UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

# Criptografía en sistemas de bajos recursos

Informe de Proyecto de Grado presentado por

Santiago Jaureche y Jorge Merlino

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Tutor

Alfredo Viola

Montevideo
2002

# Resumen

Se denominan dispositivos de bajos recursos a todos aquellos artefactos que de alguna forma tienen limitadas sus capacidades ya sean de almacenamiento como de procesamiento. En este proyecto se estudiaron distintos algoritmos de encriptación y desencriptación y se investigó su posible adecuación al uso en los mencionados dispositivos. En particular nuestra atención se centró en las tarjetas inteligentes.
Además de esto se implementaron en dichas tarjetas algunos de los criptosistemas estudiados y se realizaron comparaciones entre los mismos con respecto a parámetros como memoria ocupada y tiempo de ejecución.

# Índice

# 1. Introducción

## 1.1. Definición del problema

Se denominan dispositivos de bajos recursos a todos aquellos artefactos comprendidos en el marco de la tecnología informática, que de alguna forma tienen limitadas sus capacidades ya sean de almacenamiento como de procesamiento debido a que en su mayoría son equipos portátiles por lo que cuentan con un tamaño físico muy limitado. Algunos ejemplos de los dispositivos existentes son los siguientes:

- PC de bolsillo (PC pocket).
- DBA (Data Base Administrator) remotos.
- Teléfonos celulares.
- Agendas electrónicas.
- PDA (Personal Digital Assistant).
- Tarjetas inteligentes.

En este proyecto se realizó un relevamiento de diferentes algoritmos de encriptación y desencriptación de información (que llamaremos criptosistemas) e investigamos su posible adecuación al uso en los mencionados dispositivos. En especial, de entre todos los dispositivos de bajos recursos, se investigaron en forma más detallada las tarjetas inteligentes, y dentro de ellas las *Java Cards.*

Este es un tipo particular de tarjetas inteligentes que se caracterizan por poseer integrada en su hardware una máquina virtual de Java lo cual permite que el desarrollo de las aplicaciones que corren en la misma se realice en este lenguaje. Esto, por supuesto, es una ventaja dado que no es necesario aprender desde cero un lenguaje propietario específico para poder programarlas.

A continuación se implementaron en las propias tarjetas algunos de los criptosistemas estudiados con el fin de interiorizarnos en los detalles de la programación de *Java Cards* así como en las dificultades que trae aparejada la programación de algoritmos criptográficos en ambientes de bajos recursos.

Además de esto, para poder probar los algoritmos implementados en una aplicación con cierta utilidad práctica, se desarrolló la llamada aplicación de referencia cuya función es la de autenticar a un usuario ante cierto sistema basándose en que el mismo conozca una contraseña y además posea la tarjeta asociada a la misma. Para esto también se requirió estudiar los mecanismos mediante los cuales las aplicaciones externas se comunican con los procesos que corren en la tarjeta.

Para concluir se realizaron comparaciones entre los diferentes algoritmos criptográficos implementados. Dichas comparaciones tuvieron en cuenta la memoria consumida por cada una de las implementaciones, el tiempo que tarda en ejecutarse la aplicación de referencia al realizar sus operaciones criptográficas con cada uno de dichos algoritmos y además se comparó la seguridad de las diferentes implementaciones. Las comparaciones de seguridad fueron realizadas contrastando que tan factibles eran los ataques conocidos contra los algoritmos considerando los parámetros reales de las implementaciones realizadas.

Este informe se presenta de la siguiente manera. En este primer capítulo se da una introducción a los dispositivos de bajos recursos, sus aplicaciones prácticas y el soporte criptográfico que poseen integrado; seguimos con el capítulo 2 en el cual se profundiza en particular sobre las tarjetas inteligentes y sus características específicas. Luego, el capítulo siguiente es una reseña de los criptosistemas estudiados a lo largo del proyecto y su aplicabilidad en ambientes informáticos de recursos limitados. A continuación en el capítulo 4 se detalla todo lo referido al modelo de tarjeta usado en el proyecto así como los pormenores de las diferentes implementaciones realizadas (tanto los criptosistemas como la aplicación de referencia); además es aquí donde se muestran los resultados de las comparaciones llevadas a cabo. Para finalizar, el capítulo 5 contiene las conclusiones obtenidas a partir del estudio realizado y el siguiente plantea los posibles desarrollos de este proyecto como trabajos futuros.

Luego del informe se encuentran diez apéndices complementarios los primeros cinco de los cuales describen en mayor detalle los diferentes criptosistemas estudiados. A continuación el apéndice VI trata sobre ataques específicos a algoritmos criptográficos implementados sobre tarjetas inteligentes mientras que el siguiente contiene los datos de las mediciones comparativas de tiempo y los cálculos realizados a fin de obtener intervalos de confianza para los mismos. Por último se encuentran el glosario que explica algunos de los términos usados en el documento y la bibliografía consultada.

## 1.2. Motivación

Los motivos que nos llevaron a plantear el desarrollo de este proyecto fueron primeramente un interés en la tecnología de las tarjetas inteligentes, la cual es un área relativamente nueva con muchas posibilidades para el futuro. Junto con esto, existía el interés en la criptografía, surgido en parte al realizar el curso sobre el tema que se dicta en la Facultad, por lo que intentamos unir estas dos áreas de modo de investigar en implementaciones prácticas de la criptografía sobre la plataforma de las tarjetas inteligentes.

Asimismo, dados los actuales usos reales de las tarjetas como, por ejemplo, billeteras de dinero electrónico o en sistemas de identificación (aplicaciones que requieren un alto nivel de confidencialidad y confiabilidad por parte tanto del proveedor como del usuario final), siempre es necesario tener algoritmos criptográficos implementados en las tarjetas inteligentes, por lo que el relevamiento tiene también un sentido práctico real.

En particular el proyecto se basó principalmente en las *Java Cards* dado que la facultad poseía el hardware necesario y además de esto ya existía un trabajo previo sobre el tema en un proyecto de taller V anterior, lo cual nos permitió partir de cierta base ya armada y dedicarnos un poco más al tema de la criptografía en sí, sin necesidad de aprender desde cero el funcionamiento de las tarjetas.

## 1.3. Uso de los dispositivos de bajos recursos

Este tipo de dispositivos se ha desarrollado mucho en los últimos tiempos debido al gran avance que ha tenido la relación *(capacidad de procesamiento)/tamaño* en los microprocesadores. Entre ellos se encuentran los teléfonos celulares, los equipos móviles de comunicación, agendas electrónicas personales y las tarjetas inteligentes.

