring* que funcionará de forma parecida al método *Generate_Ring*. En este caso, se calcularán todos los valores para poder llamar al método *Combining_direct* y por último se obtendrá el valor del *gap* para incluirlo en la última posición de la firma digital.

```
public static byte[][] Simulate_Ring(BigInteger[][] keys, byte[] message, Random r)
 myRSA rsa = new myRSA();
 byte[] k = HashFunction(message,256);
 byte[] v = new byte[1024/8];
 r.nextBytes(v);
 BigInteger[] x = new BigInteger[keys.length];
 BigInteger[] y = new BigInteger[keys.length];
 for(int i = 0; i<keys.length;i++){


        x[i] = new BigInteger(1024,r);
        y[i] = rsa.encrypt(x[i],keys[i][0],keys[i][1]);

 }
 byte[] cd = Combining_direct(k,v,y);
 byte[] gap = xor(v,cd);
 byte[][] result = new byte[keys.length + 2][];
   for (int i = 0; i < keys.length; i++) {
     byte[] X = x[i].toByteArray();
     if (X.length == 129) {
         X = trimLeading(X);
     }
     result[i] = X;
   }
 result[keys.length] = v;
 result[keys.length+1] = gap;
 return result;
}
```

2.4 Implementación de Threshold Ring Signature

El esquema de encriptación de firma digital de umbral fue expuesto por primera vez en 2001 por Emmanuel Bresson, Jaques Stern y Michael Szydlo. Su esquema ofrecía la posibilidad de que una parte de un conjunto de personas colaboren activamente en la creación de una firma digital.

El objetivo principal de este proyecto consistía en una implementación en java de este algoritmo. Tras la búsqueda bibliográfica y el consiguiente análisis matemático de los algoritmos, se estimó necesario realizar primeramente una implementación de Ring Signature y de la simulación de anillo como se ha comentado con anterioridad. Indiscutiblemente, este hecho limitó de forma notoria el tiempo dedicado en exclusiva a la implementación del algoritmo Threshold Ring Signature. Este proyecto ha tenido una duración de 3 meses, debido a la falta de tiempo y a la dificultad de la implementación de estos algoritmos matemáticos no se ha podido realizar una implementación completa del algoritmo ya que eso requeriría de aproximadamente dos meses más para poder presentar una implementación completamente funcional y eficiente. Por este motivo, a continuación se expondrá un esquema de los métodos que una clase Threshold Ring Signature debería tener juntamente con una breve descripción de cada método.

Tras la explicación e implementación de los componentes fundamentales para la construcción del esquema Threshold Ring Signature, a continuación se detallará el funcionamiento del algoritmo y cómo debería implementarse.

Primeramente, se elegirán valores aleatorios de inicialización para cada *sub-anillo* de cada partición. Del mismo modo que en Ring Signature, el firmante elegía aleatoriamente el valor de inicialización *v,* en este esquema se elegirán $t*p$ valores de inicialización. El número de particiones depende del número de firmantes del siguiente modo: $p = 2^t log\ n$. El reparto de usuarios dentro de cada *sub-anillo* de cada partición debería realizarse como una permutación, con el único requisito de que exista al menos una partición en la que cada firmante resuelva uno de los *sub-anillos* que la forman. Esa partición será conocida con el nombre de partición válida.

A continuación se simularán *p-1* particiones, calculando el *gap* de cada *sub-anillo* simulado y concatenando ese valor con el *gap* correspondiente en la siguiente partición. La implementación de este proceso podría realizarse del siguiente modo. En una matriz de tamaño $p*t$ se concatenarían todos los *gaps* del primer *sub-anillo* de cada partición en la fila 0, ocupando así hasta la posición *p-1* donde el primer firmante computaría su valor y_s teniendo en cuenta el *gap* acumulado para poder completar la primera fila de la matriz. Análogamente, la segunda fila de la matriz la calcularía el segundo firmante y de este modo hasta que todos los firmantes hayan computado su valor y_s respectivamente. Posteriormente, la partición válida podría colocarse de forma aleatoria en cualquier columna de la matriz. Para poder completar la firma digital, se deberá añadir a la matriz todos los valores x_i y finalmente los valores de inicialización de todos los *sub-anillos* de cada partición. Por último la verificación consistirá en resolver cada fila de la matriz $p*t$ una vez todos los parámetros hayan sido reconstruidos. Dando la firma digital por válida si cada fila ha sido resuelta.

