Estudio e implementación de una autoridad de certificación distribuida

Memòria del projecte de final de carrera corresponent als estudis d'Enginyeria Superior en Informàtica presentat per Álvaro Merino de la Concepción i dirigit per Helena Rifà Pous.

Bellaterra, Juny de 2006

Resum

Aquest projecte presenta, en primer lloc, un estudi dels protocols de generació de claus criptogràfiques y autoritats de certificació distribuïdes més destacables fins al moment. Posteriorment, implementem un protocol tolerant a fallades de generació distribuïda de claus RSA, sense servidor de confiança, orientat a xarxes ad-hoc. El protocol necessita la participació conjunta de n nodes per generar un mòdul RSA $N = pq$, un exponent d'encriptació públic e y les particions d_i de l'exponent privat d, seguint un esquema llindar (t, n) on qualsevol subconjunt de $t \leq (n - 1/2)$ nodes no obté prou informació per poder factoritzar N.

Resumen

Este proyecto presenta, en primer lugar, un estudio de los protocolos de generación de claves criptográficas y autoridades de certificación distribuidas más destacables desarrollados hasta el momento. Posteriormente, implementamos un protocolo tolerante a fallos de generación distribuida de claves RSA, sin servidor de confianza, orientado a redes ad-hoc. El protocolo necesita la participación conjunta de n nodos para generar un módulo RSA $N = pq$, un exponente de encriptación público e y las particiones d_i del exponente privado d, siguiendo un esquema umbral (t, n) donde cualquier subconjunto de $t \leq (n - 1/2)$ nodos no obtiene información suficiente para poder factorizar N.

Abstract

In this project we introduce, first of all, a study of the most relevant distributed key generation protocols and certification authorities. Subsequently, we design a fault-tolerant distributed RSA-key generation protocol, without a trusted dealer, oriented to ad-hoc networks. In the protocol, a number n of parties jointly generate a RSA modulus $N = pq$, a public encryption exponent e and the shares, d_i, of a private exponent d, using a (t, n) threshold scheme where no subset of $t \leq ((n-1)/2)$ parties gather enough information to be able to factorize N.

El firmant, Helena Rifà Pous, professor del Departament d'Enginyeria de la Informació i de les Comunicacions de la Universitat Autònoma de Barcelona

CERTIFICA:

Que la present memòria ha sigut realitzada sota la seva direcció per Álvaro Merino de la Concepción

Bellaterra, Juny de 2006

Firmat: Helena Rifà Pous

“No one can build his security upon
the nobleness of another person.”

Willa Cather (1873 - 1947)

Índice general

1. Introducción **1**

1.1. Presentación 1

1.2. Objetivos 3

1.3. Estructura de la memoria 4

2. Estudio del estado del arte **5**

2.1. Redes ad-hoc y criptosistemas distribuidos 5

2.1.1. Seguridad en redes ad-hoc 5

2.1.2. Criptosistemas distribuidos 6

2.2. Criptosistemas RSA 9

2.2.1. Generación distribuida y eficiente de claves RSA . . . 9

2.2.2. Algoritmo de generación distribuida de claves RSA . . 12

2.2.3. Generación distribuida, robusta y eficiente de claves RSA 15

2.2.4. Aproximación al RSA proactivo umbral 18

2.2.5. RSA umbral completamente distribuido 19

2.2.6. Computación compartida y eficiente de productos de números primos seguros 21

2.2.7. Firmas umbral 23

2.2.8. Firmas umbral sin servidor de confianza 25

2.2.9. DICTATE 26

2.2.10. COCA 28
2.3. Criptosistemas basados en logaritmos discretos 31
2.3.1. Criptosistema umbral sin servidor de confianza 31
2.3.2. Generación de claves distribuida basada en logaritmos discretos 32
2.4. Criptosistemas basados en curvas elípticas 33
2.4.1. ECDKG 33
2.5. Conclusiones 36

3. Estudio de viabilidad de la aplicación 40
3.1. Objeto 40
3.1.1. Descripción de la situación a tratar 40
3.1.2. Perfil de usuario y entorno de integración 41
3.2. Análisis de requerimientos 42
3.2.1. Requerimientos funcionales 42
3.2.2. Requerimientos no funcionales 43
3.3. Descripción del sistema 45
3.3.1. Recursos 45
3.3.2. Evaluación de riesgos 45
3.4. Planificación del proyecto 48
3.4.1. Planificación de tareas 48
3.4.2. Diseño de la aplicación 50
3.5. Conclusiones sobre la viabilidad 54

4. Desarrollo de la aplicación 55
4.1. Descripción de la aplicación 55
4.2. Protocolo 60
4.2.1. Inicio y configuración del sistema 60
4.2.2. Computación distribuida de N 61
4.2.3. Test distribuido de doble primalidad 62
4.2.4. Computación distribuida de los exponentes público y privado 63
4.3. Clases 64
4.3.1. El generador de claves 64

4.3.2. Operaciones del protocolo 67
4.3.3. La interfaz de comunicaciones 68
4.3.4. Las nuevas estructuras de datos 69
4.3.5. Raíces cuadradas de números enteros grandes 71
4.4. Versiones alternativas 71
4.4.1. Implementación con threads 71

5. Pruebas y resultados 73
5.1. Análisis de resultados 73
5.1.1. Resultados de la implementación sin threads 73
5.1.2. Resultados de la implementación multithread 77
5.2. Algoritmos de mejora del rendimiento 78

6. Conclusiones 81
6.1. Repaso de objetivos 81
6.2. Formación académica 82
6.3. Líneas de continuación 83

Índice de figuras

2.1. *Estructura del protocolo DICTATE construido sobre PILOT (en gris)* 28
2.2. *Modelo de sistema de COCA* 30

3.1. *Planificación de tareas* 49
3.2. *Diagrama de Gannt* 49
3.3. *Diagrama de clases de la aplicación* 51
3.4. *Diagrama de actividades de la aplicación* 52
3.5. *Diagrama de secuencia de la aplicación* 53

CAPÍTULO 1

Introducción

1.1. Presentación

Los recientes avances de las tecnologías inalámbricas, así como la diversificación y extensión en el uso de los dispositivos móviles han hecho que se despierte un gran interés por las redes ad-hoc. No obstante, para que estas redes sean funcionales se requiere que dispongan de servicios de seguridad que certifiquen la autenticidad, integridad y confidencialidad de las comunicaciones. La mejor forma de ofrecer estos servicios es mediante los criptosistemas de clave pública basados una autoridad de certificación (CA), responsable de emitir y revocar los certificados digitales que establecen la relación entre la identidad de un usuario y su clave pública. Las redes ad-hoc no cuentan con ningún tipo de infraestructura, y son los propios nodos que forman la red los encargados de actuar como routers y reencaminar los paquetes hacia el destino. Por lo tanto, la seguridad en este tipo de redes es fundamental, ya que solamente para que los paquetes puedan viajar por ella, se necesitan mecanismos que aseguren el correcto reenvío de los paquetes por el camino que corresponde.

El proyecto desarrollado está dividido en dos partes. La primera consiste en el análisis de diversos sistemas distribuidos de computación de claves

asimétricas, así como de autoridades de certificación distribuidas. Los sistemas analizados los hemos dividido en tres grupos según estén implementados usando criptosistemas RSA, criptosistemas basados en logaritmos discretos o criptosistemas basados en curvas elípticas. De todos ellos, hemos analizado los protocolos que utilizan en su implementación, sus características técnicas, los modelos de red utilizados, los tipos de adversarios ante los que son eficaces y su rendimiento una vez implementados en entornos reales.

La segunda parte del proyecto consiste en el diseño de un sistema de computación de claves RSA distribuido, que será la base para la implementación de una autoridad de certificación distribuida. El protocolo desarrollado es completamente distribuido, es decir, elimina la necesidad de que participe un servidor de confianza que genere y distribuya las claves. Esto implica que un número n de nodos trabajen de forma conjunta para computar un módulo $N = pq$, donde p y q son dos números primos. Al final del proceso, todas las partes están convencidas de que N es el producto de dos números primos grandes, pero ninguna de ellas posee la información suficiente para poder factorizarlo. A partir de este módulo, los nodos participantes computan un exponente público e y obtienen una partición del exponente privado d, tal que $e * d = 1 \bmod \phi(N)$. Estas claves permiten realizar el proceso encriptado y desencriptado siguiendo un modelo de criptografía umbral, según el cual, dado un conjunto n de nodos se capacita a cualquier subconjunto t de ellos a generar la firma RSA de un mensaje dado. Según esto, ningún subconjunto formado por $t - 1$ nodos o menos, será capaz de computar las claves RSA.

Esta aplicación está orientada a ser implementada en entornos de redes ad-hoc. Éstas son redes espontáneas y autónomas (generalmente inalámbricas), que no dependen de una infraestructura. Los propios dispositivos son los que crean la red de área local (LAN) y no existe ningún controlador central ni puntos de acceso. Esta topología es práctica en lugares en los que pueden reunirse pequeños grupos de equipos que no necesitan acceso a otra red, en operaciones de emergencia, reuniones, convenciones, etc. Los dispositivos que conforman este tipo de redes pueden ser muy variados, incluyendo ordenadores, PDAs o teléfonos móviles. Por consiguiente, la aplicación desarrollada es independiente de la plataforma, lo que permite que sea instalada en una amplia variedad de dispositivos, y no precisa que dispongan de gran

capacidad de procesamiento, por lo que permite que dispositivos como las PDAs o los móviles puedan participar en el protocolo.

1.2. Objetivos

El objetivo que persigue este proyecto es principalmente el de proveer mecanismos de autentificación fiables y robustos en redes ad-hoc. Este tipo de redes, debido a sus características y a que no ha tenido un gran auge hasta hace unos pocos años, adolece de cierta falta de infraestructuras en lo referente a seguridad, y puede ser más sensible que el resto de redes a ciertos tipos de ataques. Tampoco disponen de puntos de acceso hacia otras redes, lo que impide que puedan acceder a las autoridades de certificación normales, por lo tanto, el hecho de que puedan crear una autoridad de certificación robusta y eficiente ellas mismas supone un punto muy importante dentro de la gestión de la seguridad dentro de estas redes.

De esta manera, mediante este proyecto se pretende crear la infraestructura necesaria para la implementación de una autoridad de certificación distribuida. Dicha infraestructura comprende tanto los aspectos teóricos, con la realización de un estudio del estado del arte; como técnicos, con el desarrollo de un sistema distribuido de computación de claves RSA.

En definitiva, los objetivos que persigue este proyecto son:

- Realizar un estudio del estado del arte. Analizando las soluciones existentes al problema de generar una autoridad de certificación distribuida.

- Diseñar e implementar una autoridad de certificación distribuida que permitirá emitir y revocar certificados para todos los usuarios de una red ad-hoc. Estos certificados serán la base de los posibles servicios de seguridad, aportando autenticidad, integridad y confidencialidad a las comunicaciones.

- Evaluar la viabilidad del proyecto y analizar los resultados obtenidos.

1.3. Estructura de la memoria

Esta memoria está organizada de la siguiente manera: en el capítulo 2 se ha realizado un estudio del estado del arte (autoridades de certificación distribuidas, sistemas de generación de claves, esquemas de firmas en modelos de criptografía umbral), que es el paso previo al desarrollo de nuestra aplicación. El capítulo 3 contiene un estudio de viabilidad de la aplicación que hemos implementado, en el que se analizan todas las características técnicas y requerimientos del sistema, llegando, finalmente, a unas conclusiones sobre su viabilidad técnica. El capítulo 4 contiene un análisis en profundidad de la aplicación desarrollada. Posteriormente, se realiza un análisis de los resultados obtenidos en el capítulo 5. Los dos últimos capítulos se corresponden con las conclusiones y la bibliografía. En el primero comentamos nuestras impresiones a la finalización del proyecto, así como las posibles líneas de continuación del proyecto; y en el segundo hacemos una referencia a todas las fuentes de información y consulta utilizadas.

CAPÍTULO 2

Estudio del estado del arte

2.1. Redes ad-hoc y criptosistemas distribuidos

2.1.1. Seguridad en redes ad-hoc

Las redes ad-hoc están formadas por conjuntos de nodos que se comunican entre ellos, generalmente, a través de enlaces inalámbricos. Estas redes carecen de una infraestructura dedicada, lo que hace que los protocolos de seguridad para este tipo de redes sean más difíciles de implementar. Además, las conexiones inalámbricas son más vulnerables a los ataques. Para proveer seguridad en estas redes se necesita una infraestructura de claves pública (PKI). Esta PKI es una combinación de elementos hardware, software, políticas y procedimientos que permiten asegurar la identidad de los participantes en un intercambio de datos usando criptografía de clave pública. La implantación de una PKI generalmente involucra la existencia de una Autoridad de Certificación (CA). Estas CA son entidades de confianza que certifican la autenticidad de la unión entre una clave pública y la identidad de un ente mediante la expedición de certificados. Los certificados son estructuras de datos firmadas por una autoridad de certificación, que sirven para asociar una clave pública con un usuario.

La implementación de un servidor centralizado que actúe de CA en una red ad-hoc es una tarea difícil debido al gran dinamismo de estas redes y a que surgen de forma espontánea. Además, las implementaciones distribuidas poseen una mayor disponibilidad y mejoran la robustez del sistema frente a posibles fallos del servidor. Sin embargo, todos estos beneficios se obtienen pagando un coste bastante alto en lo referente a complejidad del protocolo. Por un lado, los servidores deberán estar sincronizados para asegurar que tras una petición se devuelve el certificado correcto. Por otro lado, se necesita que las claves se vayan actualizando cada cierto intervalo de tiempo para hacer frente a adversarios activos, que modifican su comportamiento a lo largo del tiempo. Esto implica que no se puedan implementar sistemas asíncronos dado que la complejidad de implementar un sistema proactivo[1] sería demasiado alta.

2.1.2. Criptosistemas distribuidos

El concepto de criptografía distribuida fue introducido por primera vez por los trabajos de Desmedt [Des87] en 1987. El objetivo es el de diseñar sistemas criptográficos en los que las operaciones requieran la colaboración de varios usuarios. Más concretamente, se podría definir como un criptosistema de clave pública en el que la clave secreta está distribuida entre un conjunto de usuarios.

Hasta el momento han aparecido diversos criptosistemas distribuidos, la mayoría de los cuales presenta una estructura de criptosistema umbral. Este tipo de estructura permite que subconjuntos de t servidores de los n que participan sean capaces de llevar a cabo la acción requerida. De esta forma la seguridad, eficiencia y robustez del sistema se ven incrementadas, debido a que el sistema es tolerante a fallos y a diversos ataques por parte de agentes maliciosos.

El concepto de compartición de secretos [Sha79] forma la base de la criptografía umbral, de tal manera que un secreto s se comparte entre n nodos de los cuales se necesita la participación de al menos $(t+1)$ nodos para

[1]Se adapta dinámicamente a los adversarios. La ejecución se divide en intervalos de tiempo. Tolera adversarios dinámicos, que se ven limitados a corromper los hosts dentro de estos intervalos.

poder recuperar el secreto s. Posteriormente, aparecieron los esquemas de compartición verificable de secretos (Verifiable Secret Sharing - VSS). Estos son protocolos tolerantes a fallos que permiten asegurar que la información distribuida por un servidor P es consistente y define un único secreto.

Todos estos conceptos dieron lugar a la aparición de sistemas de generación de claves de forma distribuida, que generan una clave pública y distribuyen particiones de la clave secreta. Estos sistemas deben asegurar que los nodos (probablemente corruptos) no obtienen ninguna información sobre la clave privada (sistemas de conocimiento cero). Hasta el momento han aparecido diversos sistemas implementados bajo los tipos de sistemas criptográficos más comunes, como son los de logaritmos discretos, RSA y curvas elípticas.

Como se ha comentado, en 1987 Desmedt introdujo el concepto de criptografía umbral. Dos años más tarde, Desmedt y Frankel [DF90] presentaron un esquema El Gamal de criptografía umbral no robusto basado en el esquema de compartición de secretos de Shamir [Sha79] y en la interpolación polinomial sobre un campo finito. Este esquema usaba particiones de claves de tamaño pequeño y necesitaba una interacción sincronizada. Pedersen [Ped91] definió en 1991 el primer sistema de compartición de claves basadas en logaritmos discretos. En 1994, Harn presentó el primer criptosistema El Gamal robusto [Har94], y en 1996, Gennaro et al. [GJKR96] presentaron un estándar de firmas digitales (DSS) robusto. Estos dos últimos protocolos presentaban las mismas características que el de Desmedt y Frankel en lo referente al tamaño de las particiones y a la interacción sincronizada.

También en el año 1989, Desmedt y Frankel hicieron referencia al problema de los criptosistemas RSA umbral en [DF90]. En este trabajo describieron la problemática de trabajar con criptosistemas RSA, dado que la interpolación polinomial sobre un anillo de coeficientes $\mathbb{Z}_{\phi(N)}$, donde N es el módulo RSA y ϕ la función totient de Euler, es bastante complicada. Dos años después, presentaron el primer esquema no robusto y sin análisis de seguridad [DF91]. El avance más significativo no se produjo hasta el año 1997, en el que Boneh y Franklin [BF97] mostraron cómo un grupo de participantes podía generar una función RSA de forma eficiente, demostrando además que el protocolo era seguro en el modelo de nodos "honestos, pero curiosos", pero

dejando abierto el problema de la generación de claves en presencia de nodos maliciosos. Este problema fue solucionado posteriormente por Frankel et al. [FMY98] que demostraron cómo hacer un protocolo que fuese eficiente y además, robusto.

Otro esquema de gran interés fue el propuesto por Shoup [Sho00] el año 2000. Este no era un esquema completamente distribuido, pues utilizaba un servidor de confianza para generar las claves y distribuir las particiones entre los nodos, pero presentaba un esquema seguro y robusto en el modelo de oráculo aleatorio. Trabajos posteriores, como el de Fouque y Stern [FS01], demostraron cómo se podía implementar el esquema de Shoup utilizando el sistema de generación de claves de Boneh y Franklin, de tal forma que el sistema no necesitara un servidor centralizado.

Los primeros métodos para diseñar criptosistemas distribuidos de curvas elípticas basados en logaritmos discretos no fueron muy exitosos y no consiguieron una complejidad subexponencial, excepto en casos muy específicos. En el año 2002, Tang [Tan02] propuso un nuevo protocolo de distribución de claves que tenía como bloque principal un criptosistema basado en curvas elípticas. Este protocolo disfruta de las ventajas que tienen otros protocolos de distribución de claves no basados en curvas elípticas, añadiendo las ventajas de la criptografía de curvas elípticas (ECC), como son flexibilidad, generación eficiente de claves, menores necesidades de almacenamiento y menor congestión de la red.

En los siguientes apartados realizamos un análisis de los criptosistemas distribuidos más importantes de los tipos más comunes de sistemas criptográficos: basados en logaritmos discretos, RSA y basados en curvas elípticas. Este análisis sentará la base teórica para la posterior implementación de un sistema de generación de claves distribuido, base del desarrollo de una autoridad de certificación distribuida para redes ad-hoc. En el siguiente apartado veremos las conclusiones obtenidas tras este estudio y decidiremos cómo vamos a afrontar la implementación de nuestra aplicación.

2.2. Criptosistemas RSA

2.2.1. Generación distribuida y eficiente de claves RSA

Introducción

Uno de los protocolos más importantes que hemos estudiado es el de [BF97], que se describe a continuación, y que ha servido de base para el desarrollo de otros protocolos, como por ejemplo [MWB99], [FMY98], [Rab98] y [FS01]. Este protocolo describe técnicas eficientes para la generación de una clave RSA entre varios hosts. El objetivo es computar un módulo RSA $N = pq$. Todos los servidores participantes en el protocolo estarán convencidos de que N es el producto de dos primos grandes. Este algoritmo generará también un exponente de encriptación público (e) y otro privado (d), del que cada uno de los servidores poseerá una parte.

La implementación de estas claves se realiza de forma completamente distribuida, es decir, sin la necesidad de un servidor central de confianza. Se sigue un modelo *honest but curious parties* (todos los hosts siguen honestamente el protocolo, pero ninguno es capaz de aprender nada). El protocolo está diseñado siguiendo un esquema de criptografía umbral, en el que hay k servidores participando y se necesita la colaboración de un conjunto t de ellos para generar una firma RSA (por lo que el modelo es $t-1$ seguro).

El modelo de comunicación en el que se basa este protocolo presenta las siguientes características:

conectividad total: cada uno de los nodos se puede comunicar con todos los demás. Los nodos deben de estar comunicados mediante un medio de broadcast común, y se deben permitir también las conexiones extremo a extremo entre todos los nodos.

canales privados y autentificados: los mensajes enviados desde un nodo A a otro nodo B son privados y no pueden ser manipulados en el camino, dado que ambas partes comparten una clave secreta que usan para encriptar los mensajes y asegurar la integridad del mensaje y su autenticidad. Para ello, todas las comunicaciones se encriptan usando el protocolo SSL.

nodos honestos pero curiosos: se asume que todas las partes siguen correctamente el protocolo. Sin embargo, al final del proceso ningún subconjunto de nodos por debajo del umbral t llega a poseer suficiente información para factorizar el módulo N.

complicidad: el protocolo es $\lfloor \frac{k-1}{2} \rfloor$ privado. Esto implica que una coalición de como máximo $\lfloor \frac{k-1}{2} \rfloor$ nodos son incapaces de computar $N = pq$, aunque una coalición superior sí que lo sería. Esto se debe a que parte de los pasos del algoritmo están basados en el protocolo de Ben-Or, Goldwasser y Widgerson (BGW) [BOGW88]. El protocolo BGW provee seguridad basándose en la teoría de la información [Sha48] y por lo tanto, está limitado a conseguir una privacidad de $\lfloor \frac{k-1}{2} \rfloor$. También, han aparecido protocolos que son $k-1$ privados, como los de Poupard y Stern [PS98], y el de Gilboa [Gil99], pero se ha demostrado que son menos eficientes.