Los usos que se le dan en la actualidad a estos dispositivos son muy variados y dependen en gran medida del dispositivo específico que se tenga en cuenta. En particular hay algunos, como los PDAs, que actualmente casi se puede decir que no entrarían en esta categoría ya que son prácticamente equivalentes a una PC de escritorio de hace pocos años atrás. Por esta razón sus usos son tan variados como los de una computadora personal.

Existen otros dispositivos más particulares, como ser los teléfonos celulares que, además de su funcionalidad básica, pueden incluir otras funciones como ser bases de datos de información personal, soporte del protocolo WAP (*Wireless Application Protocol*) para permitir navegación por Internet e incluso soporte de aplicaciones criptográficas con el fin de asegurar la confidencialidad de las comunicaciones (ver 1.4).

En el caso específico de las tarjetas inteligentes, en que se centrará este documento, sus usos son diversos y de hecho continuamente aparecen nuevos. A modo de reseña, algunos de los mismos son los siguientes:

- Tarjetas para cajeros automáticos o tarjetas de crédito (que a diferencia de las que se usan en nuestro país utilizan técnicas criptográficas para garantizar su autenticidad)
- Billeteras para dinero electrónico
- Identificaciones personales para autorizar acceso a lugares físicos o recursos informáticos

- Contenedores de información personal como por ejemplo historial clínico
- Bases de datos con información individual como ser por ejemplo cantidad de compras para políticas de comprador frecuente
- Tarjetas recargables de pre-pago para diversos servicios (muy usadas en Europa para la televisión por satélite)

## 1.4. Aplicaciones de la criptografía en estos dispositivos

A partir de los diferentes usos vistos para los dispositivos de bajos recursos es claro que los mismos necesitan tener implementados mecanismos de seguridad para manejar la información que procesan.

Uno de los mecanismos más utilizados, para lograr confidencialidad, es el de cifrado de la información. Esto es imprescindible dado que estos dispositivos tienen que ser capaces de asegurarle a su usuario que los datos que están almacenando serán accedidos solamente por personas autorizadas. Un caso es el de la telefonía celular y los equipos de comunicaciones pues, debido a que estos dispositivos usan ondas para transmitir la información, son muy vulnerables a la intercepción de las mismas, por lo que es necesario evitar que cualquier persona que escuche la señal pueda tener acceso a la información transmitida.

Otro de los mecanismos utilizados para corroborar la identidad de la fuente de la información así como también garantizar la integridad de los datos manejados por estos dispositivos es la firma digital. Un ejemplo donde el uso de ese elemento es fundamental es el caso del dinero electrónico, el cual deberá estar firmado digitalmente por la institución emisora para avalar su autenticidad.

Otras aplicaciones de la criptografía también relacionadas con el dinero electrónico son las que aseguran el anonimato y el no-repudio, las cuales son esenciales tanto para los usuarios del dinero como para las organizaciones que lo emiten. El anonimato garantiza al usuario que no será posible rastrear el uso que se hizo del dinero, mientras que el no-repudio asegura que ni el emisor ni los usuarios podrán negar que participaron de una transacción una vez que la misma ha finalizado.

Por último, unas de las aplicaciones más relevantes para este proyecto son las de validación y control de acceso las cuales tienen como objetivo restringir el acceso a recursos, habilitando solamente a aquellas personas que conocen cierta información y/o poseen cierto elemento personal de identificación.

## 1.5. Soporte criptográfico

En general el soporte criptográfico en estos dispositivos se centra en la implementación de algoritmos de encripción y firma digital, dependiendo de las aplicaciones que se les den a los mismos. Estas dos funcionalidades son suficientes para asegurar la confidencialidad, integridad y autenticidad de la información manejada por los dispositivos, según lo visto en la sección anterior. En general los criptosistemas implementados en los equipos de bajos recursos son principalmente RSA y en segundo lugar curvas elípticas (ECC).

El soporte criptográfico está dado principalmente sobre la base de ciertos estándares ya definidos que intentan permitir interoperabilidad entre diferentes aplicaciones. Así se puede asegurar que dos equipos distintos que realizan operaciones criptográficas podrán compartir información encriptada, siempre y cuando sus implementaciones cumplan con lo especificado en los estándares correspondientes.

En lo que respecta a RSA existe una familia de estándares llamada *Estándares de Criptografía de Clave Pública* (Public-Key Cryptography Standards, PKCS) [6.5]. Estos estándares especifican un API, llamado *Cryptoki*, para dispositivos que guardan información encriptada así como también que realizan funciones criptográficas. *Cryptoki* sigue la línea de la programación orientada a objetos, logrando independizarse de la tecnología (de cualquier dispositivo) y permitiendo compartir los recursos

(aplicaciones múltiples accediendo a múltiples equipos) presentándole a las aplicaciones una vista común del dispositivo llamada *crypotgraphic token*.

*Cryptoki* aísla las aplicaciones de los detalles criptográficos del dispositivo por lo que las aplicaciones no deben cambiar sus interfaces de acuerdo al equipo, o al ambiente, con el que se este trabajando, convirtiéndolas en aplicaciones portables.

Por el lado de los criptosistemas de curvas elípticas, el desarrollo de estándares se ha comenzado a realizar más recientemente que con el caso de RSA. Liderando esta iniciativa existe un grupo de trabajo llamado SECG (*Standards for Efficient Cryptography Group*) [7.7] que ha publicado algunos estándares para la programación de algoritmos de ECC así como listas de parámetros recomendados para lograr buenos niveles de seguridad. Su formato y objetivos son idénticos a los buscados por los PKCS mencionados anteriormente.

# 2. Introducción a las tarjetas inteligentes

Una tarjeta inteligente es una tarjeta que contiene un microprocesador y memoria o simplemente memoria sin lógica programable. Las tarjetas con microprocesador pueden agregar, borrar y manipular la información contenida en las mismas, mientras que una tarjeta sin microprocesador (por ejemplo tarjetas pre-pagas de teléfonos celulares) solamente puede ejecutar una función predefinida. Las tarjetas inteligentes aparecieron por primera vez en 1974 creadas por el inventor independiente Ronald Moreno.

Las tarjetas inteligentes, diferenciándose de las simples tarjetas magnéticas, pueden almacenar todas las funciones e información necesaria para su funcionamiento en la propia tarjeta. Por lo tanto, no requieren acceso a bases de datos remotas en el momento de realizar una transacción. Además las tarjetas magnéticas pueden ser fácilmente leídas, escritas o borradas por equipamiento comercial estándar sin necesidad de autenticación.