Si la implementación se decide realizar de este modo, hay métodos en la clase RingSignature que no deberían ser modificados. Como la función de hash, la implementación de AES, xor, o los métodos para el cambio de espacios. En caso contrario, si se decidiera trabajar tomando cada

partición como un único nodo en lugar de ver el *súper-anillo* como anillos superpuestos la función de hash deberá modificarse para que la salida tenga una longitud de *t*l* bits. La misma modificación deberá hacerse en los métodos de AES.

Por último debería añadirse el método para la generación del *súper-anillo* y la verificación de la firma digital. El primero será el encargado de obtener los valores aleatorios de *x,* calcular los correspondientes parámetros *y* con las llaves públicas de los usuarios y finalmente acumular los valores de los *gaps* en la correspondiente fila de la matriz. Por último, invocará a una modificación del método *Combining_d_b*, en el que se le incluirá el valor del *gap* acumulado como parámetro de entrada, para poder obtener el valor y_{s_i} correspondiente. El método encargado de la verificación, *Combining_direct* deberá ser modificado del mismo modo para incluir como parámetro de entrada el *gap* para poder reconstruir todos los parámetros requeridos para certificar la validez de la signatura digital.

Análisis de seguridad de la implementación

Como en cualquier aplicación de seguridad que se deseé descentralizar, por ejemplo entre usuarios de una red P2P, debería garantizarse el uso de un sistema de comunicación seguro. La implementación presentada anteriormente, da por sentado la existencia de este canal de comunicación que ofrece seguridad a sus usuarios cuando el generador de llaves públicas y privadas las distribuye.

Todos los demás parámetros que se usan durante la construcción de la firma digital grupal no comprometen la identidad de los miembros ni del firmante, pues para descubrir a que usuario corresponde cada parámetro requeriría de su llave privada. Por lo que de momento, el único requisito indispensable para la distribución de este algoritmo es el canal de comunicaciones seguro.

Por último, el método encargado de certificar la validez de la firma digital debería hacerse privado en cuanto se distribuya al agente verificador, para que no pueda saber en qué índice del anillo se encontraba el firmante. Aunque la información que obtendría es relativamente pequeña, podría llevar a concluir con la identidad de los miembros.

3 Conclusiones

Este proyecto de fin de carrera tenía como objetivo poder dar solución a un problema de criptografía grupal en grupos Ad-hoc cumpliendo los siguientes requisitos establecidos:

1. Cada uno de los miembros del grupo debe participar de la creación de la firma grupal.
2. La parte de la firma construida por un miembro no debe ser relacionada con la identidad de tal miembro.
3. Para comprobar la validez de la firma grupal deben combinarse todas las partes creadas por cada miembro del grupo.
4. El algoritmo de firma grupal no debe de contener ningún trap-door que permita inferir la identidad de un miembro a partir de la parte que genero este

En base a estos requisitos se establecieron los objetivos generales y específicos del proyecto. La búsqueda bibliográfica juntamente con el análisis de las fortalezas y debilidades de distintos algoritmos de firma digital grupal fue el primero de los objetivos específicos. Esta parte fue fundamental para el posterior desarrollo del trabajo. Expandió mi conocimiento sobre el ámbito de la criptografía, en el que tenía un conocimiento muy superficial adquirido durante una asignatura en mi carrera universitaria. Resultó además un reto a nivel personal poder conseguir una completa asimilación de algoritmos de encriptación matemáticos tan complejos. Finalmente, tras un profundo análisis de los esquemas expuestos en este documento se dio por cumplido este objetivo cuando se eligió un algoritmo de criptografía grupal que cumplía los requisitos expuestos previamente y además conseguí un completo entendimiento del funcionamiento del algoritmo que me permitiría comenzar con la implementación en java del mismo.