Algoritmo

El objetivo del algoritmo es generar una clave RSA compartida, por lo que se necesitará generar un módulo $N = pq$, y una pareja de exponentes público (e) y privado (d), donde $ed = 1 \mod \phi(N)$. Al final de la computación N y e serán públicos y d estará compartido entre todos los nodos.

El protocolo se divide en 4 fases:

1. selección de candidatos: cada nodo i escoge un par de números enteros de n bits, p_i y q_i. El módulo $N = pq$ resultante deberá ser un entero de Blum, es decir, los factores p y q deberán ser congruentes a $3 \mod 4$. Esto se consigue haciendo que el nodo número 1 compute los factores de tal forma que $p_1 = q_1 \equiv 3 \mod 4$, mientras que el resto de nodos computan $p_i = q_i \equiv 0 \mod 4$. Posteriormente, los nodos comprueban que $p = p1 + \cdots + pk$ (y su equivalente con q_i) no es divisible por un número l menor que un número frontera $B1$, usando un protocolo privado y distribuido. Este proceso nos asegura que los factores l seleccionados que no sean válidos siempre serán rechazados. Sin embargo, puede ocurrir que se rechace un candidato válido. Para disminuir el número de

falsos negativos se repite el test dos veces, por lo que la probabilidad de que un candidato l correcto sea rechazado es de $1 - \prod_{l<B}(1 - \frac{1}{l^2}) < \frac{l}{2}$. Si falla este proceso, se vuelve a escoger otro candidato y a realizar las comprobaciones.

2. computar N de forma distribuida: $N = (p_1 + \cdots + p_k)(q_1 + \cdots + q_k)$. Para este paso se utiliza un algoritmo basado en el protocolo BGW descrito en [BOGW88], en el que han de participar un mínimo de 3 nodos. Al final del proceso todas las partes conocen N, sin que haya ninguna coalición de $\lfloor \frac{k-1}{2} \rfloor$ nodos haya aprendido ninguna información sobre los factores privados.

3. test de biprimalidad: los k nodos comprueban de forma distribuida que N es el producto de dos números primos. Durante este proceso no han de revelar ninguna información sobre la factorización de N. Si el test falla se vuelve a comenzar desde el paso 1.

4. generación de claves: se generan las particiones de $d = e^{-1} \text{ mód } \phi(N)$ dado un exponente de encriptación e. El exponente de desencriptación d será igual a $\sum_{i=1}^{n} d_i$. Para ello se describen dos protocolos: el primero trabajaría tan sólo con exponentes públicos de tamaño pequeño ($e < 1000$), pero es muy eficiente y la cantidad de información transmitida por la red es relativamente baja. El segundo trabajaría con cualquier exponente público. También es eficiente, pero requiere de una mayor comunicación entre los nodos.

Este protocolo puede implementar de forma complementaria una serie de optimizaciones (balances de cargas, filtrado de Fermat, divisiones de prueba en paralelo, etc.) que mejoran el rendimiento global de la implementación.

Tiempos de ejecución

El tiempo necesario para generar una clave RSA de 1024 bits de forma distribuida entre 3 servidores Pentium a 300 MHz es de 90 segundos. El tráfico de red total durante el proceso es de $1'16$MB. La comunicación se realiza utilizando SSL para encriptar los datos. Se pueden observar los resultados más detalladamente en la tabla 2.1 (pág. 14).

2.2.2. Algoritmo de generación distribuida de claves RSA

Introducción

Este algoritmo, descrito en [MWB99] es una implementación ligeramente modificada del protocolo visto anteriormente de Boneh et al. [BF97]. El objetivo de este algoritmo es el de generar de forma distribuida una clave RSA compartida. De tal forma que al final de la computación obtendremos $N = p * q$ que será público. Todos los servidores que participen en la computación de N acabarán convencidos de que N es el producto de dos números primos grandes, pero ninguno de ellos conocerá la factorización de N. Además, también se hará público un exponente de encriptación e y cada uno de los servidores participantes en el protocolo poseerá un trozo del exponente de desencriptación d.

Este protocolo está basado en el de Boneh, por lo que no será necesaria la existencia de un servidor de confianza, que controle todo el proceso. El algoritmo utilizado no es robusto, pero puede ser fácilmente modificado para que presente dicha propiedad.

Algoritmo

Este protocolo capacita a k servidores para que generen $N = p * q$ y los exponentes e y d; siendo p y q dos números primos grandes. La clave e será pública, mientras que d estará compartida entre todos los hosts, de tal forma que permita la generación de d con la participación de t de k hosts (esquema de criptografía umbral). Cualquier ataque sobre estos hosts para obtener el exponente d será infructuoso mientras se penetren menos de t de k hosts.

Los pasos a seguir para implementar este algoritmo son los siguientes:

1. Escoger los candidatos: Cada servidor k genera dos números aleatorios de n bits, p_i y q_i, y los guarda en secreto.

2. Computar N: El algoritmo para computar N es una adaptación del protocolo de evaluación de circuitos seguros de Ben-Or, Goldwasser y Widgerson [BOGW88]. El objetivo es computar $N = \sum p_i \sum q_i$ sin revelar ninguna información sobre los factores que componen N.

3. Test de primalidad: Los k servidores comprueban de forma distribuida que N es el producto de dos números primos. Para ello se realiza el test de Fermat, que comprueba $g^{N-p-q+1} \stackrel{?}{=} 1 \pmod{N}$. A diferencia de Boneh, este protocolo no realiza el test de doble primalidad. El test anterior tiene el problema de que existe una serie de valores de N que no son números primos que pueden pasar el test. La probabilidad de que esto ocurra es menor de 1 entre 10^{40}, por lo que no considera necesario aplicar un test más robusto, pero mucho más lento. Si el test falla, el algoritmo vuelve al primer punto.

4. Generación de claves: Dado un exponente público e, los servidores generan distribuidamente un exponente de desencriptación $d = e^{-1} \bmod \phi(N)$. Al final de la computación cada host debe tener un d_i tal que $d = \sum d_i$. Existen dos posibles técnicas para implementar un algoritmo que resuelva el problema anterior. La primera funciona cuando e es pequeño (aproximadamente $e < 2^{20}$). Este algoritmo es muy eficiente y requiere muy poca comunicación entre servidores. La segunda técnica funciona con cualquier valor de e, pero la cantidad de comunicación necesaria es muy superior a la de la primera. Como la generación de firmas digitales generalmente usa valores de e pequeños, sería más apropiado utilizar el primero de los métodos para que la verificación de la firma sea más rápida. Para proveer a esta implementación de tolerancia a fallos, lo que haremos será crear una compartición entre t de k servidores de la clave privada, de tal forma que cada subconjunto de t servidores sean capaces de generar dicha clave.

Implementación

Algunas de las características de la implementación son:

- La comunicación se realiza utilizando un protocolo de transporte seguro, como Secure Socket Layer (SSL). Más concretamente se utiliza el paquete SSLeay de Eric A. Young.

- Para aumentar la eficiencia se aplican técnicas de buffering y de E/S no bloqueante.

- Se utilizan threads para establecer comunicaciones en paralelo.
- Se provee una interfície intuitiva e independiente de la plataforma para leer y escribir tipos abstractos de datos.
- Se gestiona la autentificación extremo a extremo de forma transparente, mediante certificados y números de secuencia.

En esta implementación se pueden aplicar varias mejoras para incrementar la seguridad y para reducir los tiempos de ejecución (como por ejemplo, computación paralela, balance de cargas o técnicas más robustas de generación de claves). Los parámetros de configuración del sistema se guardan en un archivo, que contiene, entre otros, las direcciones IP y los puertos de conexión de todos los nodos, así como la localización de su certificado y de la clave privada. Todos los nodos utilizan el mismo archivo de configuración para reducir así el *overhead* administrativo.

Tiempos de ejecución

La ejecución más óptima tarda una media de 91 segundos en generar una clave RSA de 1024 bits entre 3 servidores utilizando máquinas Pentium II a 333 MHz ejecutando Solaris 2.5.1. Los servidores están conectados mediante una ethernet a 10 Mbps. Durante la ejecución del algoritmo cada servidor envía aproximadamente $1'2$ MB de datos.

En la tabla 2.1 observamos los tiempos para la generación de claves entre 3 servidores. Y en la tabla 2.2 podemos observar los tiempos de generación de claves cuando variamos el número de servidores y su localización.

tamaño claves	sieving time	tiempo BGW	Trial div. time	test primalidad		iteraciones		tiempo total	tráfico de red	Número de threads
				tiempo	núm	tiempo	núm			
512b	39'9	55'7	2'4	37'6	36'4	77'2	119	0'15 min	0'18 Mb	2
1024b	362'4	82'6	2'3	637'2	49'1	696'9	130	1'51 min	1'16 Mb	6
2048b	369'3	470'9	3'6	4177'3	233'9	2198'0	495	18'13 m.	7'48 Mb	6

Tabla 2.1: *Tiempo de generación de claves entre 3 servidores*

	SSL	Sieve time	tiempo BGW	trial div. time	test primalidad	tiempo iteración	tiempo total
3 serv.	no	326	413	2'2	628	695	1'51 min.
3 serv.	sí	362	83	2'3	637	697	1'51 min.
4 serv.	no	689	861	1'5	2035	1707	3'70 min.
4 serv.	sí	804	1017	1'9	2173	1909	4'14 min.
5 serv.	sí	1466	1731	1'9	2013	2589	5'61 min.
WAN	sí	774	1012	6'3	1626	1704	5'69 min.
Local	no	263	334	2'7	1702	1014	2'20 min.
Local	sí	267	337	3'3	1899	1115	2'42 min.

Tabla 2.2: *Efecto de cambiar el número de servidores y su localización*

2.2.3. Generación distribuida, robusta y eficiente de claves RSA

Introducción

El protocolo descrito en [FMY98] es probablemente uno de los más completos de los que hemos analizado en este estudio. Basado en el protocolo descrito en [BF97], este algoritmo resuelve el problema de la generación distribuida, robusta y eficiente de claves RSA. Al igual que los protocolos anteriores provee un esquema de computación, sin servidor central, de criptografía umbral ($n > 2t + 1$), donde puede haber hasta t partes corruptas participando en el protocolo. También se puede implementar un esquema $n > 3t + 1$, más seguro que el anterior. Otra de las mejoras que implementa es que el sistema es proactivo, dado que se realizan realeatorizaciones de los trozos compartidos. Es decir, el tiempo se divide en intervalos, y en cada uno de estos intervalos se generan claves nuevas. De esta forma se invalida toda la información que hubiese podido obtener un adversario.

La implementación del protocolo se ha realizado basándose en las siguientes técnicas:

- protocolos de multiplicación para la representación compartida *sum-of-poly* de valores de un campo primo (donde la robustez y la seguridad se basan en la complejidad del problema del logaritmo discreto) o de un subconjunto de enteros (donde la robustez y la seguridad se basan en la complejidad del problema RSA). Esta técnica transforma una función compartida $t - de - t$ en una distribución de polinomios $t - de - n$.

- técnicas de consistencia encadenada de información compartida, entre las que se incluyen técnicas de *cross-checking*, que mantienen la consistencia a base contrastar la información transmitida con unos verificadores; y técnicas de conocimiento cero, que permiten verificar que la información es consistente. Para este último punto se usa la prueba de conocimiento de un logaritmo discreto, por la que un probador $\mathcal{P}$ demuestra a un verificador $\mathcal{V}$ que conoce $X = g^x$ (algoritmo presentado en [GHY86]).

- un mecanismo de *commitment*, llamado *Simulator-Equivocal Commitments*. Un esquema de *commitment* permite que un emisor (Em) envíe un mensaje a un receptor (Re) sin que éste lo pueda leer, hasta que Em le dé la clave que le permita abrirlo. *Simulator-Equivocal Commitments* es un tipo especial de *commitment* llamado *trapdoor commitment*. Este tipo de *commitments* tienen la propiedad de que si se conoce una información especial, "trampilla", se puede abrir el mensaje y leer el contenido. Por lo tanto, el mecanismo de *commitment* implementado por Frankel es básicamente un *trapdoor commitment* combinado con una prueba de conocimiento de la "trampilla". Puede ser simulado por un simulador de tiempo polinomial.

El algoritmo resultante será eficiente (independiente del tamaño del circuito de tests de primalidad) y robusto (asegura que acabará correctamente incluso ante la presencia de máquinas maliciosas o que se comporten de forma inadecuada).

El modelo de red se basa en un grupo de n servidores conectados a un medio de broadcast común. Asume que el sistema está sincronizado, y que los mensajes enviados llegan instantáneamente a todos los nodos receptores. No se especifica ningún tipo de algoritmo de encriptación para realizar la comunicación a través de canales seguros y auténticos.

Algoritmo

El objetivo del algoritmo es generar una clave RSA compartida. Necesitará generar un módulo $N = pq$, y una pareja de exponentes público (e)

y privado (d). Al final de la computación N y e serán públicos y d estará compartido entre todos los nodos.

Los pasos que sigue el algoritmo son:

1. Arrancar el protocolo de inicialización de los *Simulator-Equivocal Commitments*. El mecanismo de *commitments* cuenta con un protocolo de inicialización que sólo se ha ejecutar una vez. En este proceso se elige un número primo seguro P y un generador g para $\mathbb{Z}_P^*$ de forma distribuida. Posteriormente, cada nodo envía a los demás un valor g' y demuestra que conoce el logaritmo discreto de g' base g.

2. El paso anterior permitirá a los nodos acordar conjuntamente qué servidores son "honestos" (eliminando a los corruptos del protocolo) y cual es el límite de servidores corruptos ($t = n + 1/2$, o bien, $t = n + 1/3$), que soportará el algoritmo.

3. Ejecutar la computación distribuida de N. Cada nodo elige los factores $p_i, q_i \in [\frac{1}{2}\sqrt{H}, \sqrt{H}]$, que posteriormente se distribuyen al resto de nodos utilizando el esquema de compartición de secretos de Shamir [Sha79], el esquema de compartición verificable de secretos (VSS) e incondicionalmente seguro de Pedersen [Ped92], mecanismos de transformación de la representación y mecanismos de multiplicación sobre los enteros.

4. Ejecutar el test de doble primalidad de N robusto y distribuido. Se usa el esquema de test de doble primalidad de Boneh-Franklin junto con mecanismos de *commitment* para hacerlo robusto. Si el test falla, el protocolo se reinicia contando tan sólo con la participación de los nodos declarados honestos.

5. Ejecutar la generación robusta y distribuida de las claves pública y privada. Para este proceso se puede utilizar cualquiera de los dos métodos (uno para claves públicas pequeñas y otro para claves públicas arbitrarias) vistos en la página 11.

El sistema cuenta con un parámetro de seguridad h que ha de estar comprendido entre 1024 y 2048. El tamaño de las claves no debe ser inferior a

400. El número de nodos participantes debe estar comprendido entre 3 y 10. La complejidad del algoritmo está dominada por el protocolo de multiplicaciones y está entorno a $24n(t+1)$.

2.2.4. Aproximación al RSA proactivo umbral

Introducción

El protocolo analizado a continuación se describe en [Rab98]. El objetivo de esta implementación es el de generar de forma distribuida las claves RSA pública y privada. Además, esta implementación también describe el proceso de generación de firmas digitales. Al igual que el protocolo de Frankel et al. [FMY98], es robusto y eficiente. La clave privada está en todo momento compartida de forma aditiva, lo que permite realizar firmas de forma simple y actualizar eficientemente las claves de forma proactiva. Se necesita un reloj global común para realizar este proceso de proactivación. El tiempo de vida del sistema se divide en periodos de tiempo en los cuales se asume que hay menos de $t \leq (n-1)/2$ servidores corruptos. La existencia y disponibilidad de una clave a lo largo del tiempo de vida del sistema está garantizada sin ninguna probabilidad de error.

Este esquema asume la existencia de un servidor de confianza, aunque se puede implementar el mecanismo de generación de claves de forma distribuida de Boneh y Franklin [BF97].

El modelo de comunicación asume la existencia de n hosts que pueden ser modelados por máquinas de Turing de tiempo polinomial aleatorio. Los hosts estarán conectados extremo a extremo por una red de canales privados, a la vez que dispondrán también de un canal de broadcast dedicado.

Algoritmo

Como se ha comentado anteriormente, este protocolo asume la existencia de un servidor de confianza, que puede ser sustituido por el esquema de Boneh y Franklin de forma que la generación de claves se realice de forma completamente distribuida. Si utilizamos un servidor de confianza, el protocolo se basará en distribuir de forma segura las claves generadas por el servidor. La clave privada quedará distribuida de forma aditiva (la clave d

será el sumatorio de todos los trozos d_i distribuidos) entre los hosts. A continuación, especificaremos cómo se realiza la distribución de claves en el caso de que se utilice un servidor central:

1. El servidor central genera un par de claves RSA de forma unilateral, y particiona la clave privada d de tal forma que $d = \sum_{i=1}^{n} d_i$. Además, añade una partición pública d_{public}, tal que $d = d_{public} + \sum_{i=1}^{n} d_i$.

2. Se distribuyen las particiones y se ejecuta un mecanismo de back-up de las particiones mediante un protocolo de compartición verificable de secretos (VSS) basado en el protocolo VSS de Feldman [Fel87]. El protocolo VSS sobre $\mathbb{Z}_n$, que implementa pruebas de conocimiento cero, será el que haga que el protocolo sea robusto.

3. El servidor genera una "firma testigo", en la forma $g^{d_i} \bmod N$, para verificar las firmas parciales generadas bajo cada una de las particiones d_i.

Una vez completado el proceso de generación de claves, se computa la firma de la siguiente forma:

1. Cada host P_i publica una firma parcial $\sigma_i = m^{d_i} \bmod N$.

2. P_i prueba mediante el protocolo de Gennaro et al. [GJKR96] que $DL_m \sigma_i = DL_g \omega_i$.

3. Si la prueba falla, se reconstruye d_i mediante una fase de reconstrucción basada en el protocolo VSS de Feldman.

4. Como tenemos que $d = d_{public} + \sum_{i=1}^{n} d_i$, la firma digital será $SIG(m) = m^d = m^{d_{public} + \sum_{i=1}^{n} d_i} = m^{d_{public}} \prod_{i=1}^{n} \sigma_i \bmod N$.

2.2.5. RSA umbral completamente distribuido

Introducción

El objetivo de esta implementación, descrita en [FS01], es la de crear un esquema de generación de claves y firmas RSA completamente distribuido. La generación de claves se realiza mediante una adaptación del esquema de

Boneh y Franklin [BF97], aunque parte del protocolo se basa en el esquema de criptografía umbral para firmas RSA de Shoup [Sho00]. La implementación de mejoras sobre estos protocolos empobrece el rendimiento final, pero se consiguen importantes mejoras en seguridad para aplicaciones de gran sensibilidad. El resultado final es un protocolo completamente distribuido, robusto y seguro ante adversarios estáticos, activos y pasivos.

El modelo de comunicaciones asume un conjunto ℓ de servidores comunicados entre sí mediante un medio de broadcast común. El adversario puede corromper un servidor en cualquier momento viendo las memorias de servidores corruptos (adversario pasivo) y/o modificando su comportamiento (adversario activo). El adversario decide los servidores a corromper al inicio del protocolo (adversario estático). Se asume que el adversario no será capaz de corromper más de t de ℓ servidores, donde $\ell \geq 2t + 1$.

Algoritmo

El objetivo del algoritmo es crear un módulo RSA tal que el grupo de cuadrados Q_N en $\mathbb{Z}_N^*$ sea un grupo cíclico cuyo orden no tenga factores primos pequeños. El resultado esperado es $N = pq$, donde $p = 2p' + 1$ y $q = 2q' + 1$, siendo $mcd(p', q') = 1$ y no hay ningún número primo que divida ni p' ni q'. Para computar este módulo, de tal forma que el grupo Q_N sea cíclico, se utiliza el protocolo de generación de claves de Boneh y Franklin junto con un algoritmo capaz de computar el máximo común divisor de un valor público y un secreto compartido (de ahora en adelante lo denominaremos protocolo MCD). La adaptación realizada del esquema de Boneh y Franklin es la siguiente:

1. Los servidores escogen $p_i, q_i \in [\lfloor 2^{(n-1)/2} \rfloor, \lfloor \frac{2^{n/2}-1}{\ell} \rfloor]$, donde n es el tamaño de N. Mediante el algoritmo de *sieving* de Malkin et al. [MWB99] se eliminan todos los valores que tengan factores primos pequeños. A continuación, se comprueba si $mcd(p-1, 4P) = 2$ y si $mcd(4P, q-1) = 2$, siendo $P = \prod_{2<p_i<B} p_i$, y B el límite de *sieving*.

2. El siguiente paso consiste en utilizar el algoritmo BGW para computar $N = \sum p_i \sum qi$ y $\phi(N) = (p-1)(q-1)$.

3. Finalmente, cada uno de los nodos ejecuta un test de primalidad similar al test de Fermat módulo N.

El protocolo MCD se basa en el algoritmo de Catalano et al. [CGH00]. El algoritmo se basa en el hecho de que $mcd(e, \phi) = mcd(e, \phi + Re)$, donde R es un número entero grande usado para enmascarar el secreto $\phi = \phi_1 + \cdots + \phi_\ell$. Por lo tanto, cada servidor i escoge de forma aleatoria un entero $r_i \in_{\mathcal{R}} [0..2^{n+k'}]$, donde k' es el parámetro de seguridad, computa $c_i = \phi_i + er_i$ y lo envía al resto de nodos. Posteriormente, cada nodo obtiene $c = \sum c_i = \phi + eR$, donde $R = \sum r_i$. Finalmente, los nodos pueden computar $mcd(e, c)$, que es equivalente a $mcd(e, \phi)$, así como u y v, tales que $eu + cv = 1$ cuando $mcd(e, \phi) = 1$.

A partir de los valores obtenidos en el protocolo de Catalano et al. se pueden obtener las particiones d_i del exponente privado d. Si reemplazamos c por $\phi + eR$ obtenemos que $e(u + Rv) = 1$. De esta forma obtenemos que $u + Rv$ es igual al inverso de e módulo ϕ, que es exactamente el valor del exponente privado buscado. El nodo 1 computa su partición $d_1 = u + vr_1$ y el resto de servidores la computa $d_i = vr_i$.