El hecho de poseer un procesador integrado que controla el acceso de las aplicaciones externas a la tarjeta les otorga un nivel mayor de seguridad que otro tipo de dispositivos. Es claro que si computadoras externas pudieran acceder a la memoria de las tarjetas directamente las mismas no serían en esencia mucho más seguras que un disquete.

## 2.1. Diferentes tipos de tarjetas

Hoy en día se pueden encontrar en el mercado 3 categorías de tarjetas inteligentes de acuerdo a sus capacidades. A continuación haremos una breve mención de cada una de ellas según lo descrito en [9.12].

- **Tarjetas con Microprocesador de Circuito Integrado**. Las tarjetas con microprocesador (generalmente denominadas como *chip card*) ofrecen, como ya se comentó, mayor capacidad de almacenamiento y mejor sistema de seguridad que las tarjetas magnéticas. También pueden procesar datos. Hace no más de 5 años las tarjetas brindaban, como máximo, una capacidad de almacenamiento entre 6KB y 8KB, hoy en día se puede contar con 128KB aunque debido a su excesivo precio son muy escasas en el mercado, es por esa razón que lo más común es encontrar tarjetas de 32KB. Estas tarjetas son usadas en una gran variedad de aplicaciones, especialmente aquellas tarjetas que contienen módulos de criptografía. Es por ello que las tarjetas con microprocesador han sido la plataforma principal para las tarjetas que almacenan una identidad digital.
  Algunos ejemplos de éste tipo de tarjetas:
  - Tarjetas que contienen dinero (billeteras digitales)
  - Tarjetas que brindan acceso seguro a redes.
  - Tarjetas que identifican teléfonos celulares frente a fraudes.

- **Tarjetas con Memoria de Circuito Integrado**. Este tipo de tarjetas puede almacenar de 1 a 4 KB de datos, pero no tiene un procesador con el cual manejarlos. Por lo tanto, dependen del lector de tarjetas para el procesamiento y por ello son adecuadas para usos donde la tarjeta ejecuta una operación fija. Las tarjetas con memoria son una alternativa popular de mayor seguridad para tarjetas magnéticas.

- **Tarjetas con Memoria Óptica**. Las tarjetas con memoria óptica lucen como una tarjeta común con un fragmento de CD pegado en una de sus caras – lo cual es básicamente lo que son. Estas tarjetas pueden almacenar más de 4MB de datos. La desventaja es que una vez que dichos datos se almacenaron ya no pueden ser alterados o directamente borrados. Es por ello que son ideales para la manutención de registros, por ejemplo archivo médicos. Hoy en día éstas tarjetas no tienen un procesador incluido aunque en poco

tiempo esto puede cambiar. A pesar de que su precio es comparable a las tarjetas con procesador, los lectores de tarjetas ópticas usan protocolos no estándar lo que los convierte en caros.

# 2.2. Seguridad

## 2.2.1. Capacidades criptográficas

Las tarjetas inteligentes actuales tienen suficientes capacidades criptográficas como para soportar algunas de las aplicaciones de seguridad más difundidas [1.8].

Las mismas soportan normalmente firma y encriptación con RSA con claves de largo 512, 768 o 1024 bits. Generalmente los algoritmos implementados hacen uso del Teorema Chino del Resto para acelerar el procesamiento y de esta forma mejorar su performance (ver apéndice VI).

Aunque las tarjetas tienen la habilidad de generar pares de claves para RSA, este proceso puede resultar muy lento e incluso la calidad de los pares de claves generados puede no ser mucha (es decir que las claves generadas pueden no ser muy seguras ante ataques conocidos).

Los tiempos que se necesitan para generar un par de claves de RSA de 1024 bits se encuentran dentro del rango de los 8 segundos a los 3 minutos, violando esta última cifra las especificaciones ISO para los tiempos de espera en las comunicaciones con las tarjetas, por lo que la mayoría de las veces se requiere hardware especializado para realizar esta operación (este hardware está en general incluido en la propia tarjeta – ver 4.1.1.1)

Por otro lado, el algoritmo de firma digital (DSA) no esta tan implementado en tarjetas inteligentes como el RSA y además cuenta con la desventaja que cuando se encuentra una implementación, ésta cuenta solamente con claves de largo de 512 bits.

## 2.2.2. Protección interna

La unidad de procesamiento de una tarjeta inteligente es la encargada de ejecutar el sistema operativo contenido en el chip, el cual a su vez, en algunos casos, implementa un sistema de archivos jerárquico en la memoria no volátil de la tarjeta. Además de este sistema de archivos, la unidad de procesamiento también implementa un conjunto de operaciones de acceso tanto para la tarjeta en sí misma como para el propio sistema de archivos [1.8].

El uso de estas operaciones es lo que permite, por ejemplo, que el archivo que contiene la clave privada este diseñado de tal forma que la información sensible de la clave nunca abandone el chip. Además, el uso de la clave privada esta protegida por el número de identificación personal (PIN), por lo que el poseer la tarjeta no implica la posibilidad de poder firmar con la misma.

Para hacer más difícil aún la extracción de información acerca del chip o del sistema de archivos, existen varios métodos de seguridad de monitoreo del hardware. Un método muy común es el de contar con un fusible (irreversible) que deshabilita cualquier código de prueba programado en la memoria de la tarjeta durante su fabricación. Otro método con el cual cuentan las tarjetas inteligentes es el de poseer circuitos resistentes a intromisiones; estos circuitos actúan anulando la respuesta de las funciones de salida de información. También existe una capa dieléctrica de resistencia cubriendo el chip, la cual protege al mismo de impurezas, del polvo y además previene el pasaje de radiación asociado con intentos de extracción de información de forma no autorizada.

Estos circuitos son capaces de reaccionar con la luz (indicando que la capa de resistencia ha sido perforada), la temperatura, el voltaje y fluctuaciones de la frecuencia fuera del rango específico de operaciones del chip. Existen mecanismos de protección para la memoria física los cuales incluyen el

*desordenar la memoria* (memory scrambling). Esto hace más difícil aplicar re-ingeniería e imposibilita un ataque que tenga como finalidad borrar una porción de memoria en particular. Más información sobre estos temas se puede encontrar en el apéndice VI.

Por último pero no menos importante, con la finalidad de evitar la clonación de tarjetas, un número serial inalterable es quemado en la memoria asegurando su unicidad [10.6].