El siguiente objetivo específico de este proyecto de fin de carrera era la implementación del algoritmo seleccionado durante la fase previa. Fue en este punto donde me encontré con los mayores obstáculos del trabajo pues mis conocimientos de programación eran reducidos, esto requirió una profunda dedicación a recordar el funcionamiento del lenguaje de programación java además de un estudio previo que permitió realizar una implementación por partes de forma eficiente. Todos los obstáculos que han aparecido durante los dos meses que ha durado la fase de la implementación del algoritmo han sido resueltos satisfactoriamente, gracias al continuo crecimiento de mi nivel de programación. De las cuatro partes en las que el algoritmo fue dividido, he conseguido una implementación eficiente y completamente funcional de tres de las partes. Además, la mayoría de los métodos ya implementados serán necesarios para la implementación de la parte final del algoritmo, cuya realización podría construirse del modo expuesto anteriormente. Debido a la diferencia de calendarios entre mi universidad de origen y la universidad donde se ha desarrollado el proyecto, la duración del proyecto se ha visto reducida de modo que ha resultado imposible lograr una implementación completa del algoritmo de firma digital grupal Threshold Ring Signature. Pese a este inconveniente, más de las tres cuartas partes del objetivo específico han sido resueltas satisfactoriamente.

Desde un punto de vista académico deseo recalcar que este proyecto me ha ayudado a aumentar mi conocimiento sobre criptografía y programación, ámbito en el que me gustaría seguir formándome. Además de poder tratar un tema, a mi parecer trascendental en el futuro

de las tecnologías, como es el anonimato de los usuarios en internet y la protección de sus datos. En el ámbito personal, este proyecto ha resultado ser un reto apasionante que día a día me ha ofrecido la posibilidad de ahondar en un tema que cada vez me parece más absorbente y apasionante. En un futuro próximo, es mi intención proseguir con la implementación de este esquema de encriptación grupal así como la de seguir formándome en el sector de la criptografía.

4 Bibliografía

[1] Jörgen Johansson, Timo Rantalainen, “Location Based Services”, Ericsson and Nokia, 1995

[2] Ron Rivest, Adi Shamir, Leonard Adleman, “RSA”, Massachussets Institute of Technology, 1977

[3] D. Chaum, E. van Heyst, “Group Signatures”, Eurocrypt’91

[4] Emmanuel Bresson, Jacques Stern, Michael Szydlo, “Threshold Ring Signature and Applications to Ad-hoc Groups”, Crypto’02, Santa Barbara, Californnia

[5] Ron Rivest, Adi Shamir, Yair Tauman, “How to leak a secret”, Asiacrypt’01

[6] Mihir Bellare, Gregory Neven, “Identity-Based Multi-Signatures from RSA”, 2007

[7] Sherman S.M. Chow, Lucas C.K. Hui, and S.M. Yiu, IdentityBasedThreshold Ring Signature, Department of Computer Science TheUniversity of Hong Kong Pokfulam, Hong Kong, 2006

[8] Benoît Libert, Jean-Jacques Quisquater, “Efficient revocation and threshold pairing based cryptosystems”, Massachussets, 2001

[9] Shin-Jia Hwang, Chiu-Chin Chen, “New Threshold-Proxy Threshold Signature schemes”, Computer & Electrical Engineering, 2005

[10] John Daemen, Vincent Rijmen, “RIJNDAEL”, Instituto Nacional de Estándares y Tecnología (NIST), 1997

5 ANEXO: Implementación completa del algoritmo

Clase MyRSA:

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

public class myRSA {

   private BigInteger p;
   private BigInteger q;
   private BigInteger N;
   private BigInteger phi;
   private BigInteger e;
   private BigInteger d;
   private final int bitlength = 1024;  // 1024 o 2048 bits
   private Random r;


    public  BigInteger[] Generate() {

      r = new Random();
      p = BigInteger.probablePrime(bitlength/2, r);
      q = BigInteger.probablePrime(bitlength/2, r);
      N = p.multiply(q);

      phi = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
      e = BigInteger.probablePrime(bitlength, r);

      while (phi.gcd(e).compareTo(BigInteger.ONE) > 0 && e.compareTo(phi) < 0 ) {
         e.add(BigInteger.ONE);
      }
      d = e.modInverse(phi);