Para la firma de un mensaje m se utiliza el protocolo de Shoup con ciertas mejoras para hacerlo robusto contra adversarios activos.

2.2.6. Computación compartida y eficiente de productos de números primos seguros

Introducción

El objetivo de este protocolo es la computación distribuida de claves RSA, donde el módulo N es el producto de dos números primos seguros (*safe primes*), más eficiente que los métodos anteriormente conocidos. Un número primo seguro es un número primo de la forma $2p + 1$, donde p es también un número primo (número primo Sophie Germain).

Una de las principales diferencias de este algoritmo, descrito en [ACS02], con los anteriores es la forma en la que se generan las claves compartidas. Los protocolos anteriores utilizan implementaciones basadas en el esquema BGW (BenOr, Goldwasser y Widgerson), mientras que este nuevo esquema utiliza una nueva implementación más eficiente basada en números primos

seguros (aunque se puede implementar sin usar números primos seguros para obtener un mayor rendimiento). El modelo en el que se basa, al igual que el de Boneh et al., es el de *honest but curious parties*, en un esquema de criptografía umbral.

Este protocolo incorpora ciertas diferencias con los trabajos estudiados previamente. De ellas cabría destacar dos: primero, a diferencia de otros protocolos está basado en la multiplicación distribuida sobre un campo primo en vez de sobre los enteros, como por ejemplo el de Frankel et al. [FMY98]. No obstante, puede hacerse robusta usando las técnicas tradicionales de compartición verificable de secretos módulo un número primo. Y segundo, está basado en esquemas de compartición de secretos $k - de - k$. Esto provoca que se haya de asumir que ninguno de los nodos participantes abandone el protocolo antes de su finalización. De todos modos, los esquemas $k - de - k$ se pueden transformar fácilmente en esquemas $(\tau+1) - de - k$, donde $\tau = \lfloor \frac{k-1}{2} \rfloor$, mediante el método de Rabin [Rab98].

En lo concerniente al modelo de comunicaciones implementado, nos encontramos con que existen k hosts que se comunican mediante canales seguros y autentificados. El protocolo es seguro contra adversarios estáticos y *honest-but-curious*, que controlen hasta τ hosts. De todos modos, es posible forzar a los adversarios a que se comporten de forma honesta mediante mecanismos como pruebas de conocimiento cero. La seguridad de este modelo se ha comprobado siguiendo el esquema de Canetti [Can00].

Algoritmo

El bloque principal del algoritmo es un protocolo eficiente para reducir un secreto módulo p. Todos los cálculos se realizan utilizando comparticiones de un número primo Q, cuyo tamaño es aproximadamente el doble de p. Este algoritmo se basa en la idea de reducción modular de Aho et al. [AHU74], por lo que el problema se reduce a computar de forma distribuida $\lceil \frac{c}{p} \rfloor$, donde $c \equiv ab \pmod{p}$. Para afrontar este problema se computa de forma distribuida una aproximación de punto flotante de $1/p$ y posteriormente se computa $\lceil \frac{c}{p} \rfloor$ usando las aproximaciones anteriores. Una aproximación a $1/p$ se computa

fácilmente usando la iteración de Newton[2].

Se ha demostrado que el rendimiento de este esquema es mejor que el de Boneh, para un número pequeño de hosts, mientras que para el resto es relativamente parecido.

2.2.7. Firmas umbral

Introducción

En este apartado analizamos otro de los protocolos que han sentado un precedente en posteriores trabajos. Se halla descrito en [Sho00], y define un nuevo protocolo de generación de firmas digitales, mediante un servidor de confianza, basado en un esquema de criptografía umbral RSA. Es un protocolo simple, pero a la vez robusto. Para el correcto funcionamiento de este protocolo se imponen varias restricciones como son que el exponente público debe ser un número primo que exceda de l (donde l es el número de servidores) y que el módulo debe ser producto de dos números primos fuertes. Un número primo p es fuerte cuando cumple:

- $p = 1 \pmod{r}$
- $p = s - 1 \pmod{s}$
- $r = 1 \pmod{t}$

donde r, s y t son números primos grandes de tamaño n. Según Gordon [Gor85], tan sólo se requiere un 19 % más de computación para generar un número primo fuerte.

El resultado es un protocolo robusto e infalsificable (ningún adversario será capaz de crear una firma válida en un mensaje enviado a cualquiera de los t servidores no corruptos) dentro del modelo de oráculo aleatorio (este modelo sustituye las funciones criptográficas de hash por un oráculo aleatorio). La generación de particiones de la firma y su verificación serán procesos completamente no interactivos. El tamaño de una partición de una

[2]Algoritmo usado para computar $\sqrt{n}$ mediante la ecuación recurrente $x_{k+1} = \frac{1}{2}(x_k + \frac{n}{x_k})$, donde $x_0 = 1$.

firma estará limitado por un número constante de veces el tamaño del módulo RSA.

Esta implementación trabaja sobre un modelo de corrupción estático, donde el adversario debe elegir los servidores a corromper antes del inicio del ataque. Para proveer unas mejores garantías de seguridad se trabaja sobre un esquema umbral $k - de - l$ de parámetro dual [CKS00] que provee mayores garantías de seguridad que los esquemas de parámetro único.

Modelo de sistema

El sistema estará formado por un conjunto l de hosts y un servidor de confianza. Existirá un algoritmo de verificación de firmas, un algoritmo de verificación de particiones de las claves y un algoritmo para combinar dichas particiones.

Dividiremos el protocolo en dos partes. En la *dealing phase* el servidor de confianza genera una clave pública PK junto con las particiones de la clave privada $SK_1, \ldots, SK_l$ y las claves de verificación $VK_1, \ldots, VK_l$. El adversario obtiene las particiones de las claves secretas de los hosts que ha corrompido, junto con la clave pública y las claves de verificación. Después de esta fase, el adversario envía peticiones de firma para mensajes de su elección a los nodos no corruptos. Ante esta petición, el nodo emite una partición de la firma para el mensaje.

En el modelo existen dos parámetros: (1) el número t de hosts corruptos y (2) el número de particiones k de una clave, necesarias para obtener una firma. Los únicos requerimientos son que $k \geq t+1$ y que $l-t \geq k$. Se trabaja sobre un modelo de corrupción estático, en el que el adversario selecciona un subconjunto t de hosts a corromper al inicio del protocolo.

Algoritmos

Se han implementado dos protocolos. El protocolo 1 define el esquema básico, y puede probarse seguro cuando $k = t + 1$ en el modelo de oráculo aleatorio bajo la suposición RSA. El protocolo 2 define un esquema más general para $k \geq t+1$. Esta generalización es especialmente útil en situaciones en las que los nodos no corruptos no se ponen de acuerdo en lo que están

firmando, pero uno quiere probar que un gran número de ellos ha autorizado una firma en particular. Este protocolo se ha mostrado seguro en el modelo de oráculo aleatorio bajo la suposición RSA cuando $k = t + 1$; y cuando $k \geq t+1$ bajo una variante de la suposición de la Decisión de Diffie-Hellman (DDH) [Bon98].

En el protocolo 1 el servidor central escoge dos números primos grandes, p y q, y obtiene el módulo $N = pq$. También escoge el exponente público e, que es un número primo tal que $e < l$. La clave pública es $PK = (N, e)$. Posteriormente, el mismo servidor obtiene el exponente privado $d \in \mathbb{Z}$ tal que $ed = 1 \bmod m$, donde $m = p'q'$, y $p' = (p - 1/2)$ y $q' = (q - 1/2)$. Finalmente, enmascara la clave privada d dentro de un polinomio $f(x)$ y distribuye al resto de nodos las particiones de la clave $s_i = f(i) \bmod m$.

En el protocolo 2 se realiza una pequeña modificación del protocolo anterior. Las particiones de la clave privada se computan ahora como $s_i = f(i)\triangle^{-1} \bmod m$.

2.2.8. Firmas umbral sin servidor de confianza

Introducción

El siguiente protocolo, descrito en [DK00], es también uno de los más interesantes que hemos encontrado. En él se define un esquema de criptografía umbral RSA que es tan eficiente como la implementación más rápida del esquema de Shoup [Sho00], pero que elimina las dos suposiciones que este hacía: no necesitaremos un servidor centralizado, ni tampoco será necesario que el módulo sea el producto de dos números primos seguros.

Esta implementación es en esencia una generalización del esquema de Shoup, dado que es tan eficiente como él, usa el mismo patrón de comunicaciones, pero no necesita la suposición de los números primos fuertes. Esto se debe a dos razones: primero, porque no se sabe si existe un gran número de estos números primos seguros, y segundo, porque no llegan a ser totalmente seguros (muchos expertos afirman que es más seguro utilizar números primos elegidos lo más aleatoriamente posible).

Otro punto importante a destacar es que a pesar del nombre del protocolo, esta implementación utiliza un servidor honesto encargado de generar

la clave pública pk y las particiones que serán repartidas entre el resto de servidores, al igual que en el protocolo de Shoup. Sin embargo, los autores comentan que el servidor honesto puede ser sustituido por un esquema de generación de claves completamente distribuido usando el modelo de Frankel et al. [FMY98].

En este modelo se considera que existen l hosts y un servidor honesto, que bajo un parámetro de seguridad k genera la clave pública pk y las particiones de la clave secreta $s_1, \ldots, s_l$, donde s_i se envía al host i. Existe también un protocolo de firmado que coge como entrada un mensaje M y genera una firma σ. Finalmente, existe también un predicado de verificación V, que tomando como entrada pk, el mensaje M y la firma, devuelve *accept* o *reject*.

Para modelar el sistema también se asume la existencia de un adversario A estático y activo, que corrompe inicialmente $t < l/2$ hosts.

Algoritmo

El servidor central es el encargado de generar las particiones $p_1, \ldots, p_l$ y $q_1, \ldots, q_l \in_R [2^k - 1, 2^k]$, donde k es el parámetro de seguridad. El servidor elige estos valores hasta que $p = \sum p_i$ y $q = \sum q_i$ sean dos números primos tal que $mcd((p-1)/(2, \triangle) = 1$ y $mcd((q-1)/2, \triangle) = 1$. Obtiene también el módulo RSA $N = pq$ y el exponente público $e < l$. A continuación, ejecuta la generación de un exponente privado arbitrario como en [FMY98] para computar $d\triangle^2 = d_1 + \cdots + d_{t+1} \in \mathbb{Z}$ tal que $ed = 1 \text{ mód } \phi(N), \triangle|d_1, \ldots, \triangle|d_{t+1}$ y $|d_1| < Cl^{l+1}\triangle^{11}N^2, \ldots, |d_{t+1}| < Cl^{l+1}\triangle^{11}N^2$ para una constante $C > 1$. Finalmente, realiza una compartición de secretos sobre los enteros como en [FMY98].

La complejidad de comunicación de cada uno de los servidores es de $\theta(k + l \log l)$ bits y la computación es de $\theta(k + l \log l)$ multiplicaciones modulares, donde l es el número de servidores y k es la longitud del módulo.

2.2.9. DICTATE

Introducción

El año 2005, Patrick Eugster definió una autoridad de certificación distribuida orientada a redes ad-hoc. Para ello desarrolló un protocolo llama-

do *Distributed Certification Authority with Probabilistic Freshness for Adhoc Networks* (*DICTATE* [Eug05]). Este protocolo se basa en la existencia de un servidor central (mCA), encargado de preasignar un papel especial a los nodos que constituyen la autoridad de certificación distribuida (dCA). Esta solución asegura que la dCA siempre procesa una petición o consulta en un tiempo finito, y que siempre responde a las peticiones de consulta con la versión más reciente del certificado solicitado. Para aumentar la seguridad de esta implementación se combina el uso de una autoridad de identificación (IA) off-line, encargada de autentificar la unión inicial de la clave pública y su identidad, y una autoridad de revocación (RA) on-line distribuida, encargada de seguir el rastro del estado de los certificados. Debido a esta separación, el hecho de que la autoridad on-line se vea comprometida no capacita al adversario para poder expedir certificados a los usuarios. DICTATE hace uso de la criptografía umbral para distribuir la confianza entre los servidores RA.

Modelo de sistema y de adversario

En este modelo se asume que cada nodo posee una identidad única. En la red hay un subconjunto $N(\mid N \mid= n)$ de nodos protegidos de forma segura, que pueden ser identificados por la mCA para constituir la autoridad de certificación distribuida. Un subconjunto $T \subset N$ de estos nodos puede verse comprometido durante un cierto periodo de tiempo, donde $\mid T \mid= t < n/3$ (aunque los esquemas de criptografía umbral requieren tan sólo $t < n/2$, el rendimiento del sistema disminuye drásticamente cuando $t < n/3$).

El modelo de sistema integra una IA off-line y una RA on-line. El mCA, conectado al *backbone*, actúa como la IA off-line y asigna un conjunto de nodos n para formar la autoridad de certificación distribuida que actúa como RA. La mCA controla la admisión de un nodo a la red mediante la expedición de un certificado que relaciona la identidad del nodo con su clave pública.

La dCA tiene una pareja de claves RSA pública y privada. La clave privada está distribuida entre los servidores de la dCA constituyendo un criptosistema umbral robusto $(t + 1, n)$. Además, cada servidor posee su pareja de claves pública/privada, donde la clave pública ha sido creada por la mCA y solo representa la autoridad de un servidor individual.

El sistema de comunicaciones está basado en PILOT. Este es un sistema de comunicación de grupos que provee servicios de multicast y replicación de datos. Es un sistema de dos capas (véase figura 2.1): en la base, un protocolo multicast probabilístico, Route Drive Gossip (RDG), y sobre esta capa hay un sistema de quorum probabilístico para redes ad-hoc (PAN), en el cual se asume que la información se almacena replicada entre un grupo de nodos.

Para el modelo de adversario se han tenido en cuenta los siguientes ataques que pueden ser llevados a cabo por un nodo corrupto:

- Impersonación: un nodo pretende suplantar a otro.
- Denial of Service (DoS): un nodo intenta ralentizar el servicio atascándolo.
- Miscelánea: también se ha tenido en consideración ataques de eavesdropping, inserción de mensajes, corrupción, borrado y repetición de mensajes.

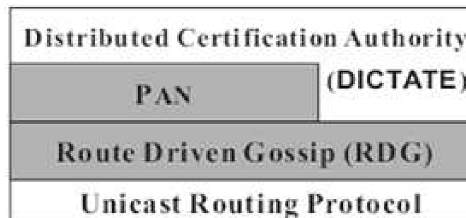

Figura 2.1: *Estructura del protocolo DICTATE construido sobre PILOT (en gris)*

2.2.10. COCA

Introducción

COCA [ZSvR00] es una autoridad de certificación on-line segura y tolerante a fallos, orientada tanto a su despliegue sobre redes de área local como en Internet. Este es el primer sistema que integra un sistema bizantino de quórum (encargado de proveer disponibilidad) con recuperación proactiva

mediante esquemas RSA de criptografía umbral (encargados de la firma digital de certificados de forma segura ante adversarios móviles). La creación de los certificados se lleva a cabo mediante un quórum entre varios de los servidores, donde uno de ellos es el delegado encargado de la comunicación con el cliente y de gestionar la creación del certificado.

COCA está implementado sobre un conjunto de servidores en una red. Un servidor puede estar en estado correcto (sigue correctamente el protocolo) o corrupto. Un servidor corrupto puede parar su ejecución, desviarse arbitrariamente del protocolo especificado (fallo Bizantino) y/o revelar información que tiene almacenada. El proceso de ejecución de COCA se divide en periodos que se han definido como "ventanas de vulnerabilidad", donde los términos correcto y corrupto son relativos a estos periodos de tiempo. Un nodo se considera correcto si, y sólo si, no se ve comprometido en ningún momento durante una ventana de vulnerabilidad. Además, COCA se basa también en un esquema de criptografía umbral t de n, donde $n \geq 3t + 1$. Los clientes y los servidores pueden firmar mensajes usando un esquema completamente infalsificable ante adversarios adaptativos.

En COCA, cada petición de un cliente se procesa por varios servidores y cada certificado se replica en múltiples servidores. El proceso de replicación se gestiona como un sistema de diseminación en quórum Bizantino [MR97] (véase fig. 2.2 en la pág. 30). Por lo tanto, los servidores se asocian en quorums, que satisfacen las propiedades de:

Quorum Intersection: la intersección de cualquier conjunto de dos quorums contiene al menos un servidor correcto.

Quorum Availability: siempre existe un quórum formado únicamente por nodos correctos.

Rendimiento del sistema

COCA contiene aproximadamente 35.000 líneas de código C y emplea esquemas de criptografía umbral RSA proactivos con claves de 1024 bits. El protocolo de comunicación está implementado usando OpenSSL. Los certificados expedidos por COCA tienen la misma sintaxis que los certificados

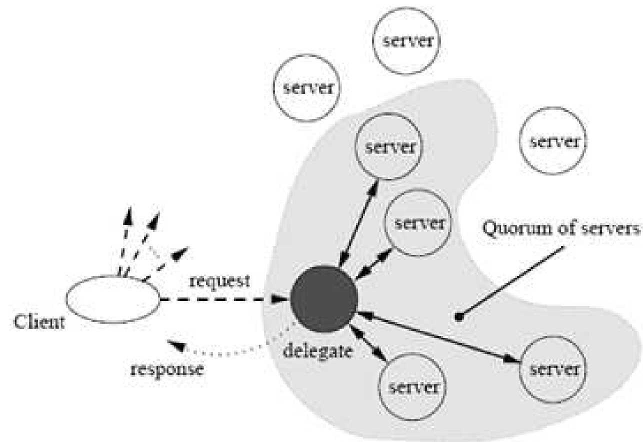

Figura 2.2: *Modelo de sistema de COCA*

X.509, excepto por el número de serie (COCA introduce un número de serie propio).

Los prototipos de este protocolo se han probado en dos ámbitos. El primero es una LAN formada por cuatro servidores comunicados a través de una Ethernet de 100Mbps. Los servidores eran máquinas Sun W420r Sparc corriendo bajo Solaris 2.6. Cada uno de los servidores incorporaba un procesador de 450 MHz. Los tiempos de ejecución promedio para las operaciones de consulta, actualización y compartición proactiva de secretos (PSS) sobre una red LAN se muestran en la tabla 2.3.

Operación COCA	Tiempo promedio (ms)	Desviación std. (ms)
Consulta	629	16'7
Actualización	1109	9
PSS	1990	54'6

Tabla 2.3: *Tiempos de ejecución en una LAN*

El segundo ámbito en el que se han realizado experimentos es el de Internet. La prueba se realizó sobre cuatro servidores situados en diferentes países (Noruega, Alemania y Estados Unidos). Todos ellos ejecutando Linux sobre procesadores con potencias entre los 266 y los 550 MHz. Los tiempos de eje-

cución para las diferentes operaciones de COCA sobre Internet se muestran en la tabla 2.4.

Operación COCA	Tiempo promedio (ms)	Desviación std. (ms)
Consulta	2270	340
Actualización	3710	440
PSS	5200	620

Tabla 2.4: *Tiempos de ejecución en Internet*

2.3. Criptosistemas basados en logaritmos discretos

2.3.1. Criptosistema umbral sin servidor de confianza

Introducción

Esta es la primera implementación que se realizó de criptosistemas umbral sin servidores de confianza. El algoritmo, desarrollado por Perdersen [Ped91] está basado en el sistema de criptografía umbral de El Gamal de Desmedt y Frankel [DF90] e implementa dos mejoras sobre este: no utiliza un servidor de confianza (aunque se necesita un servidor para descifrar un mensaje, se demuestra que éste no necesita conocer la clave secreta de ningún miembro) y demuestra que se puede compartir una clave secreta de tal forma que cualquier miembro del grupo pueda demostrar que es correcta.

Algoritmo

El objetivo del protocolo es que n miembros de un grupo $(P_1, \ldots, P_n)$ generen una clave pública de la forma (p, q, g, h), donde p y q son dos números primos grandes tal que q divide $(p-1)$, y g y h son generadores del grupo G_q (único subgrupo de $\mathbb{Z}_p^*$ de orden q). También han de ser capaces de generar la correspondiente clave privada $x = \log_g h$, de tal forma que $k - de - n$ nodos cooperen entre ellos para generarla en un esquema de criptografía umbral.

Cada uno de los nodos genera una partición de la clave pública y lo distribuye, utilizando mecanismos de *commitment*, al resto de nodos. La clave pública e será computada como $e = \prod_{i=1}^{n} e_i$, y ningún nodo será capaz de obtener la clave privada de forma unilateral. Para computar la clave privada los nodos distribuyen sus particiones x_i usando una adaptación del método de Ingemarsson y Simmons [IS90], de tal forma que al final del proceso cada uno de los nodos será capaz de generar de forma aditiva la clave privada.

2.3.2. Generación de claves distribuida basada en logaritmos discretos

Introducción

Este esquema, descrito en [GJKR99], permite que n servidores generen de forma conjunta un par de claves (pública y privada) para un criptosistema basado en logaritmos discretos, sin que ninguno de ellos tenga conocimiento en ningún momento de ninguna de estas claves y sin la necesidad de un servidor de confianza. Durante el proceso no se computa, reconstruye ni almacena ninguna clave de forma local. El protocolo se basa en un sistema de criptografía umbral donde ningún adversario puede aprender nada a menos que corrompa $t + 1$ servidores.

La idea fundamental de esté protocolo está extraída del algoritmo de Pedersen [Ped91], en el que se tienen n ejecuciones paralelas del protocolo *Verifiable Secret Sharing* de Feldman (Feldman-VSS). Asimismo, demuestra que el protocolo de Pedersen no puede garantizar que el sistema produzca salidas correctas en presencia de un adversario. La versión definida en este documento es robusta, es decir, permite obtener la clave secreta aunque haya t servidores corruptos (siempre y cuando la salida de los servidores no corruptos sea correcta). También se puede implementar una extensión para hacer que el protocolo sea seguro ante adversarios adaptativos.