Ya en el plano lógico más que físico, existen distintos tipos de seguridad interna usados en tarjetas inteligentes. Aquellos necesarios para tarjetas de "solo-memoria" son menos sofisticados que aquellos para tarjetas con microprocesadores.

Según lo especificado por la compañía *Gemplus* [1.9], el acceso a la información contenida en la tarjeta es controlado de dos formas:

- Quien puede acceder a dicha información (cualquier persona, el poseedor de la tarjeta o una entidad en particular).
- Como puede ser accedida la información (solo lectura, agregada, modificada o borrada).

### 2.2.3. Quien puede acceder a la información

La información que almacena una tarjeta inteligente puede ser dividida en tres grupos de acuerdo a quienes están autorizados a tener acceso a la misma [1.9], *Cualquier persona*, *El poseedor* o una *Entidad en particular.* A continuación detallamos cada uno de ellos,

*Cualquier Persona* - Algunas tarjetas inteligentes no requieren contraseña. Cualquiera que posea la tarjeta puede tener acceso a su información (por ejemplo: saber el tipo de sangre del poseedor en una tarjeta médica).

*Solo el Poseedor* - La forma más común de contraseña para el dueño de la tarjeta es un PIN (*Personal Identification Number*), un número de 4 o 5 dígitos. Por lo tanto, si una persona no autorizada trata de usar la tarjeta, ésta será bloqueada luego de, por ejemplo, 3 intentos fallidos a la hora de ingresar el PIN.

*Entidad en Particular* - Algunas tarjetas inteligentes solamente pueden ser accedidas por la empresa que la liberó al mercado (por ejemplo, una billetera digital tan solo puede ser recargada por el banco propietario de la misma).

### 2.2.4. Como puede ser accedida la información

Los datos almacenados dentro de una tarjeta inteligente no necesariamente todos tienen que tener el mismo propósito, es por esa razón que se puede hacer la siguiente diferenciación (según [1.9]):

- Información de solo lectura.
  Es el caso de tarjetas de identificación.
- Información agregada.
  Información agregada a la tarjeta por el propio usuario (ej.: historial médico).
- Información de solo actualización.
  Por ejemplo dinero digital.
- Información inaccesible.
  Toda la información propia de la tarjeta que solo incumbe a su funcionamiento.

# 3. Criptosistemas estudiados

A lo largo del proyecto se realizó un relevamiento de diferentes criptosistemas con el fin de conocer el estado del arte en el área y también tratar de encontrar aquellos que fueran adecuados o particularmente diseñados para su implementación en sistemas de bajos recursos.

A continuación se dará una reseña de los mismos explicando sus beneficios particulares al ser implementados en tarjetas inteligentes. Para cada uno de ellos existe también un apéndice que explica en detalle su funcionamiento.

## 3.1. ECC

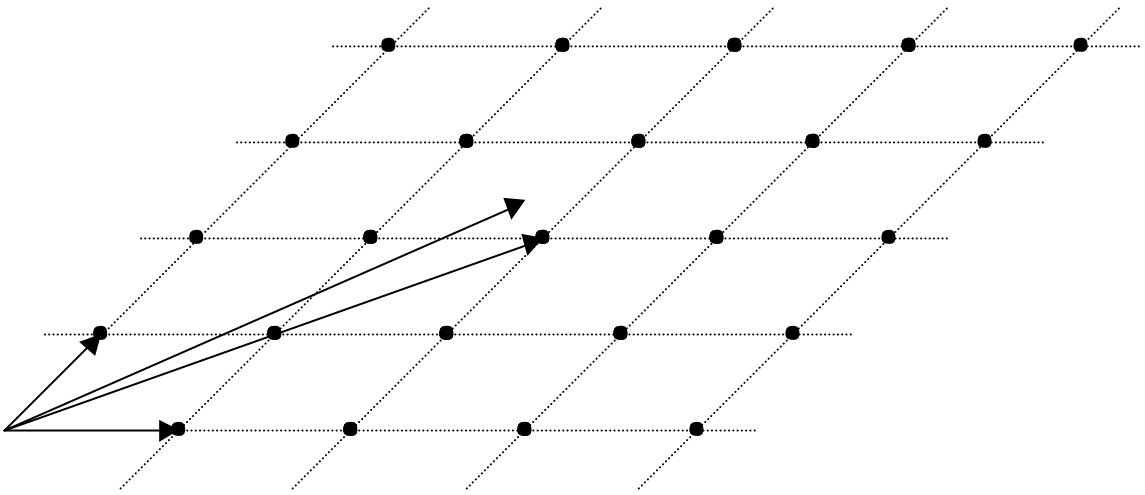
Muchos sistemas criptográficos de clave pública usan algún problema difícil (como por ejemplo el logaritmo discreto) en algún grupo. En general se usa como grupo uno de los cuerpos finitos $GF(p)$. Pero, dado que es posible también definir una estructura de grupo en las curvas elípticas, Koblitz [7.3] y Miller [7.4] propusieron independientemente alrededor de 1985 la idea de usar estas curvas como grupo donde calcular logaritmos discretos es difícil.

El hecho es que los algoritmos más eficientes para resolver este problema no funcionan en el grupo de las curvas elípticas teniendo que aplicarse los algoritmos más generales que no son tan efectivos. A causa de esto se pueden utilizar claves más chicas para lograr la misma seguridad que la que se obtiene usando cuerpos finitos. Esto redunda en que es más fácil implementar la aritmética en ambientes de bajos recursos, así como en un ahorro de memoria y de ancho de banda para el envío de claves. Más información sobre este criptosistema se encuentra en el apéndice I.

## 3.2. GGH

Este criptosistema fue propuesto por Goldreich, Goldwasser y Halevi en 1996 [3.1] y está basado en la complejidad de los problemas de reducción de lattices (ver Glosario), en particular del problema de encontrar el vector más cercano a uno dado (CVP – *Closest Vector Problem*).

A continuación en se muestran dos vectores en la esquina inferior izquierda y los puntos del lattice que generan. También aparece otro vector cualquiera (que no pertenece al lattice) junto con el vector del lattice y más cercano a él indicando cual es el tipo de problema en que consiste el CVP



GGH está considerado en general como más rápido que RSA y ElGamal, pero tiene la contra de que sus claves son más largas (según datos de [3.1]):

| | RSA / ElGamal | GGH |
|---|---|---|
| Tamaño de clave | $O(k)$ | $O(k^2)$ |
| Cálculos | $O(k^3)$ | $O(k^2)$ |

**Tabla 1: Comparación de GGH contra RSA y ElGamal**

La idea básica es que, dada una base de un lattice, es fácil encontrar un vector cercano a uno cualquiera dado adicionándole al mismo un vector de error pequeño, pero es difícil realizar la operación inversa en una base cualquiera (es decir, encontrar el vector original a partir de otro cercano) La trampa es que hay ciertas bases particulares en que resolver este problema es sencillo y una de estas bases será la clave privada. Aquí se puede ver por qué se dijo que las claves eran $k^2$ dado que las mismas están formadas por $k$ vectores de largo también $k$.