      BigInteger[] result = new BigInteger[3];
      result[0] = e;
      result[1] = N;
      result[2] = d;

      return result;
   }

    public BigInteger[] GeneratePUB() {

      r = new Random();
      p = BigInteger.probablePrime(bitlength/2, r);
      q = BigInteger.probablePrime(bitlength/2, r);
      N = p.multiply(q);

      phi = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
      e = BigInteger.probablePrime(bitlength, r);

      while (phi.gcd(e).compareTo(BigInteger.ONE) > 0 && e.compareTo(phi) < 0 ) {
         e.add(BigInteger.ONE);
      }
      d = null;

      BigInteger [] result = new BigInteger[3];
      result[0] = e;
      result[1] = N;
      result[2] = null;
      return result;
   }
```

```
  public static void main (String[] args) throws IOException{
  while(true){
     myRSA rsa = new myRSA();
     Random r = new Random();
     BigInteger mensaje;
    mensaje = BigInteger.probablePrime(512,r); //COMO MAXIMO 1023
    BigInteger encrypt,decrypt;
    BigInteger[] result = rsa.Generate();
    System.out.print("El valor es : \n"+mensaje);
    System.out.print("\n");
    encrypt = rsa.encrypt(mensaje,result[0],result[1]);
    System.out.print("encriptado : \n"+encrypt);
    System.out.print("\n");
    decrypt = rsa.decrypt(encrypt,result[2],result[1]);
    System.out.print("desencriptado : \n"+decrypt);
    System.out.print("\n");
    if(mensaje.equals(decrypt)){
       System.out.print("\n");
       System.out.print("TRUE\n");
        System.out.print("\n");
    }

    }
  }
       //Añadir métodos getPublic & getKeys
      //Para n-1 y para el firmante

   public BigInteger encrypt(BigInteger message,BigInteger ei,BigInteger Ni){
     return (message).modPow(ei,Ni);
  }


  public BigInteger decrypt(BigInteger message,BigInteger di,BigInteger Ni) {
     return (message).modPow(di,Ni);
  }

  public boolean has_private(){
     return d != null;
  }


}
```

Clase RingSignature:

```
import java.math.BigInteger;
import java.security.InvalidKeyException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Arrays;
import java.util.Random;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.SecretKeySpec;

public class RingSignature {

  public static byte[] HashFunction(byte[] mensaje,int tamaño ) throws NoSuchAlgorithmException{
       MessageDigest digest = MessageDigest.getInstance("SHA-256");
        digest.update(mensaje);
        byte[] digest1 = digest.digest();
        byte[] res = new byte[tamaño/8];  //La división entre 8 permite modificar la longitud del output
        int dlen = digest1.length;
        for (int i = 0; i < res.length / dlen; i++) {
           System.arraycopy(digest1, 0, res, i * dlen, dlen);
        }
        return res;
```

```
    }

  public static byte[] encripta(byte[] claveSimetrica, byte[] input)  throws NoSuchAlgorithmException, NoSuchPaddingException,
InvalidKeyException, IllegalBlockSizeException, BadPaddingException{
    SecretKeySpec sks = new SecretKeySpec(claveSimetrica,"AES");
    Cipher cipher  = Cipher.getInstance("AES/ECB/NoPadding");
    cipher.init(Cipher.ENCRYPT_MODE,sks);
    return cipher.doFinal(input);

  }

  public static byte[] desencripta(byte[] claveSimetrica,byte[] input) throws InvalidKeyException, IllegalBlockSizeException,
BadPaddingException, NoSuchAlgorithmException, NoSuchPaddingException{
    SecretKeySpec sks = new SecretKeySpec(claveSimetrica,"AES");
    Cipher cipher =  Cipher.getInstance("AES/ECB/NoPadding");
    cipher.init(Cipher.DECRYPT_MODE,sks);
    return cipher.doFinal(input);