El esquema de comunicaciones se basa en una red de n servidores conectados mediante canales de comunicación de extremo a extremo. Estos servidores también cuentan con acceso a una red privada de broadcast. El esquema asume un modelo de comunicación completamente síncrono, aunque a la hora de la implementación se utiliza un modelo parcialmente síncrono,

mucho más realista.

Algoritmo

La idea principal del algoritmo es que cada servidor comparte un valor aleatorio y el secreto se genera sumando todos estos valores. Este proceso se divide en dos fases: en la primera se generará el valor x y en una segunda fase se extraerá el valor $y = g^x \bmod p$.

La generación del valor x se realiza de la siguiente forma:

1. Los servidores comienzan ejecutando un mecanismo de commitment llamado *Information-Theoretic Verifiable Secret Sharing* [Ped92] de un valor aleatorio z_i.

2. A continuación, cada host crea un conjunto de hosts no descalificados $QUAL$. Todos los nodos honestos serán capaces de crear el mismo conjunto $QUAL$.

3. El secreto distribuido x será igual a $x = \sum_{i \in QUAL} s_{ji}$

El proceso de extracción de y se realiza mediante el siguiente paso:

4. Cada nodo $i \in QUAL$ expone $y = g^{z_i} \bmod p$ mediante el protocolo VSS de Feldman.

2.4. Criptosistemas basados en curvas elípticas

2.4.1. ECDKG

Introducción

Los criptosistemas de curvas elípticas se constituyen como una alternativa muy prometedora para construir esquemas de generación de claves distribuidas, dado que no hay ningún algoritmo subexponencial conocido que resuelva una versión general de un logaritmo discreto sobre un grupo aditivo derivado de un punto en una curva elíptica.

ECDKG [Tan02] es un protocolo de generación distribuida de claves basado en la técnica *Distributed Feldman Verifiable Secret Sharing* (DF-VSS

[Fel87]) y que utiliza la criptografía de curvas elípticas (ECC) como bloque de construcción. Los nuevos miembros que se quieran incorporar al protocolo lo pueden hacer mediante una primitiva de anexión al grupo, y los que lo quieran abandonar tan sólo han de dejar de responder a las peticiones del protocolo. Este protocolo disfruta de las ventajas que tienen otros protocolos de distribución de claves no basados en curvas elípticas, añadiendo las ventajas de la ECC, como son flexibilidad, generación eficiente de claves, menores necesidades de almacenamiento (se necesitan claves más pequeñas para obtener la misma seguridad que con otros sistemas criptográficos) y menor congestión de la red. Además, ECDKG se muestra seguro ante los siguientes tipos de adversarios y ataques: *halting adversaries*, *eavesdroppers*, adversarios estáticos, ataques de repetición y adversarios adaptativos. El proceso de desencriptación involucra a un cierto número de nodos en un esquema umbral, donde el mensaje puede ser desencriptado sin que ninguno de los miembros del grupo tenga que reconstruir explícitamente la clave secreta.

Modelo de comunicación y de adversario

En el modelo de comunicaciones se asume que hay dos tipos de canales disponibles, canales privados y de broadcast, donde cualquier agrupación de dos hosts se puede comunicar mediante su canal privado. También se asume que el canal privado será, como mínimo, tan seguro como los criptosistemas. Para el canal de broadcast se asume que utiliza algún mecanismo de inundación de la red (por ejemplo, inundación con restricción por TTL).

El protocolo ECDKG se ha diseñado para ser seguro ante adversarios estáticos. Estos adversarios pueden leer cualquier mensaje enviado en un canal no privado, y cuando un nodo está corrupto, su estado y resultados parciales están expuestos al adversario. Se asume que el tiempo de proceso del adversario se puede ignorar en un modelo de adversario estático bajo el supuesto de que el adversario tiene mayor capacidad de computación que todos los nodos. Dentro de este modelo de adversario diferenciamos cinco subtipos diferentes:

- *Halting adversary*: un host puede dejar de responder a los mensajes de forma deliberada o debido a un fallo del procesador.

- *Eavesdropper*: es un adversario que monitoriza de forma pasiva la red, por lo que puede leer todos los mensajes que no sean privados.
- Adversario estático: decide los jugadores a los que va a corromper antes de que se inicie el protocolo, es decir, no puede usar la información obtenida durante el tiempo de ejecución para realizar ataques.
- *Replay adversary* (ataque de repetición): trata de impersonar a otro nodo almacenando mensajes y enviándolos cuando es necesario.
- Adversario adaptativo: puede realizar ataques basándose en la información obtenida en tiempo de ejecución.

Para hacer frente a adversarios adaptativos se utilizan mecanismos de proactivación, por los que las particiones de las claves se van actualizando dinámicamente evitando así que los adversarios puedan ir acumulando información para corromper a los nodos.

Algoritmo

El protocolo ECDKG está basado en una versión mejorada del protocolo *Distributed Feldman Verifiable Secret Sharing*, en el que se incluyen mejoras para proteger el protocolo ante los ataques de los adversarios descritos en el punto anterior. El protocolo usa un *ground field* $GF(q)$, donde q es un número primo o una potencia apropiada de un número primo. Cada nodo tiene un identificador aleatorio único p_i en $GF(q)$ y todos los nodos conocen los identificadores de los demás.

El protocolo está definido por los siguientes supuestos. Sea n el número total de nodos que quieren formar un grupo seguro, y que tienen como identificadores $p_i \in GF^*(q)$. Sea $E/GF(q)$ un grupo aditivo basado en una curva elíptica adecuadamente preseleccionada, y sea T un punto en $E/GF(q)$. La cardinalidad de $E/GF(q)$ es un número primo o tiene un factor primo muy grande. Se asume que todas las multiplicaciones escalares y de puntos se realizan en el grupo aditivo $\mathcal{G}$, mientras que el resto de operaciones aritméticas se realizan en el campo finito $GF(q)$.

El algoritmo se divide en cuatro fases: fase de distribución de claves, fase de verificación de claves, fase de comprobación de claves y fase de generación

de claves. En la primera fase, los nodos eligen $(2t + 2)$ números aleatorios, $a_{ik}, b_{ik} \in GF(q)$ para generar dos polinomios grado t: $f_i(z) = \sum_{k=0}^{t} a_{ik} z^k$ y $f_i'(z) = \sum_{k=0}^{t} b_{ik} z^k$. Computan $s_{ij} = f_i(p_j) \bmod p$ y $s_{ij}' = f_i'(p_j) \bmod p$, y también obtienen $(t + 1)$ valores públicos $P_{ik} = (a_{ik}T) \oplus (b_{ik}T')$, en los que T es un punto en $E/GF(q)$ y T' es otro punto en $E/GF(q)$ cuyo logaritmo discreto respecto a T es desconocido. Finalmente, los nodos distribuyen la información pública (broadcast) y la privada (extremo a extremo a través de canales seguros).

En la fase de verificación los nodos verifican la información recibida y responden con mensajes de confirmación o no confirmación. En la siguiente fase se toma la decisión de eliminar la información proveniente de un nodo i en caso de haber recibido más de t mensajes de que la información que ha enviado dicho nodo i no es correcta. Finalmente, se reconstruyen las claves con la información de los nodos que no han sido declarados corruptos.

En una simulación realizada implementando la curva de Koblitz, el algoritmo tarda 500ms en generar una clave de 283 bits en una red PXA 255 a 200 MHz con RTT = 100ms.

2.5. Conclusiones

Hasta el momento se han hecho una gran cantidad de propuestas de esquemas de generación distribuida de claves. Las propuestas más recientes nos muestran cómo diseñar sistemas que sean eficientes, robustos y seguros ante gran cantidad de adversarios. También se han hecho multitud de avances en lo referente a los algoritmos que implementan, de tal forma que los tiempos de ejecución se minimicen.

Para decidir la forma en que realizaremos nuestra aplicación primero nos vamos a centrar en las ventajas y desventajas de cada uno de los tipos de criptosistemas estudiados, para decidir así cuál es el tipo que más nos conviene para implementar la autoridad de certificación distribuida. Por un lado tenemos que, como se muestra en la tabla 2.5 (los tiempos que muestra la tabla son para la ejecución de los algoritmos en una sola máquina, pero nos sirven de referencia a la hora de hacer suposiciones sobre los sistemas distribuidos), los tiempos de ejecución de RSA a la hora de verificar y de

los criptosistemas de curvas elípticas a la hora de firmar son mucho más óptimos que los demás, llegando a haber diferencias sustancialmente grandes. Las autoridades de certificación, la mayor parte del trabajo lo invierten en atender peticiones de verificación de las firmas, por lo que nos sería más conveniente implementar un sistema que obtenga un mejor rendimiento a la hora de realizar verificaciones que no a la hora de firmar. De tal forma que observando la tabla 2.5 podríamos considerar la opción del RSA como la más óptima.

Criptosistema	**Cifrar**	**Descifrar**	**Firmar**	**Verificar**
RSA (1024b)	17	384	384	17
El Gamal (1024b)	480	240	240	480
CC. Elípticas (160b)	120	60	60	120

Tabla 2.5: *Tiempos (ms) necesarios para ejecutar las diferentes operaciones. (RSA Labs 1997)*

Sin embargo, si observamos la tabla 2.6 vemos que el tamaño de las claves en el modelo de curvas elípticas es considerablemente inferior para obtener la misma seguridad que en los otros modelos. Esto hace que la capacidad de almacenamiento necesaria si implementamos un algoritmo de curvas elípticas sea mucho menor que con los demás. Por lo tanto, la decisión se basa en analizar que característica será más crítica en nuestro sistema, el tiempo o la capacidad de almacenamiento. En este aspecto, hemos considerado que el tiempo es la variable más crítica, puesto que analizando los algoritmos se ha visto que el rendimiento de los sistemas no era muy alto, debido a la complejidad de las implementaciones. Por otro lado, hoy en día los dispositivos de almacenamiento tienen capacidades suficientemente grandes como para que el hecho de trabajar con claves de 512-2048 bits no suponga un gran problema. Así que finalmente decidimos que la implementación que a priori parece más óptima es la de usar un protocolo basado en RSA.

No cabe duda de que el protocolo RSA es uno de los más utilizados, y por lo tanto sobre el que más se ha investigado. Así pues, el siguiente paso consistía en decidir cómo íbamos a implementar el protocolo a partir de todos

RSA-LogDisc	**CC. Elípticas**	**MIPS-Año**
512b		$3 * 10^4$
1024b	160b	$3 * 10^{11}$
2048b	234b	$2 * 10^{23}$

Tabla 2.6: *Comparación tamaño claves (Certicom Labs. 1998)*

los esquemas RSA vistos. Nuestro principal objetivo era el de diseñar una autoridad de certificación que sobre todo fuese segura, robusta y eficiente. Bajo estas premisas la implementación de Damgård [DK00], nos parecía la más completa. Se definía como un protocolo robusto y eficiente (atributos heredados del esquema de Shoup [Sho00]) y que era tan rápido como su predecesor. Además, no es tan sólo un esquema de generación de claves, si no que el protocolo especifica también los mecanismos para firmar y verificar mensajes.

A la hora de generar las claves nos presentaba la opción de utilizar el esquema de Shoup, o hacerlo de forma completamente distribuida implementando el protocolo de Frankel et al. [FMY98]. Nuestra intención era la de crear una autoridad de certificación completamente distribuida, dado que al estar orientada a redes que surgen de forma espontanea y que tienen un gran dinamismo no sería fácil definir qué servidor iba a ser el central y cuáles, actuarían como clientes. Además, cualquier fallo en el servidor central produciría un fallo en todo el proceso, por lo que nos parecía mejor utilizar un esquema distribuido que fuese más tolerante a fallos. Por lo tanto, creímos que era una buena opción utilizar el esquema de Damgård, junto con el de Frankel. El protocolo de distribución de claves de Frankel también nos parecía el más completo y seguro. Era un protocolo robusto y eficiente, lo cual lo convertía en una de las mejores implementaciones. El único inconveniente era que todas estas implementaciones de seguridad podían repercutir negativamente en la ejecución temporal. Por el contrario, la implementación de Malkin [MWB99] había demostrado un buen rendimiento, ejecutándose en unas condiciones de seguridad inferiores. Finalmente, decidimos apremiar la seguridad en nuestra aplicación y optar por basar nuestra implementación

en el desarrollo realizado por Frankel et al., implementando ciertas adaptaciones.

En los siguientes capítulos veremos todo el proceso de desarrollo de un esquema de generación de claves basado en el protocolo descrito por Frankel et al. en [FMY98]. De esta forma, sentaremos la base para una posible implementación del protocolo de Damgård (u otro esquema compatible) para crear una autoridad de certificación distribuida.

CAPÍTULO 3

Estudio de viabilidad de la aplicación

A continuación presentamos el estudio realizado para analizar la viabilidad del proyecto previo a su realización. Este estudio hace referencia tan sólo al desarrollo de la aplicación, ya que la viabilidad del estudio de los sistemas distribuidos de generación de claves es obvia. En este estudio de viabilidad hemos analizado los aspectos más importantes de lo que será el posterior proceso de desarrollo de la aplicación, para finalmente, llegar a unas conclusiones sobre su viabilidad técnica y operativa.

3.1. Objeto

3.1.1. Descripción de la situación a tratar

Durante las últimas décadas estamos asistiendo a grandes avances en el campo de las redes de comunicaciones, lo que ha propiciado que se extiendan de forma masiva en todos los ámbitos de nuestra sociedad. Desde el ámbito profesional al particular, las redes permiten que personas de todo el mundo tengan acceso a servicios remotos con el único requisito de poseer un dispositivo que les permita conectarse a estas redes de datos. La heterogeneidad de las redes por las que pasan las conexiones, así como de las personas que pueden acceder a los servicios y recursos, ha hecho que la seguridad se con-

vierta en un aspecto clave, sobre el que gira el desarrollo e implementación de estas. La aparición hace unos años de las redes inalámbricas[1], que envían los datos a través del aire en vez de utilizar cables, hizo que la seguridad tomara aún mayor importancia.

Dentro de las redes inalámbricas se encuentran las redes ad-hoc, que aunque no sean obligatoriamente inalámbricas, generalmente lo son debido a sus características (Mobile Ad-hoc NETwork - MANET). Estas redes vienen definidas por una topología de red autónoma y sin infraestructuras. Las conexiones se establecen por la duración de una sesión y no requieren de una estación base. En vez de eso, los dispositivos se descubren al resto de dispositivos dentro del rango y crean una red con ellos. Los dispositivos también buscan a los nodos fuera de rango inundando la red con paquetes que el resto de nodos reenvían. Las conexiones se realizan a través de los nodos de la red, es decir, los nodos actúan como routers, descubriendo y manteniendo rutas hacia otros nodos (*multihop ad-hoc network*). Los protocolos de encaminamiento proveen conexiones estables aún y cuando los nodos se están moviendo.

Como hemos comentado en apartados anteriores, la seguridad de estas redes la basaremos en la incorporación de una autoridad de certificación distribuida. Esto permitirá que no sea necesario que exista un servidor central en la red, y que sean los propios nodos los que generen los certificados. El hecho de distribuir la generación de los certificados entre varios nodos dota a este servicio de una mayor seguridad y disponibilidad, ya que el servicio no se concentra en un único nodo que puede funcionar mal, estar corrupto o sufrir ataques, por ejemplo, de denegación de servicios (DoS).

3.1.2. Perfil de usuario y entorno de integración

Esta aplicación está orientada a ser completamente transparente al usuario, por lo que no requiere ningún tipo de medida especial para adaptar su funcionamiento a un grupo de usuarios determinado, ni tampoco requerirá de una interfície gráfica. La aplicación está pensada para que se ejecute

[1] La primera red *wireless*, llamada ALOHAnet fue diseñada en 1971 en la Universidad de Hawaii, pero no fue hasta la aparición en 1997 del estándar IEEE 802.11 cuando empezó su verdadera expansión.

como servicio del sistema, trabajando en *background* y actuando de forma automática cuando reciba una petición por parte de un terminal que esté conectado a la red ad-hoc o por parte de otro de los nodos participantes en el protocolo, que le inste a iniciar el proceso de generación de claves.

La aplicación está pensada para ser ejecutada en cualquier tipo de máquina, desde un computador normal a dispositivos cuya capacidad de procesamiento es limitada, como pueden ser PDAs o teléfonos móviles. Así pues, un aspecto crítico de la aplicación será el tiempo de ejecución, que vendrá determinado por la cantidad de procesamiento que necesite el protocolo. Por lo tanto, la aplicación construida deberá limitarse a implementar el código necesario para un funcionamiento robusto y eficiente, eliminando todos los procesos que puedan ser redundantes o innecesarios. Otra característica a tener en cuenta será la necesidad de desarrollar una aplicación que pueda ser integrada en plataformas muy diversas sin que esto suponga ningún tipo de problema.

3.2. Análisis de requerimientos

3.2.1. Requerimientos funcionales

En este apartado describiremos los requerimientos funcionales de nuestro sistema. Estos nos definen el comportamiento deseado del software, expresando la relación entre entradas y salidas del sistema. Los requerimientos funcionales que caracterizan el comportamiento de nuestra aplicación son:

1. Los resultados de la ejecución de la aplicación deberán ser un módulo N, un exponente público e y una partición d_i del exponente privado d, conforme el estándar RSA. Los valores obtenidos deberán cumplir:

    a) N deberá ser el producto de dos números primos grandes, p y q.

    b) $d = e^{-1} \bmod \phi(N)$.

2. Seguiremos un procedimiento para claves públicas pequeñas que reduzca la complejidad del algoritmo, así como su tiempo de ejecución. Por consiguiente, y al igual que como se describe en [FMY98], asumiremos $e = 3$, donde $mcd(\phi(N), e) = 1$ y $\phi(N) = (p-1) * (q-1)$.

3. Al inicio del protocolo, cada nodo participante S_i deberá elegir p_i y q_i, que permitan computar los factores $p = \sum p_i$ y $q = \sum q_i$.

4. El tamaño de las claves deberá ser suficiente para que el módulo N no pueda ser factorizado en tiempo finito, por lo que el tamaño mínimo será de 512 bits.

5. Cada nodo participante en el protocolo deberá estar identificado de forma única por un identificador numérico. Los identificadores deberán ser consecutivos, comenzando por el número 1.

6. Antes del comienzo del protocolo se deberán identificar los nodos que van a participar en él, y estos deberán revelar su dirección IP y los puertos que usarán tanto para los envíos en multicast como para los envíos en unicast.

7. Ningún nodo deberá poseer en ningún momento información suficiente para factorizar el módulo N o computar el exponente d de forma unilateral o mediante una coalición por de $t \leq (n-1)/2$ nodos.

8. Se usarán mecanismos de transformación de la representación para realizar el envío de datos. De esta forma, el secreto s a compartir estará oculto y sólo podrá ser desenmascarado con la participación conjunta de una coalición de por lo menos $t+1$ nodos.

3.2.2. Requerimientos no funcionales

Describiremos a continuación los requerimientos no funcionales que definen nuestro sistema. Estos enumeran las restricciones que nos impone el propio problema con el que tratamos y que afectan al diseño de la aplicación. Veremos las restricciones sobre el rendimiento del sistema, las decisiones tomadas en lo referente al diseño y otros factores presentes en el entorno que nos afectan a la hora de diseñar la aplicación.

Requerimientos de rendimiento

- Requerimientos estáticos

1. El protocolo se ejecutará con la participación de $n \in [3, 10]$ nodos.
2. Se deberá obtener un resultado válido siempre que un subconjunto t de los n nodos siga el protocolo de forma correcta, donde $n \geq 2t + 1$.
3. Ningún subconjunto formado por $t - 1$ nodos deberá ser capaz de obtener un resultado válido.

- Requerimientos dinámicos

  1. El tiempo de respuesta deberá ser el mínimo posible, viniendo determinado por el tamaño de las claves.

Decisiones de diseño

1. La aplicación se implementará en lenguaje Java, pues nos aportará independencia de la plataforma sobre la que se ejecute y permitirá la integración en la mayoría de dispositivos, incluyendo los teléfonos móviles. Por lo tanto, la aplicación podrá ser ejecutada en cualquier entorno que disponga de una máquina virtual de java (JVM).
2. La máquina donde se integre la aplicación también deberá implementar el protocolo TCP/IP, usado para la transmisión de información. Deberá permitir el envío y recepción de mensajes en multicast y en unicast.
3. La aplicación deberá ser integrada en cualquier tipo de máquina que disponga de un procesador, un dispositivo de almacenamiento y una interfície de red.

Requerimientos sobre interfícies externas

1. La comunicación se realizará a través de un grupo multicast que actuará como red de broadcast para los nodos participantes en el protocolo, y mediante canales extremo a extremo para el envío de información unicast.
2. La comunicación entre nodos se realizará a través de dos puertos. Uno para recibir y enviar información al grupo multicast, y otro para realizar la comunicación unicast mediante UDP.

3.3. Descripción del sistema

3.3.1. Recursos

En esta sección describiremos los recursos hardware y software necesarios para los procesos de desarrollo y prueba del proyecto. Los recursos necesarios para implantar la aplicación en un entorno real se consideran los mismos que los recursos necesarios para hacer las pruebas.

Los recursos necesarios son:

hardware Para el desarrollo del proyecto es necesaria una estación de trabajo para codificar la aplicación y varios hosts en los que se puedan hacer las pruebas de forma que sean lo más realistas posible. Los hosts en los que se realicen las pruebas deberán estar comunicados por un medio de broadcast común, con lo cual han de tener una interfície de red y deben poseer una unidad de almacenamiento con memoria suficiente para albergar la aplicación.

De esta forma, la codificación de la aplicación se ha realizado usando un computador Intel Pentium III corriendo bajo Windows XP Pro SP2. Las pruebas se han realizado en los ordenadores del laboratorio Kleinrock de la Escola Tècnica Superior d'Enginyeria en Bellaterra. Estos ordenadores incorporan un Pentium 4 a 2 GHz. y corren bajo Fedora Core 2.6.10-1.771, estando conectados mediante una red ethernet 10/100 Mbps.

software Para el desarrollo de la aplicación tan sólo necesitaremos un entorno de programación Java que nos permita codificar la aplicación de forma sencilla. A la hora de realizar las pruebas, el único requisito software que necesitan cumplir los hosts es el de tener una máquina virtual de Java actualizada, compatible con la versión 1.5.0_06.