Entonces, el método para encriptar consiste simplemente en mapear el mensaje a un vector del lattice (combinación lineal entera de los vectores que conforman su base) y sumarle un vector de error (ver apéndice II)

En particular este algoritmo sirvió como un ejemplo de un criptosistema basado en lattices que fue quebrado, es decir que se probó que tiene fallas que hacen su uso no sea práctico. A continuación se presentará otro algoritmo basado también en lattices el cual, hasta este momento, es considerado seguro.

# 3.3. NTRU

El problema matemático que se encuentra por detrás de éste criptosistema es el de encontrar vectores extremadamente cortos en ciertos lattices (de dimensión grande). Este es un problema difícil de resolver y es de ahí que se deriva la seguridad del sistema criptográfico.

Su procedimiento de encriptación usa un sistema de mezcla basado en álgebra polinomial y reducción módulo dos números $p$ y $q$, mientras que la desencriptación usa un sistema de desmezclado cuya validez se basa puramente en teoría probabilística. También se basa en el hecho de que para la mayoría de los lattices, es difícil encontrar vectores extremadamente cortos pertenecientes a los mismos. Por más detalles ver el apéndice III.

NTRU entra en el marco general de un criptosistema probabilístico. Esto significa que la encriptación incluye un elemento aleatorio, por lo que cada mensaje tiene varias posibles encriptaciones. Encriptar y desencriptar con NTRU no requiere en promedio de muchas operaciones. Como ejemplo notamos que NTRU necesita en el orden de $N^2$ operaciones para encriptar o desencriptar un bloque de largo $N$, haciéndolo más rápido que las $N^3$ operaciones que requiere el criptosistema RSA. Además las claves del NTRU son de largo de orden $N$ en comparación con otros sistemas de clave pública que requieren claves de largo de orden $N^2$. Esto se da en casos como GGH donde se debe guardar una matriz completa como clave, mientras que en este caso alcanza con tener un polinomio de grado $N$.

Es por ello que el NTRU se presenta como un criptosistema muy interesante para su uso en dispositivos de bajos recursos como pueden ser las tarjetas inteligentes. El contar con claves de largo moderado posibilita que con pocas operaciones (y/u operaciones poco costosas y/o fáciles de implementar en hardware) se logre encriptar y desencriptar mensajes. También es importante para el procedimiento de creación de claves que las mismas sean chicas para de esa forma obtener una buena performance.

Además, como se verá más adelante, el hecho de usar polinomios con coeficientes pequeños (de 8 bits o menos) facilita la realización de las operaciones requeridas en los procesadores de las tarjetas. Por otro lado, sistemas como RSA requieren realizar operaciones aritméticas sobre grandes números (del orden de 1024 bits o más) lo cual es sensiblemente más complejo para los procesadores que utilizan las tarjetas.

## 3.4. RPK

Este criptosistema está basado en la complejidad del problema del logaritmo discreto, esto significa que en este sistema la clave pública es calculada a partir de la clave privada usando operaciones matemáticamente equivalentes a la exponenciación en cuerpos finitos.

El concepto central de RPK es la noción de *generador de mezclas* (mixture generator). Para comenzar pensemos en un generador de mezclas como una caja negra que consiste en una máquina de estados finita. Una máquina de éste tipo comienza en un estado inicial y entonces es capaz de ser guiada a través de una secuencia fija y extremadamente larga de estados diferentes, después de lo cual, si existe suficiente tiempo para llevarlo adelante (su período), dicha secuencia se repetirá. Después de cada paso, también produce una salida que consiste en un solo bit, es decir su valor puede ser o 1 o 0.

Para el generador de mezcla de nuestro interés, la secuencia de estados es tan larga que jamás podría ser físicamente recorrida a lo largo de su período. Los sucesivos estados pueden ser referenciados de acuerdo a su distancia con el estado inicial, medida en “pasos” o “pulsos de reloj”.

El generador de mezclas tiene algunas propiedades que lo hacen interesante. La primera es que puede moverse en la secuencia de estados un número fijo de pasos y ésta operación no le lleva mucho tiempo más que avanzar tan solo un estado. La segunda propiedad es que si es puesto a la fuerza en un estado en particular que se encuentra a un número de pasos *D* desconocido del origen, es capaz de volver a saltar a un nuevo estado que se encuentra a *D* pasos del estado actual sin saber cuanto vale *D* y esta acción no le lleva mucho más tiempo que avanzar un número pequeño de pasos, sin importar que tan grande sea *D*.

Finalmente, el generador también puede saltar un número repetido de veces *R* el salto que lo llevo desde el estado inicial al estado actual tomándole tan solo un pequeño múltiplo del tiempo requerido para saltar una vez.

La implementación real de los generadores de mezclas y la forma en que se utilizan en este criptosistema están explicadas con mayor profundidad en el apéndice IV.

Los generadores de mezclas mencionados aquí son implementados en general mediante el uso de *shift registers* por lo que su implementación en hardware en dispositivos de bajos recursos puede llegar a ser muy eficiente.

## 3.5. RSA

Este criptosistema fue propuesto en 1977 por Ronald L. Rivest, Adi Shamir y Leonard Adleman [6.2] cuyas iniciales de sus apellidos forman el nombre RSA. El problema matemático difícil que da la seguridad al sistema el de factorizar números enteros largos.

Es probablemente el criptosistema de clave pública más utilizado en la actualidad, por lo que también es muy utilizado en dispositivos de bajos recursos.

Este sistema es relativamente fácil de implementar en general, aunque en lo que respecta a su implementación en ambientes de bajos recursos tiene el problema de que necesita el manejo algebraico de números de muchas cifras (más de 200) lo que pude dificultar su implementación práctica. Por eso, como se verá más adelante, la tarjeta que utilizamos lo implementa utilizando un procesador dedicado.

Los detalles específicos de su funcionamiento se encuentran en el apéndice V.