  }

  public static byte[] trimLeading(byte[] b) {
    if (b.length % 128 == 0) {  //1024 % 8    en caso de llaves de 2048 la longitud de los bloques para XOR sería de 256
      return b;
    }
    if (b[0] != 0) {
      throw new IllegalStateException("Attempting to trim " + b[0]);
    }
    byte[] res = new byte[b.length - 1];
    for (int i = 0; i < res.length; i++) {
      res[i] = b[i + 1];
    }
    return res;
  }

  public static byte[] leadingZero(byte[] b) {

    if(b[0]==0){
      return b;
    }

    byte[] res = new byte[b.length + 1];
    for (int i = 0; i < b.length; i++) {
      res[i + 1] = b[i];
    }
    res[0] = 0;
    return res;
  }

  public static byte[] xor(byte[] a,byte[] b){
   int len = a.length;
    if (len !=128) {   // idéntico que en el método trimLeading
        throw new IllegalStateException("not posible");
    }
    while (len < b.length) {
      b = trimLeading(b);

    }
    while (len > b.length) {
      b = leadingZero(b);

    }
    byte[] result = new byte[len];
    for (int i = 0; i < len; i++) {
      result[i] = (byte) (((int) a[i]) ^ ((int) b[i]));
    }
    return result;

  }

  public static byte[] Combining_direct(byte[] key,byte[] v,BigInteger[] y)   throws NoSuchAlgorithmException,
NoSuchPaddingException, InvalidKeyException, IllegalBlockSizeException, BadPaddingException{
```

```
    byte[] value = v;
    for(int i =0;i<y.length;i++){
      value = xor(value,y[i].toByteArray());
      value = encripta(key,value); // Valor Z proyección de Ys
    }
    return value;
  }


  public static byte[] Combining_d_b(byte[] key,byte[] v, BigInteger[] y,int s) throws InvalidKeyException,
IllegalBlockSizeException, BadPaddingException, NoSuchAlgorithmException, NoSuchPaddingException{
    byte[] temp1 = v;
    byte[] temp2 = v;
    int r = y.length;
    temp1=desencripta(key,temp1);
    for(int i = r-1; i > s; i--){
      temp1 = xor(temp1,y[i].toByteArray());
      temp1=desencripta(key,temp1);
    }
    for(int j=0;j<s;j++){
      temp2=xor(temp2,y[j].toByteArray());
      temp2=encripta(key,temp2);
    }
    return xor(temp1,temp2); //value of ys
  }

  public static byte[][] Generate_Ring(BigInteger[][] keys, byte[] message, Random r) throws NoSuchAlgorithmException,
InvalidKeyException, IllegalBlockSizeException, BadPaddingException, NoSuchPaddingException {
    myRSA rsa = new myRSA();
    byte[] k = HashFunction(message,256);
    byte[] v = new byte[1024/8]; //Los bloques serán de 128

    r.nextBytes(v);

    int s = -1;
    for(int i = 0; i<keys.length;i++){
      if(keys[i][2]!=null){
        s=i;
      }

    }

    if (s == -1) {
      throw new IllegalStateException("1 prib key needed");
    }
     BigInteger[] x = new BigInteger[keys.length];
     BigInteger[] y = new BigInteger[keys.length];
    int u = 0;
    do{
    for(int i = 0; i<keys.length;i++){
      if(i!=s){

        x[i] = new BigInteger(1024,r);

        System.out.print("Valor aleatorio de x[i]\n"+x[i]);
        System.out.print("\n");
        y[i] = rsa.encrypt(x[i],keys[i][0],keys[i][1]); //(ei,Ni,Null)
        System.out.print("Valor aleatorio de y[i] computado a partir de x[i]\n"+y[i]);
        System.out.print("\n");
      }
    }
    byte[] signer = Combining_d_b(k,v,y,s);
    y[s] = new BigInteger(leadingZero(signer));
    x[s] = rsa.decrypt(y[s],keys[s][2],keys[s][1]); //
    System.out.print("Valor y[s] extraido de la inversa de la función combinada\n"+y[s]);
    System.out.print("\n");

    System.out.print("Valor de x[s] extraido con la llave privada del usuario s:\n"+y[s]);
    System.out.print("\n");