3.3.2. Evaluación de riesgos

En todo proyecto existen una serie de factores que afectan tanto a su desarrollo como a su funcionamiento una vez implantados en un entorno real. Este conjunto de factores de riesgo son los que pueden provocar el

fracaso de nuestra implementación de un sistema de generación de claves RSA distribuido. A continuación analizaremos los riesgos inherentes de esta implementación y buscaremos soluciones para tratar de evitarlos o por lo menos mitigarlos:

Tiempos de ejecución elevados

Como hemos comentado en apartados anteriores el tiempo de ejecución es un aspecto clave en este proyecto. Las características de este proyecto hacen que requiera de una gran cantidad de procesamiento y esto puede afectar negativamente a los dispositivos que incorporen procesadores lentos. También se requiere realizar una gran cantidad de envíos a través de la red, lo que hará que el rendimiento del sistema disminuya en caso de que la red de transmisión sea lenta.

De especial interés es comentar el método de computación de N, dado que la mayor parte del tiempo de ejecución se usará en encontrar un valor de N válido. Como se ha comentado anteriormente, $N = p * q$, donde los factores p y q los computaremos a partir de los p_i y q_i de cada uno de los nodos, de tal forma que $p = \sum p_i$ y $q = \sum q_i$. Como se puede intuir, la probabilidad de que $\sum p_i$ y $\sum q_i$ nos den ambos un número primo es inversamente proporcional al tamaño de p y q. Según [BF97], la probabilidad de que un número aleatorio de n bits sea primo es de $1/n$. Por lo tanto, probabilidad de que los dos valores sean primos es de $1/n^2$. Como los factores p y q tienen que ser muy grandes (Frankel aconseja que como mínimo sean de 400 bits) para aumentar la dificultad de que N pueda ser factorizado posteriormente, la probabilidad de que ambos sumatorios nos den como resultado un número primo es casi nula, por lo que el número de veces que tendremos que ejecutar el protocolo hasta conseguir un N válido será muy elevado.

La solución a este problema se basa en dos aspectos. Por un lado, la reducción del código a ejecutar mediante la eliminación de operaciones redundantes o innecesarias, intentando que la seguridad de la aplicación y su robustez se vean afectadas en la menor medida posible. Por otro lado, se deberían implementar las mejoras comentadas en [FMY98] y [BF97], como son *multi-threading*, *distributed sieving*, balance de cargas o el filtrado de Fermat.

Otro de los factores que pueden producir un tiempo de ejecución mayor es el hecho de que el algoritmo de Frankel et al. utilicen un valor de $e = 3$. Esto provocará que aunque encontremos un valor de N correcto, a veces lo tendremos que desechar si $mcd(e, \phi(N)) \neq 1$ y volver a realizar la computación de otro módulo, lo cual incrementa el tiempo de ejecución de forma sustancial. Utilizando $e = 3$ las posibilidades que un $\phi(N)$ sea múltiplo de 3 (por lo que $mcd \neq 1$)) son relativamente altas. La solución a este problema pasaría por modificar el algoritmo de generación de claves de Frankel y buscar una generalización que nos permita usar claves mayores.

Ataques de adversarios

Otro de los aspectos claves del proyecto será su seguridad, debiendo intentar conseguir un protocolo lo más robusto y proactivo posible, entendiendo por robustez la capacidad de dar un resultado correcto aún en presencia de nodos que se comporten de forma incorrecta y por proactivo, la capacidad de adaptarse a adversarios dinámicos, que modifican su comportamiento a lo largo del tiempo.

Dentro de nuestra red asumiremos la existencia de adversarios activos y pasivos. Estos adversarios pueden corromper un servidor en cualquier momento viendo las memorias de servidores corruptos (adversario pasivo) y/o modificando su comportamiento (adversario activo). También asumiremos que los adversarios son estáticos, es decir, los nodos que van a corromper los deciden al inicio del protocolo.

Para evitar que nuestro sistema sea vulnerable ante estos adversarios tomaremos una serie de medidas. La primera sería utilizar metodologías de conocimiento cero, por las que ningún nodo puede poseer en ningún instante información suficiente sobre el protocolo como para poder obtener los resultados de forma unilateral. Por lo tanto diremos que los nodos siguen un comportamiento "honesto pero curioso", es decir, participan en el protocolo de forma honesta, pero al final de él, ninguna de las partes tiene información suficiente como para poder factorizar N. Esto hará que, aunque un adversario pueda acceder a la información de uno de los nodos, esto no sea suficiente, pues necesitaría corromper al menos $t - de - k$ nodos. Además, en la futura

implementación de la autoridad de certificación se podrían definir ventanas de vulnerabilidad [ZSvR00], que definirían los intervalos de tiempo dentro de los cuales el adversario tendría que conseguir corromper los t nodos, pues después de este intervalo de tiempo el protocolo se reiniciaría invalidando toda la información anterior.

También sería efectivo implementar mecanismos de *commitments*, como por ejemplo los *Simulator-Equivocal Commitments*. Mediante este mecanismo podemos instar a un nodo a demostrar el conocimiento de cierta información, de tal forma que podamos detectar los nodos que pueden estar corruptos, eliminándolos del resto del protocolo.

Todo esto unido con el uso de canales de comunicación seguros, que impidan a los adversarios que estén monitorizando la red, como los *eavesdroppers*, acceder a la información mientras ésta viaja de un nodo a otro. Por ejemplo, se podrían utilizar los protocolos SSL o TLS para asegurar la información y poder certificar la autenticidad del emisor y la integridad de los datos.

3.4. Planificación del proyecto

En este apartado vamos a realizar una planificación de cómo se ha de desarrollar el proyecto y de cómo se codificará la aplicación. Lo primero que analizaremos serán las fases de desarrollo del proyecto, para así obtener una planificación temporal que nos sirva de guía a la hora de implementar la aplicación. A continuación de esto, daremos una visión general de cómo será el protocolo y analizaremos las partes de las que debe constar el programa.

3.4.1. Planificación de tareas

El proceso de desarrollo de este proyecto se ha dividido en 3 grandes fases. La primera consistió en el estudio de los sistemas de generación de claves y autoridades de certificación distribuidas. La segunda sería el desarrollo de este estudio de viabilidad. Y finalmente, tenemos las fases de desarrollo de la aplicación y redacción de la memoria.

En la figura 3.1 podemos ver la planificación realizada de las fases de desarrollo de la aplicación. Se ha intentado hacer una planificación lo más realista posible teniendo en cuenta la disponibilidad horaria que se tendrá

durante los trimestres. Se ha marcado como fecha de inicio del proyecto el día 1 de Noviembre del año 2005, y la fecha de finalización prevista es el 7 de Junio de 2006.

		Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		**Autoridad de certificación distribuida**	**137 días**	**mar 01/11/05**	**mié 07/06/06**	
2		**Recopilación de información**	**37 días**	**mar 01/11/05**	**mié 21/12/05**	
3		Recopilar documentos DKG	10 días	mar 01/11/05	lun 14/11/05	
4		Análisis documentos	25 días	mar 15/11/05	lun 19/12/05	3
5		Redacción informe estudio	25 días	jue 17/11/05	mié 21/12/05	4CC+2 días
6		**Estudio de viabilidad**	**12 días**	**jue 22/12/05**	**vie 13/01/06**	**2**
7		Análisis de requerimientos	1 día	jue 22/12/05	jue 22/12/05	
8		Evaluación riesgos	1 día	vie 23/12/05	vie 23/12/05	7
9		Planificación tareas	1 día	lun 02/01/06	lun 02/01/06	8
10		Documentar estudio viabilidad	10 días	lun 02/01/06	vie 13/01/06	7CC+2 días
11		**Desarrollo aplicación**	**73 días**	**lun 06/02/06**	**mié 17/05/06**	**6**
12		Definir estructuras datos	4 días	lun 06/02/06	jue 09/02/06	
13		Modelar procesos	4 días	vie 10/02/06	mié 15/02/06	12
14		**Implementar código**	**45 días**	**jue 16/02/06**	**mié 19/04/06**	**13**
15		Generación N	15 días	jue 16/02/06	mié 08/03/06	
16		Test primalidad	15 días	jue 09/03/06	mié 29/03/06	15
17		Generación claves	15 días	jue 30/03/06	mié 19/04/06	16
18		Pruebas y corrección de errores	20 días	jue 20/04/06	mié 17/05/06	14
19		Documentar memoria	30 días	jue 27/04/06	mié 07/06/06	18CC+5 días

Figura 3.1: *Planificación de tareas*

En la figura 3.2 se observa el diagrama de Gannt obtenido, en donde podemos ver en color rojo el camino crítico del proceso de desarrollo del proyecto.

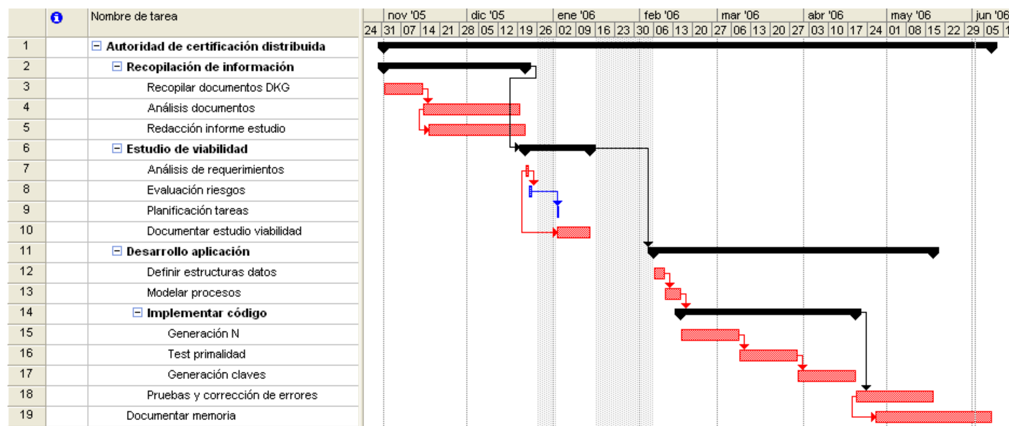

Figura 3.2: *Diagrama de Gannt*

Riesgos de planificación

La planificación de este proyecto creemos que se ajustará bastante a la realidad y que no habrá modificaciones importantes. No hay ningún tipo de riesgo de carácter externo, pues no hay dependencias de entidades externas que puedan establecer una serie de requisitos o modificaciones especiales. El diseño de la aplicación, que veremos en el siguiente apartado, tampoco esperamos que nos cause ningún problema. La aplicación se desarrolla tomando como base un protocolo ya existente, por lo que el diseño que realizaremos se ajusta a una implementación que en teoría es viable.

3.4.2. Diseño de la aplicación

Se han realizado diversos diagramas UML con la herramienta Rational Rose para modelar la aplicación que realizaremos. En los siguientes apartados analizaremos cómo desarrollaremos el programa, y veremos los diagramas que modelan la estructura de la aplicación, el funcionamiento esperado y las dependencias entre clases.

Diagrama de clases

La aplicación debe constar de diversas clases, que definan tanto las estructuras de datos que necesitemos como los módulos que ejecuten el código de la aplicación. Como estructuras de datos, necesitaremos una clase que defina los polinomios que vamos a utilizar para la compartición de secretos, así como las operaciones que se podrán realizar sobre ellos. También necesitaremos una clase que se encargue de gestionar la información necesaria de los nodos participantes (identificadores, IPs, puertos de conexión, etc). Para implementar la aplicación necesitaremos una clase que se encargue de controlar el correcto desarrollo del protocolo, otra clase que se encargue de realizar las operaciones matemáticas y, por último, una clase encargada de la comunicación con el resto de nodos. Además, utilizaremos una clase que realizará las raíces cuadradas de números enteros grandes (java.math.BigInteger).

En la figura 3.3 se puede observar cómo quedarían conformadas las clases, así como los métodos y atributos de estas (las clases DKGen y DKInfoProt no muestran información sobre métodos y atributos por motivos de espacio).

También se pueden observar las relaciones entre ellas. Las clases previstas serían las siguientes:

DKGen: Se encarga de controlar todo el protocolo

DKInfoProt: Realiza todas las operaciones matemáticas del protocolo.

DKCom: Interfaz de comunicaciones.

DKPol: Polinomios de Shamir.

DKInfoNodos: Gestiona la información de los nodos participantes.

DKBigSqrt: Realiza raíces cuadradas de enteros grandes.

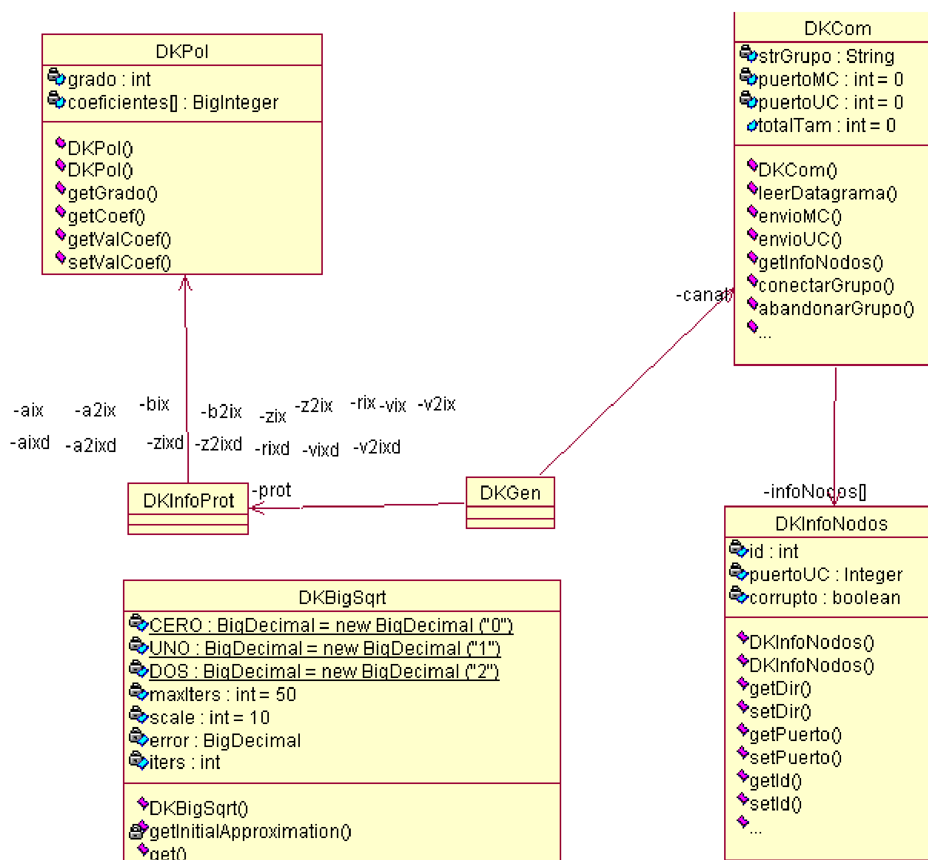

Figura 3.3: *Diagrama de clases de la aplicación*

Flujo de ejecución

En la figura 3.4 podemos observar el que sería el flujo de ejecución del protocolo. Tras iniciarse y configurarse el programa, se ejecutaría la generación distribuida de N. El resultado obtenido debe pasar el test de doble primalidad y volver a obtener otro módulo N en caso de que no lo pase. Si el resultado del test es positivo, se procede a generar el exponente público y el exponente privado de forma distribuida. En caso de que el máximo común divisor del exponente público y $\phi(N)$ no sea igual a 1, se descarta el módulo obtenido.

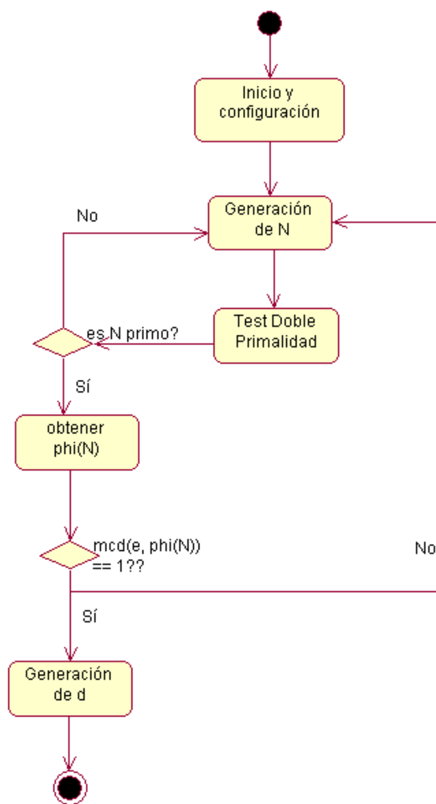

Figura 3.4: *Diagrama de actividades de la aplicación*

En la figura 3.5 vemos más detalladamente la interacción entre las clases DKGen, DKInfoProt y DKCom para llevar a cabo todo el proceso de generación de claves.

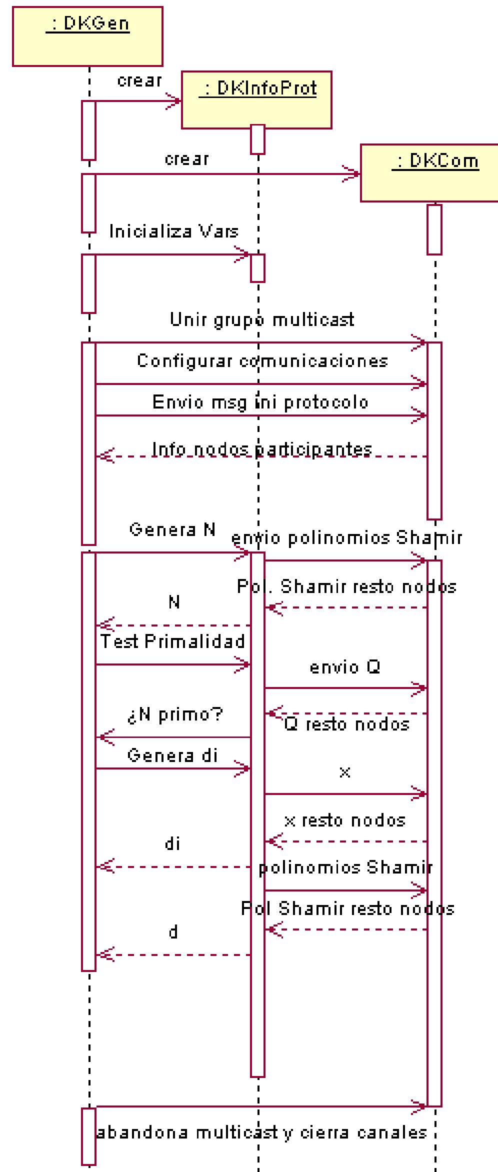

Figura 3.5: *Diagrama de secuencia de la aplicación*

3.5. Conclusiones sobre la viabilidad

Una vez analizada la viabilidad del proyecto procedemos a comentar las conclusiones a las que hemos llegado, tanto sobre la viabilidad del proyecto como de la consecución de los objetivos académicos que persigue la realización de un proyecto final de carrera.

Por un lado consideramos que la realización del proyecto es viable. Hemos visto cuales son los riesgos técnicos del proyecto y, aunque se puedan considerar importantes, hemos descrito una serie de soluciones para todos ellos. Los requerimientos técnicos de la aplicación no suponen tampoco ningún obstáculo importante, siempre y cuando se realice una planificación correcta del proceso de desarrollo de la aplicación y se siga una metodología de desarrollo adecuada. Los recursos necesarios tampoco suponen un gran problema, pues no se necesita gran cantidad de recursos y contamos con acceso a todos ellos.

Académicamente, se puede considerar que este proyecto se ajusta adecuadamente a los requisitos de un proyecto de final de carrera. Por un lado, el proyecto nos plantea una serie de desafíos como son la realización de un estudio del estado del arte sobre un mercado que está en auge, como es el de la seguridad, y finalmente el desarrollo de un nuevo protocolo, que se intenta adaptar a una serie de necesidades que han surgido recientemente debido a la aparición de nuevas tecnologías (en este caso, las redes ad-hoc, que aunque fueron diseñadas en la década de los 70, no han tenido un despliegue amplio hasta hace pocos años en que se hicieron populares los dispositivos móviles como son los ordenadores portátiles, las PDAs, los teléfonos móviles o las videoconsolas).

Cabe resaltar también que este proyecto se adecua perfectamente a los conocimientos adquiridos en la carrera, y abarca gran cantidad de materias: seguridad computacional, redes, matemática discreta e ingeniería del software serían los campos más destacables que abarca el proyecto. Además, el hecho de que el proyecto se fundamente en un campo específico que no hemos estudiado, como son los criptosistemas distribuidos, hará que tengamos que ampliar nuestros conocimientos en las materias comentadas anteriormente.

CAPÍTULO 4

Desarrollo de la aplicación

4.1. Descripción de la aplicación

La aplicación que hemos desarrollado es un nuevo protocolo de generación distribuida de claves RSA. La aplicación es eficiente, tolerante a fallos e incorpora algunos mecanismos para dotarla de cierta robustez. Nuestro protocolo está basado en el desarrollado por Frankel et al. [FMY98], pero pretende tener una complejidad menor, que lo haga más rápido, a costa de penalizar en la robustez del sistema.

La aplicación se encarga de generar un módulo RSA $N = pq$, donde p y q son dos números primos, así como los exponentes público e y las particiones d_i del exponente privado d, tal que $d = e^{-1} \bmod \phi(N)$ y $\phi(N) = (p-1)(q-1)$. La generación de las claves se realiza de forma conjunta entre n participantes, siguiendo un esquema de criptografía umbral $t - de - n$, donde $t \leq (n-1)/2$. Por lo tanto, será necesaria la participación de por lo menos $t+1$ nodos para computar unas claves válidas. Cualquier subconjunto de t nodos, será incapaz de obtener la información necesaria para computar las claves, y ningún nodo tendrá en ningún momento información que le permita factorizar N. Este protocolo es completamente distribuido y elimina la necesidad de que exista un servidor de confianza que se encargue de generar

las claves y distribuir particiones de estas al resto de nodos. Por lo tanto, el hecho de que haya uno o más nodos que fallen durante el protocolo no impedirá que se generen unas claves correctas (siempre y cuando el número de nodos correctos sea superior al umbral t). El sistema también será seguro ante adversarios estáticos, que corrompan inicialmente $t \leq (n-1)/2$ hosts y ante los denominados *halting adversaries* que pueden provocar que un host deje de participar en el protocolo.