# 4. Implementación

## 4.1. Detalles del dispositivo real utilizado

En este proyecto se usó la tarjeta *GemXpresso 211 PK* fabricada por la empresa *Gemplus* [1.9] ya que era el material de estudio con el cual contaba la Facultad que se adecuaba a nuestras necesidades; cabe aclarar que la Facultad también posee la tarjeta *CryptoFlex* de la compañía *Schlumberger* que no fue tenida en cuenta debido a que dicha tarjeta no tiene capacidades criptográficas.

### 4.1.1. Hardware y sus limitaciones

#### 4.1.1.1. GemXpresso 211 PK Card

A continuación se hará una descripción detallada del hardware específico de la tarjeta usada en este proyecto. Dicha descripción se dividirá en dos partes, primero se darán detalles generales del hardware de la tarjeta y luego se centrará en las características de su procesador.

**Ficha técnica de la tarjeta [1.9]**:

- Tamaño de la EEPROM { mínimo = 15+2 Kb; máximo = 21+2 Kb }
- No contiene applets pre-cargadas
- Cumple con el VOP 2.0.1. (ver 4.1.2.3)
- Fuente de energía = 3V/5V
- Tamaño del buffer de APDU (bytes) = 256+5
- Cuenta con un procesador Philips modelo P8WE5032
- Implementa los siguientes algoritmos de encriptación:
  - DES/3DES-ECB
  - DES/3DES-CBC
  - RSA con claves de hasta 1024 bits de largo.
  - RSA CRT

**Ficha técnica del procesador [1.11]:**

- Procesador de 8 bits
- Frecuencia de operación entre 1 y 8 MHz
- Computa una instrucción cada 6 tics del clock
- Incluye un co-procesador de DES y 3-DES
- Cuenta con un co-procesador FameX (*Fast Accelerator for Modular Exponentiation – eXtended*) de 32 bits el cual acelera las operaciones en enteros con módulo para criptosistemas de clave pública. El co-procesador necesita control apropiado por el sistema operativo y no provee una función de seguridad por sí mismo.
- Posee un generador de números realmente aleatorios por hardware (TRNG, *True Random Number Generator*) que cumple la norma FIPS 140-2 [1.10]
- Soporta 32 K de EEPROM y hasta 256 K de RAM
- Tiene sensores de variación de voltaje, temperatura y frecuencia para evitar ataques (ver apéndice VI)
- Posee 5 capas de metal y un grosor total de 0.35 *mm* lo que dificulta su análisis físico.

- Incluye una protección de descargas electro estáticas de hasta 4 kV

### 4.1.2. Software y Programación

La codificación de los programas que hacen uso de las tarjetas inteligentes se hizo en dos niveles: por un lado se encuentra el código que corre en la propia tarjeta y por el otro los programas que corren en la PC que tiene el lector de tarjetas conectado.

El código en la tarjeta utiliza el API de las *Java Cards* [9.4] y está programado usando la versión limitada de Java que aceptan las mismas, mientras que el código externo es Java tradicional que utiliza el *Open Card Framework* (OCF) [11.1] y el *Visa Open Platform* (VOP) [11.3]. OCF permite abstraerse de los detalles del hardware del lector que se utiliza para la comunicación con la tarjeta mientras que las funcionalidades de las clases del *Visa Open Platform* [11.4] posibilitan el manejo de las diferentes aplicaciones instaladas en la tarjeta.

#### 4.1.2.1. Programación en la *Java Card*

Las tarjetas inteligentes contienen un microprocesador como cualquier computadora, pero al no tener ninguna manera directa de realizar operaciones de entrada y salida, las mismas se realizan completamente a través del lector de tarjetas también llamado CAD (*Card Acceptance Device*) El mismo deberá estar conectado a una PC a través de un cable serial o una conexión de red. A causa de esto, la comunicación con la tarjeta se realiza a través de paquetes llamados APDUs (*Application Protocol Data Unit*) que en sí no son otra cosa que arreglos de bytes conteniendo comandos, datos y parámetros correspondientes los mismos y las respuestas de la tarjeta (ver Figura 1).

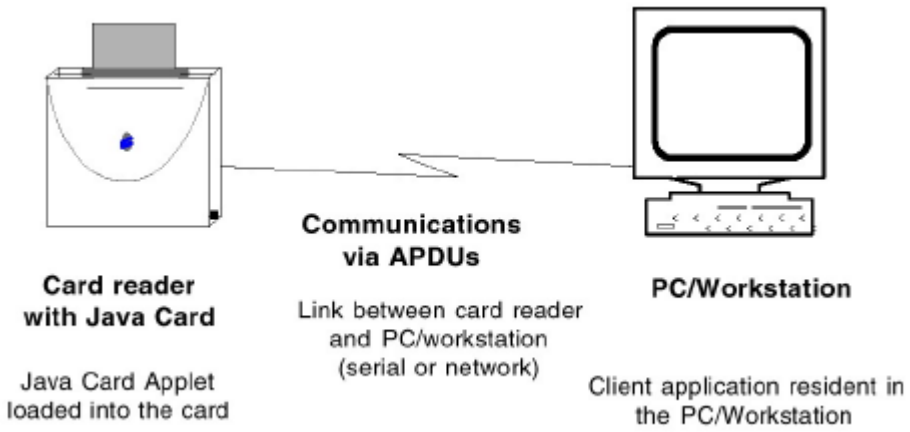



**Figura 1: Entorno de uso de las *Java Cards* (de [9.7])**

De esta forma para programar en la tarjeta lo primero que se debe hacer es codificar toda la información que se quiere que reciba la misma desde el exterior en diferentes APDUs de entrada y salida los cuales serán procesados por el applet que se colocará en la tarjeta. Para una explicación muy completa de la estructura de los APDUs y su procesamiento en el software de la tarjeta se recomienda ver [9.1]

La programación en la tarjeta se hizo siguiendo el estándar *Java Card* 2.1.1 de *Sun*. El mismo da una especificación para el llamado JCRE (*Java Card Runtime Environment*) [9.2] Una implementación del mismo (como la que existe en las tarjetas usadas en este proyecto) consiste en una implementación del JCVM (*Java Card Virtual Machine*) [9.3] y del API específico de las *Java Cards* [9.4]. La JCVM también está definida por *Sun* y la misma soporta un subconjunto de las operaciones que poseen las máquinas virtuales de Java estándar. Las características no soportadas son las siguientes:

- Carga dinámica de clases
- La clase Security Manager

- Recolección de basura (*garbage collection*)
- Threads
- Clonación de objetos
- Hay algunas restricciones en el control de acceso a las clases dentro de los paquetes
- Palabras reservadas: `native`, `synchronized`, `transient` y `volatile`
- Tipos `char`, `double`, `float`, `long` (y opcionalmente `int`)
- Arreglos multidimensionales
- La mayoría de las clases utilitarias del API de Java

En lo respectivo a las clases de criptografía el API posee dos paquetes específicos sobre el tema. Ellos son `javacard.security` y `javacardx.crypto`. Según la documentación de *Sun* no es necesario implementar ninguna de las clases de estos paquetes para satisfacer su especificación del API, aunque al menos las clases que crean instancias de los algoritmos criptográficos deben existir y lanzar la excepción `CryptoException` con la razón `NO_SUCH_ALGORITHM`. Estas clases son: `MessageDigest`, `Signature`, `RandomData`, `Cipher` y `KeyBuilder`. Este tema se comentará con más profundidad en 4.3.