     BigInteger control = rsa.encrypt(x[s],keys[s][0],keys[s][1]);   //Control reduntante debido x error en la generación de aleatorio
```

```
        System.out.print("\n");
        u = y[s].compareTo(control);


      }while(u!=0);  //Si se resuelve el error, el do-while es innecesario

      byte[] check = Combining_direct(k, v, y);   //TEST
      int d = new BigInteger(check).compareTo(new BigInteger(v));
      if (d != 0) {
         throw new IllegalStateException("invalid signature");
      }  //END of TEST

      byte[][] result = new byte[keys.length + 1][];
      for (int i = 0; i < keys.length; i++) {
         byte[] X = x[i].toByteArray();
          if (X.length == 129) {
           X = trimLeading(X);
         }
         result[i] = X;
      }
      result[keys.length] = v;
      return result;

   }

   public static boolean Verify_Ring(byte[][] signature,BigInteger[][] keys,byte[] message) throws NoSuchAlgorithmException,
NoSuchPaddingException, InvalidKeyException, IllegalBlockSizeException, BadPaddingException{
   byte[] k = HashFunction(message,256);
   byte[] v = signature[signature.length-1];
   myRSA rsa = new myRSA();
   BigInteger[] x = new BigInteger[signature.length-1];
   BigInteger[] y = new BigInteger[x.length];
   for(int i =0;i<x.length;i++){
      x[i]=new BigInteger(leadingZero(signature[i]));
      y[i]=rsa.encrypt(x[i],keys[i][0],keys[i][1]);
     }

    byte[] z = Combining_direct(k,v,y);
    return new BigInteger(z).equals(new BigInteger(v));
   }


   public static void main (String[] args) throws NoSuchAlgorithmException, InvalidKeyException, IllegalBlockSizeException,
BadPaddingException, NoSuchPaddingException{
   byte[] mensaje = new byte[8];

   Random r = new Random (5000);
   int n = 4;
    //int n = r.nextInt(20)+1;
   // int s = r.nextInt(n);
    int s = 0;
    myRSA rsa = new myRSA();
    BigInteger[] result = new BigInteger[3];
    BigInteger[][] keys = new BigInteger[n][3];//de 0 a n-1 -> n firmantes
   for (int i = 0; i<n;i++){
       if(i!=s){

          result =  rsa.GeneratePUB();  //El método getPublic facilitaría este paso
          keys[i][0] = result[0];//ei
          keys[i][1] = result[1];//Ni
          keys[i][2] = result[2];//di
       }
    }
    result = rsa.Generate(); //Identico pero con el método getKeys
    keys[s][0] = result[0];
    keys[s][1] = result[1];
    keys[s][2] = result[2];

    byte[][] sig = Generate_Ring(keys,mensaje,r);
    System.out.print("This is the Signature\n"+Arrays.deepToString(sig));
    boolean ver = Verify_Ring(sig,keys,mensaje);
```

```
    System.out.print("\n");
    System.out.print("The Signature is :\n"+ver+"\n");



}

}
```

Clase Simulation:

```
import java.math.BigInteger;
import java.security.InvalidKeyException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Arrays;
import java.util.Random;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.SecretKeySpec;

public class Simulation {


   public static byte[] HashFunction(byte[] mensaje,int tamaño ) throws NoSuchAlgorithmException{
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
         digest.update(mensaje);
         byte[] digest1 = digest.digest();
         byte[] res = new byte[tamaño/8];
         int dlen = digest1.length;
         for (int i = 0; i < res.length / dlen; i++) {
            System.arraycopy(digest1, 0, res, i * dlen, dlen);
         }
         return res;

      }

   public static byte[] encripta(byte[] claveSimetrica, byte[] input)  throws NoSuchAlgorithmException, NoSuchPaddingException,
InvalidKeyException, IllegalBlockSizeException, BadPaddingException{
      SecretKeySpec sks = new SecretKeySpec(claveSimetrica,"AES");
      Cipher cipher  = Cipher.getInstance("AES/ECB/NoPadding");
      cipher.init(Cipher.ENCRYPT_MODE,sks);

      return cipher.doFinal(input);