Siguiendo las indicaciones de [BF97], el módulo N construido es un entero de Blum. Según la teoría de números, un número natural $N = pq$ es un entero de Blum si p y q son congruentes a 3 módulo 4. Esto hace que para todo entero de Blum, -1 tiene un símbolo de Jacobi igual a 1, aunque -1 no es un residuo cuadrático de N. Conseguir que los nodos participantes computen un entero de Blum de forma distribuida es sencillo, tan solo hay que controlar que el nodo S_1 compute $p_1, q_1 \equiv 3 \pmod 4$ y que el resto de nodos computen $p_i, q_i \equiv 0 \pmod 4$.

Para distribuir los secretos de forma segura entre los nodos se sigue un esquema basado en el protocolo de compartición de secretos de Shamir [Sha79] y en técnicas más recientes de transformación de la representación de las particiones vistas en [FGMY97]. El esquema de Shamir está basado en la interpolación de polinomios. De esta forma, un dealer D elige un polinomio $f \in \mathbb{Z}_q[x]$ de grado t, tal que $f(0) = s$ (s es el secreto a compartir), y a continuación, envía a cada nodo S_i el valor de $f(i)$. El valor de s puede ser recuperado aplicando la fórmula de la interpolación de Lagrange (ec. 4.1) para $f(0)$. El esquema propuesto por Shamir implementa mecanismos de conocimiento cero, que impiden que los nodos sean capaces de aprender nada durante el protocolo.

$$f(x) = \sum_{j=1}^{k} (\prod_{l \neq j} \frac{x - i_j}{i_l - i_j}) f(i_j) \tag{4.1}$$

En combinación con el protocolo de Shamir se utilizan las técnicas de transformación de la representación *sum-to-poly* y *poly-to-sum*. La primera transforma una función compartida entre n servidores en una función $t-de-n$, mientras que la segunda realiza la operación inversa. Estas funciones han sido implementadas sobre el conjunto de los enteros. Nuestra imple-

mentación no es incondicionalmente segura como en el esquema de Frankel et al., dado que a diferencia de su implementación nosotros no hemos utilizado mecanismos de *commitment* que incrementen la robustez del sistema. Ésta fue una decisión tomada como medida para reducir la complejidad del protocolo. Por lo tanto, la utilización de mecanismos de *commitment* que permitan crear un protocolo robusto y proactivo de compartición de secretos [Ped92] queda como problema abierto para su estudio en futuras implementaciones.

El modelo de sistema implementado tiene además las siguientes características:

Conectividad total: Cada nodo S_i está comunicado con el resto de nodos S_j. Para la comunicación se utilizan dos tipos de canales. Por un lado existe un medio de *broadcast* común, que hemos implementado mediante un grupo multicast. Al comienzo del protocolo todos los nodos deberán unirse a dicho grupo para poder recibir los datagramas que se envíen en multicast. La dirección IP a la que se conecta el grupo es la 239.253.0.1, que pertenece al rango de direcciones para grupos reservados a *sites*. La aplicación utiliza el puerto 5555 para acceder a este grupo.

Por otro lado, los nodos también tienen conexión extremo a extremo con cada uno de los nodos. Se conectan mediante canales no seguros usando el protocolo UDP (a través del puerto 5556). Esto permite que no sea necesario establecer conexiones entre los nodos, lo que simplifica la implementación y aumenta el rendimiento del sistema ya que la información no hace falta codificarla. El problema de esta implementación reside en que un adversario que pueda monitorizar el canal tendrá acceso a la información que se transmita a través de él, por lo que si es capaz de leer la información de más de t canales podría reconstruir la información. Otro aspecto a tener en cuenta es que estos canales tampoco son autentificados, por lo que un adversario podría suplantar a otro nodo y enviar información falsa a los nodos.

Si hubiésemos querido implementar un sistema de comunicaciones seguro, hubiésemos necesitado implementar canales SSL. De esta manera se establecen canales por los que la información pasa encriptada, mediante algoritmos de cifrado simétrico, y también se requiere que los nodos se autentifiquen (de forma unilateral o mutua) mediante certificados. Esta implementación, aunque mucho más segura, presenta ciertos problemas. El primero es que el

algoritmo es mucho más complejo, pues al ser un protocolo TCP requiere de conexión, y el establecimiento y mantenimiento de los canales hace que aumente la complejidad y disminuya el rendimiento del sistema. Segundo, el protocolo SSL es un protocolo de estructura cliente-servidor, por lo que al trabajar con nodos que están al mismo nivel hace que un mismo nodo deba actuar como servidor con unos nodos y como cliente con otros, lo cual también repercute en la complejidad de los algoritmos de configuración. Tercero, el hecho de que la información deba ser cifrada también influye negativamente en el rendimiento, ya que la cantidad de envíos que se realizan durante el protocolo es muy alta. Y, finalmente, aparece el problema de la validación de los certificados usados en la comunicación. Recordemos que si estamos trabajando en una red ad-hoc no podemos acceder a ninguna autoridad de certificación, excepto a la que nosotros estamos creando, por lo que a no ser que el certificado estuviese en nuestro *truststore* tendríamos problemas para validarlo. Por todas estas razones decidimos implementar canales UDP no seguros, dejando el problema abierto para su estudio en futuras implementaciones.

Modelo de nodos honestos, pero curiosos: Se asume que todos los nodos siguen de forma correcta el protocolo, y que al final de éste ningún nodo ha conseguido la suficiente información como para poder factorizar el módulo N de forma unilateral. Aunque en principio se asume que los nodos participan correctamente en el protocolo cabe la posibilidad de que alguno de ellos abandone el protocolo de forma inesperada. Nuestro protocolo es tolerante a este tipo de fallos, siendo capaz de detectarlos, tratarlos y obtener un resultado siempre que el número de nodos correctos sea superior al umbral t. Los nodos detectan que otro nodo ha abandonado el protocolo de forma súbita cuando se produce un *timeout* al esperar el envío de datos por parte de ese nodo. El *timeout* se ha establecido en 5 segundos, dado que hemos considerado que es tiempo más que suficiente para que se pueda realizar una comunicación en una red ad-hoc aunque haya cierta congestión en la red, y porque se ha intentado que la penalización sea la menor posible para que no afecte en gran medida al tiempo de ejecución total. Una vez detectado el abandono de un nodo la aplicación procede a activar un flag de nodo corrupto, de tal forma que durante el resto del protocolo no se intente

conectar con este nodo y se asigna el valor 0 a toda la información relativa a este nodo. De esta forma, se podrán obtener las claves mediante la ecuación de Lagrange sin que la información de ese nodo afecte al resultado.

Asincronismo parcial: El modelo de Frankel, al igual que la mayoría de modelos, supone que los servidores actúan de forma sincronizada, sin embargo, el sistema que hemos desarrollado no tiene que ser completamente síncrono. Los datagramas pueden llegar incluso desordenados, y nuestra aplicación los ordenaría en función del indicador de estado. Tan sólo se requiere que cada nodo haya recibido los paquetes que necesita para realizar la siguiente operación. Esta flexibilidad en la suposición de sincronismo también dota a la aplicación de una menor complejidad.

Durante el protocolo se realizan gran cantidad de envíos, y más aún cuantos más nodos participan. La forma de saber a qué nodo pertenece y qué es lo que contiene cada datagrama se consigue identificando cada uno de los paquetes con un identificador de nodo y con un valor de estado. El identificador de nodo es el mismo identificador que posee cada uno de los nodos dentro del protocolo. Los nodos deben de ser consecutivos, empezando desde el número 1 hasta el número de nodos n. En el caso de que se intenten asignar identificadores que estén fuera de este rango, o en el caso de que existan nodos con identificadores iguales la aplicación abortará su ejecución informando al usuario del problema. Estos identificadores se asignan de forma manual al ejecutar la aplicación. En futuras versiones se podría estudiar la forma de implementar un protocolo para configurar los nodos de forma automática y transparente al usuario, usando algoritmos como, por ejemplo, el de Nakano [NO00]. Los estados son divisiones que se han realizado del código, de tal forma que dentro de un estado tan sólo se realiza un envío. De esta forma, podemos saber el contenido del datagrama mirando el estado al que pertenece, y de esta forma guardar la información recibida en las variables correctas. Por lo tanto, todos los envíos realizados por el protocolo tendrán el siguiente formato: *id_estado#id_nodo#valor1#valor2#...*

Principio de conspiración: Nuestro protocolo, como hemos comentado anteriormente, tiene una privacidad $\lfloor \frac{t-1}{2} \rfloor$. Por lo tanto, ninguna coalición de como máximo $t-1$ nodos es capaz de obtener la información necesaria para factorizar $N = pq$. Esto se debe a que parte del protocolo está basado

en el protocolo de Ben-Or, Goldwasser y Widgerson (BGW) [BOGW88]. Esto también implica que el número de participantes debe ser como mínimo 3. El número máximo de nodos permitidos es de 10, por lo que en redes en las que quieran participar más de 10 nodos se deberá establecer algún protocolo para realizar quorums de nodos, como por ejemplo en [ZSvR00].

4.2. Protocolo

Durante este capítulo describiremos el algoritmo implementado y lo analizaremos paso a paso. El algoritmo principal del protocolo se muestra a continuación:

1. Inicio y configuración del sistema.
2. Computación distribuida de N.
3. Test distribuido de doble primalidad. Si el test falla, volvemos al punto 2.
4. Computación distribuida de los exponentes público y privado.

4.2.1. Inicio y configuración del sistema

Durante la fase de inicio y configuración, el primer paso que se realiza es el de llamar a los constructores de todas las clases que participan en el sistema, de tal forma que todas las variables queden inicializadas. Una vez hecho esto procedemos a configurar el sistema de comunicaciones. En primer lugar creamos e inicializamos los sockets que usaremos y nos conectamos al grupo multicast. Una vez está todo inicializado y nos hemos conectado al grupo multicast, los nodos se quedan a la espera de recibir por multicast un mensaje indicando el comienzo del protocolo. Este mensaje es tan sólo un datagrama que contiene como valor de estado el número 0, y que en nuestra implementación es enviado por el nodo S_1, forzando así a que el protocolo comience. A continuación, todos los nodos responden enviando un datagrama donde se indica el puerto de conexión que utilizan para las comunicaciones en unicast. A partir de estos datagramas podemos saber cuantos nodos participan, sus direcciones IP y los puertos por los que escuchan, por

lo que ya tenemos toda la información necesaria para poder comenzar con la generación de claves.

4.2.2. Computación distribuida de N

El objetivo de esta fase es obtener de forma distribuida $N = \sum p_i \sum q_i$. Para ello comenzamos calculando los valores $L = n!$ y $H = 2^h$, donde h es el parámetro de seguridad, que en nuestro caso tendrá el mismo valor que el tamaño de las claves. Después, generamos N aplicando los siguientes pasos:

1. Cada nodo S_i elige $p_i, q_i \in [\sqrt{H}/2, H]$
2. A continuación distribuiremos estos valores entre el resto de nodos utilizando el esquema de Shamir sobre los enteros, junto con las técnicas de transformación de la representación vistas en la sección anterior (pág. 56). Los pasos que se realizan son los siguientes:
   - *a*) Una vez que los nodos S_i tienen los factores p_i y q_i los distribuyen, mediante conexiones unicast, al resto de nodos S_j usando polinomios aleatorios $a_i(x) = \sum_{j=0}^{t} a_{i,j}x^j$ y $b_i(x) = \sum_{j=0}^{t} b_{i,j}x^j$, tal que $a_i(0) = L^2 p_i$ y $b_i(0) = L^2 q_i$. Cada nodo S_j recibe $a_i(j)$ y $b_i(j)$.
   - *b*) Cada nodo S_i genera una pareja de secretos compañeros p'_i y q'_i y los distribuyen al resto de nodos mediante los polinomios $a'_i(x) = \sum_{j=0}^{t} a'_{i,j}x^j$ y $b'_i(x) = \sum_{j=0}^{t} b'_{i,j}x^j$, tal que $a'_i(0) = L^2 p'_i$ y $b'_i(0) = L^2 q'_i$.
   - *c*) Los nodos generan además otros polinomios aleatorios $z_i(x) = \sum_{j=1}^{2t} z_{i,j}x^j$ y $z'_i(x) = \sum_{j=1}^{2t} z'_{i,j}x^j$. Los coeficientes de estos polinomios se eligen tal que $z_{i,j} \in [0, L^{11}H^3]$ y $z'_{i,j} \in [0, L^{11}H^4]$. Como se puede observar, los coeficientes de grado 0 de estos polinomios toman el valor 0. $z_i(x)$ y $z'_i(x)$ se distribuyen al resto de nodos.
   - *d*) Cada S_i genera un último polinomio aleatorio $r_i(x) = \sum_{j=0}^{t} r_{i,j}x^j$ y distribuye las particiones.

   Los valores de los coeficientes de grado superior a 0 de los polinomios usados en el protocolo, excepto los de $z_i(x)$ y $z'_i(x)$, para compartir un

secreto $s \in [0, K]$ se escogen $coef_j \in_{\mathcal{R}} [0, L, 2L, \ldots, L^3K^2]$, donde $\mathcal{R}$ es el conjunto de números aleatorios.

3. Sean $A = \sum_{i=1}^{n} a_i(x)$ y $A' = \sum_{i=1}^{n} a'_i(x)$. Cada S_i obtiene $v_i(x) = Ab_i(x) + z_i(x)$ y $v'_i(x) = A'b_i(x) + z'_i(x) + r_i(x)$.

4. A continuación, cada S_i computa $v(x) = \sum_{i=1}^{n} v_i(x)$ y $v'(x) = \sum_{i=1}^{n} v'_i(x)$ y revela estos valores al resto de nodos enviándolos en multicast.

5. Finalmente, cada nodo interpola los valores de $v(x)$ recibidos para obtener $v(0)$. El valor del módulo N será $N = v(0)/L^4$.

4.2.3. Test distribuido de doble primalidad

El objetivo de este test es comprobar si N es el producto de dos números primos, y en caso contrario descartarlo. Para realizar este proceso hemos implementado una adaptación entre el esquema de test probabilístico de Boneh y Franklin [BF97] y el de Frankel et al. [FMY98]. Este proceso permite realizar el test de primalidad sin revelar ninguna información sobre los factores de N. Esta implementación no es tan robusta como la de Frankel et al., pero su complejidad es menor, al igual que la cantidad de información transmitida.

Los pasos descritos a continuación se repiten h veces para obtener el nivel de seguridad deseado (recordemos que h es el parámetro de seguridad):

1. El nodo S_1 elige $g \in \mathbb{Z}_N^*$ tal que el símbolo de Jacobi $(\frac{g}{N}) = 1$, es decir, g deberá ser residuo cuadrático módulo N. No obstante, como se ha comentado en secciones anteriores, el módulo N es un entero de Blum, por lo que -1 toma el valor 1 sin ser residuo cuadrático. S_1 hace público este valor ante el resto de nodos.

2. S_1 distribuye en multicast $Q_1 = g^{N+1-p_1-q_1}$. Cada S_i, siendo $1 < i \leq n$, calcula $Q_i = g^{p_i+q_i}$ y lo distribuye en multicast a los nodos del grupo.

3. Finalmente, los nodos verifican que se cumple la equivalencia de la ecuación 4.2. En caso de que no sean equivalentes declaran que N no

es el producto de dos números primos, provocando que los nodos deban elegir otro módulo RSA N.

$$\frac{Q_1}{\prod_{i=2}^{n} Q_i} \stackrel{?}{=} \pm 1 \pmod{N} \tag{4.2}$$

4.2.4. Computación distribuida de los exponentes público y privado

A la hora de afrontar el problema de generar de forma distribuida los exponentes público y privado el protocolo de Frankel nos ofrece dos alternativas. La primera funcionaría sólo para exponentes públicos pequeños (por ejemplo, $e < 1000$) y la segunda funcionaría con cualquier exponente. Para esta aplicación decidimos implementar la primera de las aproximaciones, dado que es muy eficiente y requiere de menos comunicaciones que la segunda.

Al igual que Frankel et al. hemos asumido que el exponente público toma el valor $e = 3$, dado que esto reducirá en gran medida la complejidad del protocolo. El objetivo por lo tanto es que cada nodo S_i obtenga una partición d_i del exponente privado $d = e^{-1} \bmod \phi(N)$. Hay que tener en cuenta que en este caso $\phi(N)$ satisface $\phi(N) = \sum_{i=1}^{n} \phi_i$ donde $\phi_1(N) = N - p_1 - q_1 + 1$ y $\phi_i(N) = -p_i - q_i$, $\forall i > 1$. Los pasos que se siguen para obtener d_i son los siguientes:

1. S_1 elige $g \in_{\mathcal{R}} [1, N-1]$ y lo distribuye en multicast.

2. Cada S_i calcula $g^{p_i+q_i} \bmod N$ y $x_i = p_i + q_i \bmod 3$, y distribuye ambos valores en multicast.

3. Los nodos computan $\phi(N) \bmod 3$ a partir de la equivalencia $\phi(N) \equiv N + 1 - \sum_{i=1}^{n} x_i \bmod 3$. En caso de que $\phi(N) \equiv 0 \pmod{3}$ nos vemos obligados a descartar el módulo N obtenido, ya que $mcd(e, \phi(N)) \neq 1$. En este caso lo ideal hubiese sido descartar el exponente público, dado que nos evitaría gran cantidad de computación para volver a obtener N, sin embargo, Frankel no provee un algoritmo general para generar el exponente privado, si no que tan solo provee un algoritmo para $e = 3$.

4. S_1 obtiene $r = N + 2 - \sum_{i=1}^{n} x_i$ si $\phi(N) \equiv 2 \pmod{3}$, u obtiene $r' = 2N + 3 - 2\sum_{i=1}^{n} x_i$ si $\phi(N) \equiv 1 \pmod{3}$.

5. Cada nodo S_i calcula su partición del exponente privado en función del valor de $\phi(N)$:

    a) Si $\phi(N) \equiv 2 \pmod 3$: S_1 computa su partición $d_1 = \frac{r-(p_1+q_1-x_1)}{3}$ mientras que el resto de nodos $S_i \forall i \in [2,n]$ computa $d_i = \frac{-(p_i+q_i-x_i)}{3}$.

    b) Si $\phi(N) \equiv 1 \pmod 3$: S_1 computa su partición $d_1 = \frac{r'-2(p_1+q_1-x_1)}{3}$ mientras que el resto de nodos $S_i \forall i \in [2,n]$ computa $d_i = \frac{-2(p_i+q_i-x_i)}{3}$.

Cuando los nodos necesitan reconstruir la clave d realizan el mismo proceso que se hizo para la generación de N. En este caso la cantidad de polinomios necesarios es menor que para computar N ya que sólo existe un secreto a compartir en vez de dos. Cada nodo S_i genera los polinomios $a_i(x), a'_i(x), z_i(x), z'_i(x)$ y $r_i(x)$, con las mismas propiedades que para la generación de N. Los valores de $v_i(x), v'_i(x)$ en esta ocasión se computan $v_i(x) = A + z(x)$ y $v'_i(x) = A' + z'_i(x) + r_i(x)$, siendo $A = \sum a_i(x)$ y $A' = \sum a'_i(x)$. Tras calcular y distribuir $v(x)$ los nodos serán capaces de obtener d interpolando estos valores para obtener $v(0)$.

4.3. Clases

La aplicación que hemos desarrollado consta de 6 clases: DKGen, DKInfoProt, DKCom, DKInfoNodos, DKPol y DKBigSqrt. Las clases están relacionadas entre ellas como vimos en la figura 3.3 en la página 51. A continuación daremos una visión general del funcionamiento de las clases.

4.3.1. El generador de claves

La clase DKGen, implementada en el archivo DKGen.java, contiene el algoritmo de control de todo el proceso. En la figura 3.4 en la página 51, se puede ver el flujo de ejecución general del protocolo, que es el que implementa esta clase.

La clase DKGen es de ámbito público y es la que hemos de ejecutar para que se inicie el proceso de generación de las claves RSA. El comando de ejecución tiene el formato

```
>java DKGen idNodo
```

donde *idNodo* es el identificador que le queremos asignar al nodo. Estos identificadores se comprueban de forma automática para comprobar que no estén fuera de rango. Los identificadores de los nodos han de pertenecer al rango $[1, n]$, y es el constructor de la clase DKGen el encargado de comprobar que los nodos pertenezcan a este rango. Posteriormente, cuando la clase DKCom reciba la información de configuración del resto de los nodos comprobará que los identificadores no estén repetidos para dos o más nodos. En caso de que los identificadores no se asignen de forma correcta se abortará la ejecución.

Las variables miembro de la clase DKGen se encargan del control de todo el proceso. Por un lado tenemos las variables que se encargan de controlar el número de nodos que participan en el protocolo y su estado. De esta forma tenemos una variable que lleva la cuenta de los nodos, otra que lleva la cuenta de los nodos corruptos (nos referimos con corruptos a que han dejado de participar en el protocolo de forma repentina por cualquier motivo) y un array que nos indicará cuáles son los nodos corruptos con los que no debemos interactuar. Estas variables controlan que el protocolo cuente con un número de nodos entre 3 y 10, y que el número de nodos corruptos no supere el umbral t, pues el resultado que se obtendría no sería correcto. Por otro lado tenemos las instancias de las clases con las que interactúa para realizar el proceso (DKInfoProt y DKCom). Y finalmente, tenemos las variables que controlan el estado interno del host, que son el identificador y el controlador de estado (indica que proceso estamos realizando y nos permite identificar los valores recibidos en los datagramas).