#### 4.1.2.2. Software de desarrollo

Para el desarrollo de las aplicaciones prácticas se contó con el ambiente de desarrollo que provee *Gemplus* conjuntamente con sus tarjetas, el denominado *GemXpresso RAD 211*. Este ambiente contiene los componentes necesarios para el desarrollo y depuración de applets. Los componentes de mayor utilidad y que se destacan más son el *GemXpresso Simulator* y el *JcardManager*.

El primero es un simulador que, como lo indica su nombre, emula el comportamiento de la tarjeta permitiendo de esta forma probar las distintas aplicaciones creadas por el usuario en un ambiente controlado sin la necesidad de trabajar directamente sobre la tarjeta, lo cual es sensiblemente mas lento.

Por su parte el *JcardManager* es una aplicación de tipo cliente genérica mediante la cual se logra la comunicación con las applets así como también testear las mismas sin necesidad de desarrollar nuestro propio cliente. Es capaz de comunicarse tanto con una tarjeta real como con el simulador.

Sus principales funciones son las siguientes:

- Tener acceso a un *convertidor* para transformar los archivos .class a un formato pronto para ser cargado en la tarjeta o en el simulador, dichos archivos tienen extensión .cap (*Card Applet Package*).
- Llamar a un *verificador* que controla que todas las sentencias Java utilizadas al programar la aplicación sean soportadas por la tarjeta.
- Manejar la capa de comunicación del OCF.
- Tener acceso al *Deployment Editor* para permitir cargar, seleccionar e instalar applets en un solo paso.
- Realizar la autenticación tarjeta/cliente.
- Cargar e instalar applets en la tarjeta.
- Manejar algunas de las funcionalidades criptográficas de la tarjeta.
- Desplegar y grabar un archivo log de toda la actividad entre el programa y el objetivo.
- Desplegar los contenidos de una tarjeta.

Tanto el *verificador* como el *convertidor* mencionados anteriormente son parte del kit de desarrollo de *Java Cards* proporcionado por *Sun*. En este caso el *JcardManager* funciona solamente como una interfase gráfica para estas dos aplicaciones.

También existe un utilitario para calcular el tamaño del archivo .cap. Este utilitario, por no tratarse de una versión comercial, simplemente devuelve el tamaño del código en sí mismo, no incluye las clases que se instancian dentro de la aplicación, por lo que no fue de gran utilidad.

#### 4.1.2.3. Programación del cliente externo

El código del cliente externo (que en este caso es la llamada aplicación de referencia) fue programado en Java utilizando las clases del OCF [11.2] y del VOP [11.4] que están incluidas en el kit de la tarjeta para comunicarse con la misma. En este caso utilizamos la versión 1.2 del OCF publicada en diciembre de 1999 y la versión 2.0.1 del VOP con fecha de abril de 2000.

El OCF es una arquitectura abierta propuesta por varios proveedores de equipamiento para tarjetas inteligentes entre los que se encuentran *IBM*, *Gemplus*, *Schulmberger*, *Sun* y *Visa* que provee un conjunto de APIs que permiten una interfase común entre lectores de tarjetas y aplicaciones externas. OCF garantiza que las applets y aplicaciones cliente que se desarrollen a partir de él van a funcionar con tarjetas y lectores de diversos proveedores siempre y cuando éstos soporten a su vez esta arquitectura. Es decir que los fabricantes deben proporcionar implementaciones de las clases del API que trabajen con sus productos.

Además de esto, otro beneficio del uso de OCF en la programación de los clientes externos, es que su API de alto nivel evita tener que conocer el funcionamiento completo del sistema de comunicación específico con el CAD que se esté usando. De esta forma el proceso de desarrollo de las aplicaciones es más rápido y menos propenso a errores.

La arquitectura básica del OCF se divide en dos capas: *CardTerminal* y *CardService* y además posee una tercer componente llamada *ApplicationManagement* para lograr mantenerse independiente de los emisores de la tarjeta (ver Figura 2).

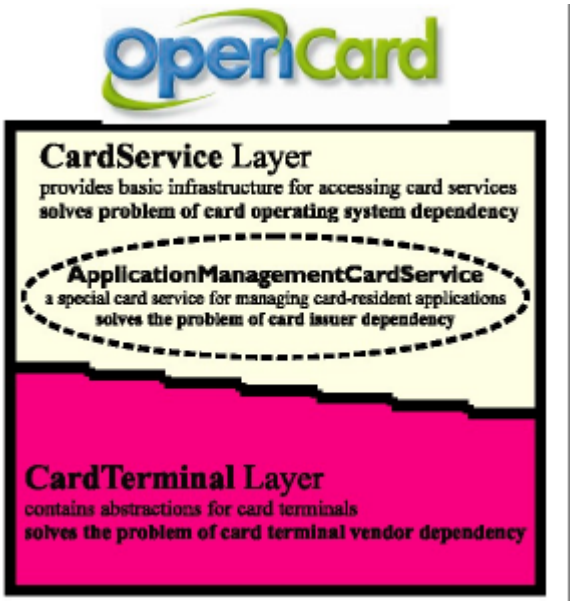



**Figura 2: Partes principales de la arquitectura del OCF (de [11.2])**

La capa *CardTerminal* provee el acceso físico a las terminales lectoras de tarjetas a partir de drivers apropiados que deben ser implementados por los fabricantes, y es usando la misma que nuestra aplicación externa accede a la tarjeta a través del CAD. El *ApplicationManagement* por su parte soluciona los problemas que surgen a causa de que en una misma tarjeta pueden haber diferentes aplicaciones y la forma de guardar y acceder a las mismas puede variar entre una tarjeta y otra. Por esta razón es este componente del OCF el que permite localizar, seleccionar, instalar y desinstalar aplicaciones en una tarjeta, entre otras funciones.