   }

   public static byte[] desencripta(byte[] claveSimetrica,byte[] input) throws InvalidKeyException, IllegalBlockSizeException,
BadPaddingException, NoSuchAlgorithmException, NoSuchPaddingException{
      SecretKeySpec sks = new SecretKeySpec(claveSimetrica,"AES");
      Cipher cipher =  Cipher.getInstance("AES/ECB/NoPadding");
      cipher.init(Cipher.DECRYPT_MODE,sks);
      return cipher.doFinal(input);

   }

   public static byte[] trimLeading(byte[] b) {
      if (b.length % 128 == 0) {
         return b;
      }
      if (b[0] != 0) {
         throw new IllegalStateException("Attempting to trim " + b[0]);
      }
      byte[] res = new byte[b.length - 1];
      for (int i = 0; i < res.length; i++) {
         res[i] = b[i + 1];
      }
      return res;
```

```
    }

    public static byte[] leadingZero(byte[] b) {
        byte[] res = new byte[b.length + 1];
        for (int i = 0; i < b.length; i++) {
            res[i + 1] = b[i];
        }
        res[0] = 0;
        return res;
    }

    public static byte[] xor(byte[] a,byte[] b){
      int len = a.length;
        if (len != 128) {
                throw new IllegalStateException("not posible");
        }
        while (len < b.length) {
            b = trimLeading(b);

        }
        while (len > b.length) {
            b = leadingZero(b);

        }
        byte[] result = new byte[len];
        for (int i = 0; i < len; i++) {
            result[i] = (byte) (((int) a[i]) ^ ((int) b[i]));
        }
        return result;

    }

    public static byte[] Combining_direct(byte[] key,byte[] v,BigInteger[] y)   throws NoSuchAlgorithmException,
NoSuchPaddingException, InvalidKeyException, IllegalBlockSizeException, BadPaddingException{
        byte[] value = v;
        for(int i =0;i<y.length;i++){
            value = xor(value,y[i].toByteArray());
            value = encripta(key,value);
        }
        return value;
    }


        public static byte[][] Simulate_Ring(BigInteger[][] keys, byte[] message, Random r) throws NoSuchAlgorithmException,
InvalidKeyException, IllegalBlockSizeException, BadPaddingException, NoSuchPaddingException {
        myRSA rsa = new myRSA();
        byte[] k = HashFunction(message,256);
        byte[] v = new byte[1024/8];
        r.nextBytes(v);
        BigInteger[] x = new BigInteger[keys.length];
        BigInteger[] y = new BigInteger[keys.length];
        for(int i = 0; i<keys.length;i++){


                x[i] = new BigInteger(1024,r);
                y[i] = rsa.encrypt(x[i],keys[i][0],keys[i][1]); //(ei,Ni,Null)

        }
        byte[] cd = Combining_direct(k,v,y);
        byte[] gap = xor(v,cd);
        byte[][] result = new byte[keys.length + 2][];
          for (int i = 0; i < keys.length; i++) {
            byte[] X = x[i].toByteArray();
            if (X.length == 129) {
                X = trimLeading(X);
            }
            result[i] = X;
          }
        result[keys.length] = v;
        result[keys.length+1] = gap;
        return result;
```

```
    }


   public static void main (String[] args) throws NoSuchAlgorithmException, InvalidKeyException, IllegalBlockSizeException,
BadPaddingException, NoSuchPaddingException{

   Random r = new Random ();
   int n = 4;

    myRSA rsa = new myRSA();
    BigInteger[] result = new BigInteger[3];
    BigInteger[][] keys = new BigInteger[n][3];
    byte[] mensaje = new byte[32];
      r.nextBytes(mensaje);

    for (int i = 0; i<n;i++){


          result =  rsa.GeneratePUB();
          keys[i][0] = result[0];//ei
          keys[i][1] = result[1];//Ni
          keys[i][2] = result[2];//di

    }


    byte[][] sim = Simulate_Ring(keys,mensaje,r);
    System.out.print("This is the Simulation\n"+Arrays.deepToString(sim)+"\n");



}

}
```