La clase contiene una función para cada una de las fases del protocolo: generación distribuida de N, test de primalidad y generación de los exponentes público y privado. Estas funciones se encargan de realizar las llamadas a las funciones matemáticas que se necesitan para generar las claves (que son métodos de la clase DKInfoProt) y de interactuar con la interfaz de comunicaciones para enviar y recibir la información del resto de los nodos (métodos de la clase DKCom). Todas estas funciones devuelven una variable booleana que indica si la ejecución ha sido correcta y se puede continuar con el protocolo. Todo el proceso está controlado por la función *iniProt()*, que implementa el algoritmo general visto en el diagrama de actividades (fig. 3.4 en pág. 52), y que mostramos a continuación:

```
//nos unimos al grupo multicast
canal.conectarGrupo();

//obtenemos info del resto nodos
canal.config(id, prot.getNumNodos());
DKInfoNodos[] in = canal.getInfoNodos();

while (!correcto)
{
    //computamos el módulo N
    correcto = this.computarN();

    //ejecutamos el test de doble primalidad secParam veces
    int i = 0;
    while ((correcto) && (i < prot.getSecParam()))
    {
        correcto = this.testDoblePrimalidad();
        i++;
    }

    //computamos el exponente privado d
    if (correcto)
        correcto = this.computarClaves();
}

//abandonamos el grupo multicast
canal.abandonarGrupo();
```

En el algoritmo anterior podemos ver las tres funciones principales del proceso de generación de las claves:

- computarN(): Se encarga de generar de forma distribuida el módulo RSA N. Elige p_i y q_i, distribuye los polinomios de Shamir al resto de nodos, y realiza la interpolación para $v(0)$, obteniendo así el módulo N. Éste queda almacenado en la variable N de la clase DKInfoProt.

- testDoblePrimalidad(): Esta función se encarga de comprobar que N sea el producto de dos números primos. En caso de que pase el test devuelve *true*, y si no devuelve *false*, provocando que se vuelva a computar otro módulo N. Como se puede apreciar, el test se ejecuta el número de veces que determine el parámetro de seguridad (en nuestro caso 512 veces), dado que es un test probabilístico y podría dar un resultado positivo para un valor que no fuese producto de dos números primos. De esta forma, si todas las veces que se ejecuta devuelve positivo, reducimos las posibilidades de error hasta hacerlas prácticamente nulas.
- computarClaves(): Elige el exponente de encriptación público e (en nuestro caso $e = 3$), y genera las particiones d_i del exponente privado. En el caso de que $mcd(e, \phi(N)) \neq 1$ devuelve *false*, y en el resto de casos consigue computar d_i correctamente y devuelve el valor *true*. Los valores resultantes se almacenan en las variables e y d_i de la clase DKInfoProt.

Hay que destacar que en las funciones anteriores los envíos de polinomios a los nodos se hacen de forma secuencial. Es decir, en vez de enviar todos los polinomios y los datos en bloque se envían en el mismo orden que se especifica en el documento [FMY98]. (primero, $a_i(x)$ y $b_i(x)$, después, $a'_i(x)$ y $b'_i(x)$, ...). De esta forma dejamos el algoritmo preparado por si se desea implementar en el futuro, y de forma sencilla, algún mecanismo de *commitment*.

Aparte de estas funciones tenemos una serie de funciones auxiliares, que se encargan de realizar funciones de extracción de información de los datagramas recibidos, generación de los mensajes que se han de enviar, comprobación de invariantes (las precondiciones también se comprueban dentro de cada función) y eliminación de información de nodos que han sido declarados corruptos.

4.3.2. Operaciones del protocolo

Todas las operaciones y cálculos matemáticos que necesita el protocolo están implementadas en la clase DKInfoProt (archivo DKInfoProt.java). Esta clase cuenta con una gran cantidad de variables miembro. En primer lugar

tenemos las variables que almacenarán los resultados del protocolo, es decir, el módulo N y los exponentes público y privado, así como los valores p_i y q_i. En segundo lugar tenemos todos los polinomios que usaremos durante el protocolo, así como los arrays que contienen los valores $f(i)$ de los polinomios que nos han enviado el resto de hosts. Por último, tenemos el resto de variables que se usan en el protocolo: el parámetro de seguridad, L, g, ...
Las variables que hemos utilizado para almacenar los números enteros grandes son de tipo java.math.BigInteger y los polinomios utilizados en el esquema de compartición de secretos de Shamir se han instanciado mediante la clase DKPol que hemos definido nosotros.

Esta clase provee las funciones para que se ejecuten las tres fases del protocolo. Proporciona los mecanismos para que se generen los factores p_i y q_i, los polinomios que permitirán distribuir los factores, los métodos para generar $v_i(x)$ y $v(k)$ e implementa el método de Lagrange para obtener el módulo N. Para realizar el test de doble primalidad proporciona los métodos para obtener g, el símbolo de Jacobi $(\frac{g}{N})$ y para generar los Q_i y verificar la primalidad. Para computar d_i proporciona los métodos necesarios para obtener $\phi(N)$ y que finalmente nos permitan computar d_i. Además también se han implementado métodos para poder generar de forma distribuida el exponente público d (que nos permite comprobar, al final del proceso, que las claves $E_{(e,N)}$ y $D_{(d,N)}$ computadas son correctas).

4.3.3. La interfaz de comunicaciones

En el archivo DKCom.java se encuentra la clase DKCom encargada de la comunicación entre nodos. Como variables miembro tiene toda la información referente al host en el que se encuentra: dirección IP, dirección del grupo multicast, puertos de conexión con el grupo multicast y para los envíos unicast, y finalmente, los sockets de conexión.

La clase nos proporciona todos los métodos necesarios para comunicarnos con el grupo multicast y para permitir la comunicación con los nodos a través de conexiones unicast. El grupo multicast al que nos unimos es el 239.253.0.1, que pertenece al rango de direcciones para grupos reservados a *sites*. Para acceder a este grupo se usa el puerto 5555. En las comunicaciones nodo a

nodo, se utiliza el puerto 5556 y la dirección IP del resto de nodos se obtiene extrayendo la dirección del campo origen de los datagramas que se reciben en multicast.

Como se ha comentado antes, esta clase utiliza un mecanismo de *timeouts* para detectar cuando un nodo ha abandonado el protocolo por cualquier motivo. Todas las lecturas que se hagan, tanto del grupo multicast como en unicast, implementan este mecanismo de *timeout*, excepto la lectura del mensaje de inicio del protocolo. Cuando los nodos se unen al protocolo se quedan a la espera de que llegue un mensaje con la petición de inicio del protocolo (lleva como identificador de estado el número 0), que no se sabe cuando puede llegar. Al producirse un *timeout* se lanza una excepción del tipo *SocketTimeoutException*, que se recoge mediante una estructura *try_ catch*, pero vuelve a ser relanzado para que lo coja la clase DKGen, que es la que lleva el control del proceso y de los nodos que participan. Cuando la clase DKGen coge esta excepción identifica cuál ha sido el nodo que no ha respondido y borra toda la información que se había recibido de él, de forma que no interfiera en el protocolo.

Esta clase también cuenta con una función que permite gestionar la información del resto de nodos participantes en el protocolo. La información se guarda en una estructura de datos denominada DKInfoNodos, que se explica a continuación. Cuando el host quiera comunicarse con un nodo en unicast, leerá la información del nodo contenida en estas estructuras.

4.3.4. Las nuevas estructuras de datos

Para el desarrollo de la aplicación hemos tenido que implementar nuevas estructuras de datos que nos simplifiquen la tarea. Se han implementado dos nuevas estructuras: una clase que define los polinomios que se usan en el esquema de compartición de secretos de Shamir, y que utilizaremos para esconder en ellas los secretos a transmitir. También define las operaciones que podemos realizar sobre estos polinomios. Y, en segundo lugar, una clase que almacena la información sobre los nodos, y que nos permite comunicarnos con ellos.

Polinomios

El protocolo de compartición de secretos de Shamir está basado en la interpolación de polinomios. De esta forma, los servidores eligen un polinomio $f \in \mathbb{Z}_q[x]$ de grado t, donde el coeficiente $f(0)$ contiene el secreto a compartir.

Así pues, hemos diseñado la clase DKPol (en DKPol.java) que nos permite construir estos polinomios y trabajar con ellos. Estos polinomios vienen definidos por su grado y por un array que contiene los valores de cada uno de los coeficientes del polinomio. El constructor de la clase se encarga de generar un polinomio aleatorio (los coeficientes $f(i)$, tal que $1 \leq i \leq$ grado, se generan de forma aleatoria) del grado especificado, y almacena en el coeficiente $f(0)$ el secreto a compartir (en el caso de los polinomios $z_i(x)$ y $z'_i(x)$ el coeficiente de grado 0 toma el valor 0).

Todos los valores aleatorios de este proyecto se han obtenido utilizando la clase java.security.SecureRandom, que es una extensión de la clase java.util.Random. Esta clase provee generadores de números aleatorios criptográficamente fuertes (RNG). Un número aleatorio criptográficamente fuerte es el que cumple con los tests de generadores estadísticos de números aleatorios especificados en FIPS 140-2 [Nat02]. Adicionalmente, SecureRandom produce salidas no deterministas por lo que se requiere que la "semilla" (*seed*) generadora sea impredecible, para que las salidas sean secuencias criptográficamente fuertes como se describe en el RFC 1750 [ECS94].

Información sobre los nodos

Para gestionar la información necesaria para que un nodo se comunique con los demás también hemos diseñado una nueva clase: DKInfoNodos. La información que guarda sobre los nodos es: identificador, dirección IP, puerto de comunicación y una variable que indica si el nodo está corrupto o participa correctamente en el protocolo. Esta clase también provee todos los mecanismos necesarios para poder leer la información almacenada y modificarla en caso de que sea necesario.

4.3.5. Raíces cuadradas de números enteros grandes

La API de Java no nos ofrece ninguna solución para realizar raíces cuadradas de elementos de la clase BigInteger. Por lo tanto, como a lo largo del protocolo se necesitan hacer cálculos de raíces cuadradas en varios puntos diferentes del proceso, decidimos crear una clase con este cometido. La clase DKBigSqrt implementa el método de Newton (también denominado por algunos autores como método de Heron de Alejandría) para obtener raíces cuadradas mediante aproximaciones sucesivas. Este método aplica la ecuación recurrente 4.3 para calcular $\sqrt{n}$. Asumiremos $x_0 = 1$.

$$x_{k+1} = \frac{1}{2}(x_k + \frac{n}{x_k}) \tag{4.3}$$

El algoritmo para calcular la raíz cuadrada de un número n mediante el método de Newton es el siguiente:

1. Se elige una aproximación inicial g. Por simplicidad, hemos elegido $g = 1$.
2. Calculamos $nuevo_g = \frac{(\frac{n}{g}+g)}{2}$
3. Repetimos el algoritmo desde el punto 2, tomando como g el valor recién obtenido. El proceso se repite hasta que dos resultados consecutivos sean iguales.

4.4. Versiones alternativas

4.4.1. Implementación con threads

Con la intención de intentar mejorar el rendimiento del sistema hemos implementado una versión que realiza la computación de claves en paralelo, utilizando una técnica similar a la que se describe en el punto 4.3 de [BF97]. De esta forma se lanzan un número i de threads comunicados entre ellos, que ejecutan el protocolo hasta que uno de ellos consigue computar unas claves RSA correctas. De esta forma conseguimos que el número de iteraciones que se realizan para computar N disminuya de forma drástica. Supongamos que ℓ es el número de iteraciones de una implementación con un único thread.

La ecuación 4.4 nos muestra el número de iteraciones esperado en una implementación de i threads.

$$iteraciones = \frac{1}{1 - (1 - \frac{1}{n})^i} \tag{4.4}$$

Para implementar esta versión hemos realizado cambios en la clase DKGen vista en el apartado anterior. En primer lugar hemos creado una clase DKThread que hereda de la clase Thread y que está anidada dentro de la clase DKGen. Este hecho nos permitirá que los threads puedan coordinarse utilizando las variables miembro de la clase DKGen. Dentro de la clase DKThread controlamos todo el proceso de generación de las claves al igual que se hacía anteriormente en la clase DKGen. La implementación es básicamente igual, con ciertas modificaciones que permiten a los threads coordinarse y si saber si alguno de los demás threads ha conseguido un resultado correcto, por lo que ya no hace falta que continue con su ejecución. Otra pequeña modificación que hemos tenido que hacer es controlar los puertos que usa cada uno de los threads. Esto se controla sumando un número de puerto base con el identificador del thread.

La clase DKGen ahora se encarga solamente de la ejecución de los threads y de su coordinación. Mediante las variables booleanas de esta clase podemos hacer que un thread informe a los otros de que ha conseguido acabar el protocolo de forma satisfactoria y que pueden finalizar su ejecución. Además se ha creado una clase auxiliar, llamada DKGenTh, que es la clase que contiene el método *main()* que se encarga de crear un objeto de la clase DKGen.

Los resultados obtenidos con esta implementación se muestran en el capítulo 5. También se realiza una pequeña comparación con los resultados obtenidos en la implementación sin threads.

CAPÍTULO 5

Pruebas y resultados

5.1. Análisis de resultados

5.1.1. Resultados de la implementación sin threads

Se han realizado un gran número de pruebas para comprobar el correcto funcionamiento de la aplicación, así como su rendimiento. Las pruebas se han realizado utilizando ordenadores Pentium IV a 2 GHz. que incorporan el sistema operativo Fedora Core 2.6.10-1.771. Los ordenadores están conectados mediante una red Ethernet 10/100 Mbps.

Los tiempos de ejecución obtenidos son, de término medio, bastante pobres. La causa para la obtención de estos tiempos reside en la complejidad de la generación de N de forma distribuida. Los factores p y q que generan N, se obtienen mediante la suma de valores aleatorios, y la probabilidad de que el sumatorio de estos valores aleatorios generen un número primo es inversamente proporcional al tamaño de los números. Según Boneh [BF97], la probabilidad de que un número aleatorio de n bits sea primo es de $1/n$. Por lo tanto, la probabilidad de que los dos números sean primos es de $1/n^2$, lo que implica que se tengan que ejecutar n^2 iteraciones de media para conseguir un módulo N válido. El tamaño de las claves que genera nuestro protocolo es de

512 bits (el tamaño mínimo recomendado por Frankel et al. para las claves es de 400 bits, pero nosotros hemos utilizado valores de 512 bits, que ofrecen una mayor seguridad y consideramos que debía ser el valor mínimo a la hora de realizar la implementación), por lo que el número medio de iteraciones que debería realizar nuestro protocolo sería de unas 262144. Como podremos observar en la tabla 5.1, las optimizaciones introducidas en nuestra aplicación han hecho que las medias sean considerablemente inferiores a la media teórica.

Otra de las causas para la obtención de estos resultados es que en esta primera implementación no se han incorporado mejoras en los algoritmos de generación de claves (las cuales analizaremos en la siguiente sección). Finalmente, también hay que destacar que el hecho de que el algoritmo de Frankel utilice como clave pública $e = 3$ provoca que se tengan que desechar una cantidad notable de módulos correctos porque $mcd(e, \phi(N)) \neq 1$, lo que provoca que el tiempo de ejecución se multiplique. Para solucionar este aspecto se debería intentar implementar una generalización del esquema propuesto por Frankel, que permita utilizar claves de tamaño mayor. También se podría utilizar el otro esquema de generación de claves propuesto por Frankel, que permite utilizar un exponente público e de cualquier tamaño, pero que es de una complejidad superior.

Los resultados obtenidos tras la ejecución de la aplicación básica (sin threads) se muestran en tabla 5.1. En ellos mostramos los tiempos de ejecución óptimo (mejor resultado obtenido) y medio para la generación de claves por parte de 3 y 4 servidores.

	n	Tam. claves	Iteraciones			Tiempos			
			N inc.	d inc.	total	N	TDP	d	**total**
Ejecución óptima	3	512b	876	0	877	16s 710ms	26s 526ms	22ms	**43s 279ms**
Ejecución media	3	512b	8019	1	8021	2m 28s 444ms	3m 0s 671ms	31ms	**5m 29s 280ms**
Ejecución óptima	4	512b	5104	0	5105	1m 41s 634ms	1m 8s 390ms	29ms	**2m 50s 130ms**
Ejecución media	4	512b	16060	2	16063	5m 13s 708ms	5m 37s 972ms	42ms	**11m 51s 620ms**

Tabla 5.1: *Resultados de la ejecución de la aplicación básica*

donde:

- n: número de nodos participantes
- Tam. claves: tamaño de las claves en bits
- N inc.: número de módulos N incorrectos
- d inc.: número de exponentes d incorrectos
- N: generación distribuida de N
- TDP: test de doble primalidad
- d: generación distribuida de d

La cantidad de datos transmitidos (carga total que circula por la red) de término medio para la ejecución con 3 nodos es de 36596 KB (12198 KB por nodo), que corresponde a una media de 1'5 KB por cada iteración de cada uno de los nodos S_i. Como se puede observar, la cantidad de información transmitida por iteración del protocolo no es muy grande, no obstante, debido al número tan elevado de iteraciones que se realizan la cantidad total de datos transmitidos es bastante elevada. Para la implementación de cuatro servidores la media total de datos transmitidos por la red asciende a 104452 KB (26113 KB por nodo). Por suerte, no se necesita almacenar toda esta información, si no que en cada iteración sobrescribimos los datos de la iteración anterior, permitiendo que la cantidad de memoria necesaria sea pequeña.

Si comparamos los resultados que hemos obtenido, con los de otras implementaciones completamente distribuidas, como por ejemplo la de Malkin, Wu y Boneh [MWB99] (es la única implementación en la que se hace un análisis completo de los resultados), nos damos cuenta de que los resultados obtenidos son sustancialmente peores. Las razones por las que Malkin et al. obtienen mejores resultados son tres: (a) su test de primalidad consiste en aplicar una iteración del test de Fermat, mientras que en nuestro caso se aplican del orden de 512 iteraciones del test de doble primalidad que es más complejo. Esto hace que en Malkin los tiempos del test de primalidad sean muy pequeños mientras que en nuestra implementación ocupan aproximadamente la mitad del tiempo de procesamiento. (b) nuestra implementación tiene una complejidad mayor dado que implementa más mecanismos de seguridad. (c) el número de iteraciones necesarias para encontrar un módulo

N válido es de 119, mientras que en nuestro caso la media era de 8021. Esto provoca que nuestra implementación sea mucho más lenta (unas 68 veces más lenta) que en el caso de Malkin. Si se pudieran implementar mejoras que redujesen la cantidad de iteraciones necesarias para obtener un módulo N válido, se podrían reducir drásticamente los tiempos de ejecución. Esto también nos podría permitir la utilización de claves de un tamaño mayor a 512 bits (1024 - 2048).

Para probar la capacidad de reacción del protocolo ante fallos inesperados se hicieron pruebas en las que un nodo abandonara de forma repentina el protocolo (se forzaba el abandono de uno de los nodos incluyendo una instrucción *System.exit()* en medio de su código). En todas las pruebas realizadas los nodos detectaron el abandono de uno de los nodos y fueron capaces de identificarlo correctamente, anulando así todos los valores recibidos por este nodo. En el caso de que el número de nodos restantes fuese menor o igual al umbral t, o fuese inferior a 3, los nodos abortaban la ejecución avisando de que no se había podido llevar a cabo la ejecución por la falta de nodos correctos participando en el protocolo. Los tiempos de ejecución obtenidos se corresponden con los de la gráfica anterior (fig. 5.1) para el número de nodos restantes. Es decir, si en una implementación de 4 nodos abandonaba 1, se obtenían los tiempos correspondientes a la computación con 3 nodos (hay que tener en cuenta que aparte del tiempo de computación hay que sumar los 5 segundos que los nodos esperan hasta producir el *timeout*).

Todos los resultados obtenidos han sido comprobados. Para ello, tras obtener el módulo N, el exponente público e y la partición del exponente privado d_i, los nodos obtenían el exponente privado d, y cifraban y descifraban (ver ecuaciones 5.1) un valor dado. Si el valor original era igual que el descifrado, las claves obtenidas son correctas. El porcentaje obtenido de ejecuciones correctas es del 100 %.

$$E_{(e,N)}(x) = x^e \bmod N \qquad D_{(d,N)}(x) = y^d \bmod N \tag{5.1}$$

5.1.2. Resultados de la implementación multithread

A pesar de que la implementación con threads pretendía mejorar los resultados obtenidos por la versión normal, no se ha conseguido tal efecto y las ejecuciones han sido considerablemente más lentas. Al aumentar el número de threads hemos conseguido que se aumente el *throughput* general del sistema, es decir, el número de iteraciones que realizan entre todos los threads (ver tabla 5.3). Sin embargo, esto no ha servido para disminuir el número de iteraciones necesarias para computar las claves, si no que han aumentado considerablemente. Lo cual se traduce en un aumento sustancial de los tiempos.

Los resultados obtenidos para la ejecución con 2, 4 y 6 threads se muestran en la figura 5.2. En la figura 5.3 se realiza una comparativa entre el rendimiento de todas las diferentes implementaciones vistas.

	n	th	Iteraciones			Tiempos			
			N inc.	d inc.	total	N	TDP	d	**total**
Ejecución óptima	3	2	1931	0	1931	1m 11s 130ms	47s 729ms	54ms	**1m 58s 995ms**
Ejecución media	3	2	46155	1	46157	21m 8s 421ms	14m 25s 758ms	284 ms	**35m 34s 280ms**
Ejecución óptima	3	4	9398	0	9399	11m 21s 238ms	6m 50s 978ms	96ms	**18m 12s 498ms**
Ejecución media	3	4	17787	0	17788	22m 48s 277ms	15m 43s 722ms	90ms	**36m 9s 388ms**
Ejecución óptima	3	6	16740	0	16741	33m 27s 663ms	17m 55s 162ms	128 ms	**51m 23s 187ms**
Ejecución media	3	6	23147	0	23148	43m 40s 646ms	23m 7s 650ms	165 ms	**66m 48s 832ms**

Tabla 5.2: *Resultados de la ejecución de la aplicación multithread*

donde:

- n: número de nodos participantes
- th: número de threads utilizados
- N inc.: número de módulos N incorrectos
- d inc.: número de exponentes d incorrectos
- N: generación distribuida de N
- TDP: test de doble primalidad

- d: generación distribuida de d

nodos	threads	throughput (iters/sec)	throughput general
3	-	24'36	24'36
4	-	22'57	22'57
3	2	21'62	43'24
3	4	8'19	32'76
3	6	5'77	34'62

Tabla 5.3: *Rendimiento del sistema*

donde:

- throughput: número de iteraciones que realiza cada thread por segundo
- throughput general: número de iteraciones por segundo que realizan entre todos los threads

Un factor a tener en cuenta, es que, aunque Boneh y Franklin recomienden el uso de multiples threads para obtener mejoras en la ejecución, esto no es del todo extrapolable a nuestra aplicación. El modelo de Boneh y Franklin es sincronizado, por lo que se usan los momentos en los que un thread está a la espera para sincronizarse para ceder el uso del procesador a otro thread. De esta forma se consigue reducir notablemente el overhead de sincronización. En nuestro caso utilizamos un modelo asíncrono, por lo que los threads cedían el uso del procesador (función *yield()*) mientras esperaban la llegada de los paquetes de datos a través de la red. Sin duda que las mejoras que haya podido introducir esta implementación no han podido contrarrestar el empeoramiento producido por la mayor complejidad de esta implementación respecto a la normal.

5.2. Algoritmos de mejora del rendimiento

A continuación describiremos algunos de los algoritmos que se podrían aplicar en futuras implementaciones para mejorar el rendimiento del sistema. La implementación llevada a cabo por Frankel et al. en la que hemos basado el desarrollo de este proyecto no hace referencia a estas mejoras. No

obstante, la base de este protocolo es el esquema descrito por Boneh y Franklin, por lo que podrían ser implementadas en nuestro protocolo las mejoras descritas en dicho documento. Estos algoritmos se describen más en profundidad en [BF97] y también se muestran los resultados obtenidos tras la implementación de estas técnicas en [MWB99].

Distributed sieving Esta técnica permite asegurar que los elementos p_i y q_i no son divisibles por números primos pequeños, de tal forma que $p = \sum p_i$ y $q = \sum q_i$ tampoco sea divisible por números primos pequeños. Esta técnica no revela ninguna información sobre p_i y q_i al resto de nodos, y permite mejorar 10 veces el rendimiento del sistema (pruebas realizadas para claves de 1024 bits).

Test de candidatos en paralelo Este método permite reducir el *overhead* de sincronización entre los nodos y buscar varios candidatos de forma paralela, de tal forma que cuando uno de los threads consigue obtener un módulo correcto avisa a los demás para que paren su ejecución. Los mejores resultados se obtienen cuando los tiempos de sincronización son altos, puesto que se aprovechan los momentos de espera de la sincronización para pasar la ejecución a otro thread. En nuestra aplicación, sin embargo, al ser un modelo parcialmente asíncrono que no necesita de sincronización, no se aprecian estas mejoras. La adaptación a esta técnica que hemos implementado ha sido descrita en el apartado 4.4.1.

Divisiones de prueba en paralelo Una vez obtenido el módulo N, y antes de ejecutar el test de primalidad, los nodos pueden realizar una serie de divisiones de prueba para comprobar que N no es divisible por números primos pequeños. Si cada uno de los nodos se encarga de comprobar que N no es divisible por un conjunto definido de números, los nodos pueden dividir el tiempo necesario para realizar todas las comprobaciones entre el número de nodos.

Balance de cargas El servidor S_1 tiene una carga de trabajo superior a la del resto. Para equilibrar esto se puede hacer que en cada thread el servidor 1 sea un nodo diferente. En el test de primalidad el servidor

S_1 computa $Q_1 = g^{(N+1-p_1+q_1)/4}$ mód N donde N tiene un tamaño de $2n$. Mientras tanto, el resto de nodos calcula $Q_i = g^{(p_i+q_i)/4}$ mód N donde p_i+q_i tiene un tamaño de n bits. Esta diferencia puede provocar que S_1 pueda llegar a tardar hasta el doble de tiempo en realizar el test de primalidad.

Filtro de Fermat Evita que se realice el test de doble primalidad para todos los candidatos. De esta forma tan sólo se ejecutará el test de primalidad a aquellos candidatos que superen el test de Fermat: $g^{N+1-p-q} = g^{N+1-\sum(p_i+q_i)} \equiv 1 \pmod{N}$. Es más, en la implementación realizada por Malkin et al. [MWB99], ni siquiera implementan el test de doble primalidad. El filtro de Fermat tiene el problema de que algunos valores específicos de N pueden pasar el test, no obstante la probabilidad de que se compute uno de estos valores es extremadamente pequeña ($1/10^{40}$).

Evitar la deceleración cuadrática El hecho de que p y q se generen a la vez provoca que se produzca una deceleración cuadrática en comparación con los sistemas no distribuidos en los que un único usuario genera las claves. Esto se puede solucionar haciendo que N se compute como $N = p_1q_2(r_1+r_2+r_3)$, donde S_1 tiene (p_1, r_1) y p_1 es un número primo de n bits. S_2 posee q_2, p_2 donde q_2 es también un número primo de n bits. Finalmente, S_3 tiene r_3. El número de pruebas que se han de realizar hasta conseguir que $(r_1+r_2+r_3)$ sea primo es igual a las que ha de realizar un sistema monousuario (como hemos visto anteriormente, aproximadamente $1/n$). Además se puede asegurar que ninguna de las partes conoce la factorización completa de N (aunque sí de forma parcial). No obstante, esta aproximación cuenta con ciertos problemas. El primero es que para asegurar una privacidad t, N debe ser el producto de $t+2$ primos. Otro problema sería que ahora el test de primalidad debería comprobar que N es el producto de tres números primos en vez de dos.

CAPÍTULO 6

Conclusiones

6.1. Repaso de objetivos

Los objetivos de planteados al principio de este proyecto se han cumplido de forma bastante satisfactoria. Por un lado se ha realizado un estudio que refleja de forma fiel el estado del arte de los criptosistemas distribuidos, proporcionando así una base teórica suficiente para afrontar el problema de la construcción de una autoridad de certificación distribuida en el futuro. Asimismo, se ha desarrollado un sistema de generación de claves RSA distribuido, estableciendo así la infraestructura base de la autoridad de certificación.

En lo referente a la seguridad, se han implementado mecanismos que nos permiten tener un protocolo fiable aún en presencia de cierto tipo de adversarios y de errores en los nodos participantes. Este protocolo es completamente distribuido, y no necesita la participación de un servidor central de confianza que genere las claves y distribuya las particiones pertinentes. De esta forma, hemos conseguido un protocolo más seguro, pues la seguridad de las claves está distribuida entre un conjunto de nodos y no reside en uno sólo. No obstante, el hecho de distribuir todo el protocolo también nos genera el problema del aumento de tiempos de ejecución, que no hubiésemos

tenido en un modelo centralizado. El protocolo creado también es tolerante a fallos, permitiendo que el protocolo compute las claves deseadas aún habiendo nodos que dejen de participar en el protocolo de forma repentina. Define un esquema de criptografía umbral, impidiendo que ninguna coalición de como máximo t nodos pueda obtener la suficiente información para poder factorizar N. El protocolo generado no es tan robusto ante los diferentes ataques de adversarios como el esquema de Frankel et al. [FMY98], pero se ha reducido su complejidad y se ha dejado el problema de la utilización de mecanismos de *commitment* abierto, por si en posteriores trabajos se quiere plantear la posibilidad de implementarlos. Aunque no se han implementado estos mecanismos, el protocolo se ha diseñado de tal forma que permitan una fácil integración en el futuro.

Los tiempos de ejecución, y el rendimiento del algoritmo son francamente mejorables. Se han analizado los resultados obtenidos, viendo que las razones de la obtención de estos tiempos residen en la complejidad inherente del propio algoritmo. Y simultáneamente, se han descrito posibles soluciones por si futuras implementaciones intentan mejorar el rendimiento del sistema. De todos modos, la viabilidad del sistema se pone en entredicho por los resultados obtenidos. En el caso de querer implementar el proyecto en un entorno real necesitaríamos que los nodos participantes tuviesen un alto rendimiento, no pudiendo formar parte de la autoridad de certificación dispositivos que tengan una potencia limitada, como por ejemplo, PDAs o teléfonos móviles. Además, la red ad-hoc tendría que tener un ancho de banda bastante grande para poder soportar las tasas de transferencia tan elevadas de la aplicación. Por lo tanto, el ámbito en el que se podría instalar nuestro sistema se reduciría a algún tipo de entorno controlado, como por ejemplo, una convención donde usuarios con dispositivos más potentes serían los únicos que podrían formar la autoridad de certificación (no obstante, el resto de usuarios podría hacer uso de ella).

6.2. Formación académica

Aparte de los objetivos del estudio y desarrollo de la aplicación también hay que tener en cuenta los objetivos formativos alcanzados con la realiza-

ción de este proyecto. Los conocimientos previos sobre criptología y teoría de números nos permitieron afrontar el proyecto con seguridad y sin una gran dificultad, pero a la hora de tener que afrontar un aspecto nuevo para nosotros como son los criptosistemas distribuidos tuvimos que indagar y mejorar en gran medida nuestros conocimientos sobre estas materias. Por un lado nos hemos encontrado con la gran dependencia matemática que tienen estos algoritmos, que nos ha permitido conocer propiedades que hasta ahora nos eran desconocidas (enteros de Blum, símbolos de Jacobi y Legendre, interpolaciones de Lagrange, ...) pues los protocolos no distribuidos no hacen uso de estas propiedades. Por otro lado, hemos mejorado los conocimientos criptográficos previos, ahondando en nuevos campos, que nos han dado una visión completamente diferente a la que conocíamos de antemano. Ha sido también de gran interés encontrarnos una gran variedad de mecanismos (mecanismos de *commitment*, de conocimiento cero, de transformación de la representación, de consistencia encadenada, ...) que actúan de forma complementaria a los protocolos criptográficos y que los dotan de nuevas características, como por ejemplo, mayor seguridad, robustez, proactividad, eficiencia, tolerancia a fallos.

6.3. Líneas de continuación

Este proyecto cuenta con dos líneas de continuación obvias. Una posible línea sería la de intentar desarrollar un sistema de generación de claves que mejore el rendimiento de la implementación descrita en este documento. Como se ha visto en el capítulo 5, parte de culpa de la obtención de tiempos tan elevados se debe a que el número de iteraciones para computar N es muy alta. Por lo tanto, se podrían implementar las técnicas que se han descrito en ese mismo capítulo para intentar reducir el número de iteraciones que realiza el protocolo y mejorar el funcionamiento global del sistema.

La segunda línea de continuación consistiría en desarrollar la autoridad de certificación distribuida partiendo del sistema de generación de claves que hemos implementado. Para ello, se tendrían que implementar algoritmos distribuidos de firmado y verificación. A partir del estudio visto en el capítulo 2, nosotros recomendamos la implementación de Damgård [DK00] pues nos

parece la más completa y está diseñada para ser complemento del protocolo de Frankel en el que hemos basado el desarrollo de nuestra aplicación. A continuación, sería necesario definir protocolos que gestionen todos los servicios que ofrezca la autoridad de certificación, así como algoritmos que permitan la gestión interna de la CA (protocolos de adherencia de nuevos nodos a la CA, protocolos de intercambio de información entre nodos, etc.).

Bibliografía

[ACS02] Joy Algesheimer, Jan Camenisch, and Victor Shoup, *Efficient computation modulo a shared secret with application to the generation of shared safe-prime products*, CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology (London, UK), Springer-Verlag, 2002, pp. 417–432.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

[BF97] Dan Boneh and Matthew Franklin, *Efficient generation of shared RSA keys*, Lecture Notes in Computer Science **1294** (1997), 425+.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson, *Completeness theorems for non-cryptographic fault-tolerant distributed computation*, STOC '88: Proceedings of the 20th annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 1988, pp. 1–10.

[Bon98] Dan Boneh, *The decision Diffie-Hellman problem*, Lecture Notes in Computer Science **1423** (1998), 48–63.

[Bra05] Phillip G. Bradford, *Notes on one-way functions*, 2005.

[Cac05] Christian Cachin, *Distributed criptography*, IBM Zurich Research Lab, 2005.

[Can00] Ran Canetti, *Security and composition of multiparty cryptographic protocols*, Journal of Cryptology: the journal of the International Association for Cryptologic Research **13** (2000), no. 1, 143–202.

[CGH00] Dario Catalano, Rosario Gennaro, and Shai Halevi, *Computing inverses over a shared secret modulus*, Lecture Notes in Computer Science **1807** (2000).

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup, *Random oracles in Constantinople: practical asynchronous byzantine agreement using cryptography (extended abstract)*, PODC '00: Proceedings of the 19th annual ACM symposium on Principles of distributed computing (New York, NY, USA), ACM Press, 2000, pp. 123–132.

[CMTW] Chris Crutchfield, David Molnar, David Turner, and David Wagner, *On-line/off-line threshold signatures without trusted dealers*, University of California.

[Des87] Yvo Desmedt, *Society and group oriented cryptography: A new concept*, CRYPTO '87: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (London, UK), Springer-Verlag, 1987, pp. 120–127.

[DF90] Ivo Desmedt and Yair Frankel, *Threshold cryptosystems*, Advances in Cryptology - proceedings of CRYPTO '89 (Milwaukee, WI, USA), 1990, pp. 307–315.

[DF91] Yvo Desmedt and Yair Frankel, *Shared generation of authenticators and signatures (extended abstract)*, CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology (London, UK), Springer-Verlag, 1991, pp. 457–469.

[DK00] Ivan Damgård and Maciej Koprowski, *Practical threshold RSA signatures without a trusted dealer*, 2000.

[ECS94] Donald E. Eastlake, Stephen D. Crocker, and Jeffrey I. Schiller, *RFC 1750 - randomness recommendations for security*, 1994.

[ES98] Shawna M. Eikenberry and Jonathan P. Sorenson, *Efficient algorithms for computing the Jacobi symbol*, J. Symb. Comput. **26** (1998), no. 4, 509–523.

[Eug05] Patrick T. Eugster, *DICTATE: Distributed certification authority with probabilistic freshness for ad hoc networks*, IEEE Trans. Dependable Secur. Comput. **2** (2005), 311–323.

[Fel87] Paul Feldman, *A practical scheme for non-interactive verifiable secret sharing*, IEEE Symposium on Foundations of Computer Science, 1987, pp. 427–437.

[FGMY97] Yair Frankel, Peter Gemmell, Philip D. MacKenzie, and Moti Yung, *Optimal-resilience proactive public-key cryptosystems*, FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (Washington, DC, USA), IEEE Computer Society, 1997, p. 384.

[Fis01] Marc Fischlin, *Trapdoor commitment schemes and their applications*, Ph.D. thesis, Johann Wolfgang Goethe - University of Frankfurt, December 2001.

[Fis05] M. J. Fisher, *The Legendre and Jacobi symbols*, 2005.

[FMY98] Yair Frankel, Philip D. MacKenzie, and Moti Yung, *Robust efficient distributed RSA-key generation*, STOC '98: Proceedings of the 13th annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 1998, pp. 663–672.

[FMY02] Yair Frankel, Philip MacKenzie, and Moti Yung, *Adaptively secure distributed public-key systems*, Theory in Computer Sciences **287** (2002), no. 2, 535–561.

[Fro02] Agustín Froufe, *Java 2 manual de usuario y tutorial*, 3 ed., Ra-Ma Editorial, Madrid, España, June 2002.

[FS90] Uriel Feige and Adi Shamir, *Zero knowledge proofs of knowledge in two rounds*, CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology (London, UK), Springer-Verlag, 1990, pp. 526–544.

[FS01] Pierre-Alain Fouque and Jacques Stern, *Fully distributed threshold RSA under standard assumptions*, ASIACRYPT '01: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security (London, UK), Springer-Verlag, 2001, pp. 310–330.

[GHY86] Zvi Galil, Stuart Haber, and Moti Yung, *Symmetric public-key encryption*, Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85 (New York, NY, USA), Springer-Verlag New York, Inc., 1986, pp. 128–137.

[Gil99] Niv Gilboa, *Two party RSA key generation*, CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (London, UK), Springer-Verlag, 1999, pp. 116–129.

[GJKR96] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin, *Robust threshold DSS signatures*, Lecture Notes in Computer Science **1070** (1996), 354.

[GJKR99] ______, *Secure distributed key generation for discrete-log based cryptosystems*, Lecture Notes in Computer Science **1592** (1999), 295.

[GJKR03] ______, *Revisiting the distributed key generation for discrete-log based cryptosystems*, 2003.

[GMR85] Safhi Goldwasser, Silvio Micali, and Charles Rackoff, *The knowledge complexity of interactive proof-systems*, STOC '85: Pro-

ceedings of the 7th annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 1985, pp. 291–304.

[Gor85] John A. Gordon, *Strong primes are easy to find*, Proc. of the EUROCRYPT 84 workshop on Advances in cryptology: theory and application of cryptographic techniques (New York, NY, USA), Springer-Verlag New York, Inc., 1985, pp. 216–223.

[Har94] Lein Harn, *Group-oriented (t, n) threshold digital signature scheme and digital multisignature*, IEE Proc. Comput. Digit. Tech., 1994.

[IS90] Ingemar Ingemarsson and Gustavus J. Simmons, *A protocol to set up shared secret schemes without the assistance of mutually trusted party*, EUROCRYPT '90: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology (New York, NY, USA), Springer-Verlag New York, Inc., 1990, pp. 266–282.

[Jar01] Stanislaw Jarecki, *Efficient threshold cryptosystems*, Ph.D. thesis, 2001, Supervisor - Shafi Goldwasser.

[Knu98] Jonathan Knudsen, *Java cryptography*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[Lam86] Leslie Lamport, *LaTeX: a document preparation system*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Mül04] Siguna Müller, *On the computation of square roots in finite fields*, Des. Codes Cryptography **31** (2004), no. 3, 301–312.

[Moo05] Ross Moore, *Interpolation for polynomial parametric curves*, 2005.

[MR97] Dahlia Malkhi and Michael Reiter, *Byzantine quorum systems*, 1997, pp. 569–578.

[MS04] Michael A. Marsh and Fred B. Schneider, *Codex: A robust and secure secret distribution system*, IEEE Trans. Dependable Secur. Comput. **1** (2004), no. 1, 34–47.

[MWB99] Michael Malkin, Thomas Wu, and Dan Boneh, *Experimenting with shared generation of RSA keys*, Proceedings of the Internet Society's 1999 Symposium on Network and Distributed System Security (SNDSS), 1999, pp. 43–56.

[Nat02] National Institute of Standards and Technology, *Security requirements for cryptographic modules*, December 2002.

[NO00] Koji Nakano and Stephan Olariu, *Randomized initialization protocols for ad hoc networks*, IEEE Trans. Parallel Distrib. Syst. **11** (2000), no. 7, 749–759.

[oMAtC] Research Group on Mathematics Applied to Cryptography, *Some trends for future research in distributed*, Tech. report, Dept. Matemàtica Aplicada IV, Universitat Politècnica de Catalunya.

[OPHS00] Tobias Oetiker, Hubert Partl, Irene Hyna, and Elisabeth Schlegl, *The not so short introduction to LaTeX 2ε*, September 2000.

[Ped91] Torben P. Pedersen, *A threshold cryptosystem without a trusted party*, Lecture Notes in Computer Sciences; 547 in Advances in Cryptology — EUROCRYPT'91 (Brighton, UK), Springer-Verlag, April 1991, pp. 522–526.

[Ped92] ———, *Non-interactive and information-theoretic secure verifiable secret sharing*, CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology (London, UK), Springer-Verlag, 1992, pp. 129–140.

[PS98] Guillaume Poupard and Jacques Stern, *Generation of shared RSA keys by two parties*, ASIACRYPT '98: Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security (London, UK), Springer-Verlag, 1998, pp. 11–24.

[Rab98] Tal Rabin, *A simplified approach to threshold and proactive RSA*, CRYPTO '98: Proceedings of the 18th Annual International

Cryptology Conference on Advances in Cryptology (London, UK), Springer-Verlag, 1998, pp. 89–104.

[Sae96] Mugino Saeki, *Elliptic curve cryptosystems*, 1996.

[Sao95] Yannick Saouter, *A new method for the generation of strong prime numbers*, Tech. Report 2657, Institut National de Recherche en Informatique et en Automatique, September 1995.

[Sco01] Hugo D. Scolnik, *Métodos modernos de criptografía e identificación remota*, 2001.

[Sha48] Claude E. Shannon, *The mathematical theory of communication*, The Bell System Technical Journal, 1948.

[Sha79] Adi Shamir, *How to share a secret*, Commun. ACM **22** (1979), no. 11, 612–613.

[Sho00] Victor Shoup, *Practical threshold signatures*, Lecture Notes in Computer Science **1807** (2000), 207–220.

[Sil97] Robert D. Silverman, *Fast generation of random, strong RSA primes*, RSA Laboratories' Cryptobytes **3** (1997), no. 1.

[SS93] Jeffrey Shallit and Jonathan Sorenson, *A binary algorithm for the Jacobi symbol*, SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation) **27** (1993), no. 1, 4–11.

[Tan02] Caimu Tang, *ECDKG: A distributed key generation protocol based on elliptic curve discrete logarithm.*

[TLL] Chunming Tang, Zhuojun Liu, and Jinwang Liu, *The statistical zero-knowledge proof for Blum integer based on discrete logarithm.*

[WDBD03] Claire Whelan, Adam Duffy, Andrew Burnett, and Tom Dowling, *A Java API for polynomial arithmetic*, PPPJ '03: Proceedings of the 2nd international conference on Principles and

practice of programming in Java (New York, NY, USA), Computer Science Press, Inc., 2003, pp. 139–144.

[ZSvR00] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse, *COCA: A secure distributed on-line certification authority*, Tech. report, Ithaca, NY, USA, 2000.

Firmat: Álvaro Merino de la Concepción

Bellaterra, Juny de 2006