Por último la capa *CardService* trabaja en alto nivel sobre los diferentes sistemas operativos de las tarjetas manejando operaciones generales que luego serán implementadas en forma específica para cada sistema operativo particular. Un ejemplo de uno de estos servicios es el *SignatureCardService* que ofrece métodos para crear y verificar firmas digitales con diferentes algoritmos.

Esta última capa del OCF fue utilizada como la unión entre el OCF y el VOP. El VOP también es una especificación desarrollada por *Visa* y otros fabricantes (*Schlumberger*, *Hitachi*, *Gemplus*, etc.) que fue pensada originalmente para el manejo de tarjetas inteligentes con múltiples aplicaciones en el entorno bancario. Sus objetivos primarios son:

- Permitir el manejo de las aplicaciones que corren en la tarjeta (esto es: cargarlas, instalarlas y enviarles comandos)
- Establecer una canal seguro entre la tarjeta y la terminal

El mismo además incluye un documento que especifica los comandos disponibles en el framework y los principios de interoperabilidad ente las *Java Cards* y otros tipos de tarjetas.

Como se mencionó, ambos estándares interoperan mediante la definición del VOP como un *CardService* del OCF. De esta forma se tiene una clase de OCF que implementa también las funcionalidades del VOP permitiendo a través de ella realizar diferentes operaciones sobre la tarjeta programando en alto nivel.

En nuestro caso, las clases del OCF fueron usadas para:

- Iniciar la comunicación con el CAD e indicarle que tipo de tarjeta se espera (en nuestro caso podría ser una tarjeta real o el simulador)
- Abstraer los detalles de la conexión con el CAD
- Manejar los APDUs de entrada y de salida y las IDs de las aplicaciones como clases de Java

Por otro lado el VOP se utilizó para:

- Realizar la autenticación contra la tarjeta
- Seleccionar applets en la misma
- Enviarle APDUs

### 4.1.3. Soporte criptográfico y de seguridad integrado

Como se dijo anteriormente el API de las *Java Cards* posee dos paquetes incluidos que definen diferentes operaciones criptográficas. Si bien no es necesario implementar sus operaciones para satisfacer el estándar de *Sun* la tarjeta que fue utilizada durante el proyecto en realidad implementa algunas de estas funcionalidades. Ellas, según [9.8], son las siguientes:

- Tipos de clave
  - DES
  - RSA
  - RSA CRT (Chinese Reminder Theorem)

- Largos de clave
  - RSA 512 bits
  - RSA 768 bits
  - RSA 1024 bits
  - DES 64 bits
  - DES 128 bits
  - DES 192 bits

- Algoritmos de firma
  - DES MAC4 NOPAD
  - DES MAC8 NOPAD
  - RSA SHA PKCS1

- Algoritmos de hashing
  - SHA

    - MD5

- Generación de claves
    - RSA 512 bits
    - RSA 1024 bits
    - RSA CRT 512 bits
    - RSA CRT 1024 bits

- Algoritmos de cifrado
    - DES CBC NOPAD
    - DES ECB NOPAD
    - RSA NOPAD

En esta lista, las referencias a *NOPAD* y *PKCS1* se refieren a las diferentes estrategias de relleno (*padding*) que se pueden aplicar. Las mismas consisten básicamente en agregar algunos datos aleatorios a los datos a encriptar de forma que, aunque se encripte el mismo texto original dos veces, el texto cifrado sea diferente cada vez. Esto permite evitar ciertos tipos de ataques que, para el caso particular de RSA, están comentados en el apéndice V.

Por otro lado los códigos *MAC4* y *MAC8* se refieren a la utilización de los primeros 4 u 8 bytes de la encripción por DES del texto a firmar como la firma digital correspondiente al mismo. Asimismo los códigos CBC (*Cipher Block Chaining*) y ECB (*Electronic Code Book*) se refieren a dos modos diferentes de uso del algoritmo DES (ver [1.4]).

En realidad al tratarse este caso particular de una tarjeta que puede ser exportada sin necesidad de ninguna licencia particular, la misma tiene además ciertas restricciones adicionales como ser que las claves de DES sólo pueden tomar un número determinado de valores fijados previamente y que las claves de RSA están limitadas a 512 bits (de hecho la tarjeta puede crear claves de largo mayor, pero las mismas serán truncadas cuando se corran los algoritmos de cifrado o descifrado)

#### 4.1.3.1. Ventajas

En lo que respecta a los algoritmos de clave pública, que son el punto principal de enfoque de este proyecto la tarjeta ofrece bastantes opciones en lo referido a largos de clave, pero solamente limitadas a RSA. Esto es claramente así pues la tarjeta posee un coprocesador especializado para correr este algoritmo por lo que todas sus operaciones de clave pública están centradas en el mismo.

Como se verá más adelante, en la sección de comparaciones, estos algoritmos corren con mucha eficiencia por lo que pueden ser usados perfectamente en aplicaciones prácticas.

Por otro lado, la definición por parte de *Sun* de un API estándar para las implementaciones de algoritmos criptográficos en *Java Cards* (al igual que existe para Java con el llamado JCE) permite mucha flexibilidad dado que se pueden utilizar diferentes implementaciones de dichos algoritmos sin necesidad de modificar el código fuente de los programas.

#### 4.1.3.2. Desventajas

Las mismas están centradas básicamente en la falta de opciones en cuanto a algoritmos a usar, aunque este detalle viene heredado de la propia API de *Sun*, la cual (aunque fuera implementada en forma completa) no admite posibilidades de usar algoritmos de clave pública distintos de RSA.

Por otro lado hay desventajas específicas con respecto a la tarjeta usada, básicamente en lo que respecta a sus limitaciones impuestas por motivos legales.

En lo referido a la implementación en sí, la misma no posee fallas de seguridad que estén documentadas o hayamos podido detectar por nuestra cuenta. Si bien es cierto que utiliza algunas técnicas particulares para reducir el número de operaciones necesarias (como la de usar un exponente público fijo

de 65537) que la separan de la implementación tradicional del algoritmo, las mismas no están consideradas como inseguras en la literatura. Por más información sobre esto último se recomienda ver el apéndice V.

#### 4.1.3.3. Extensiones posibles

En este proyecto se intentó plantear una posible extensión de esta API con diferentes algoritmos de clave pública. Como se verá a continuación, lo que se propuso fue implementar otros paquetes paralelos a los definidos por *Sun*, con las mismas clases y funciones, pero incluyendo la posibilidad de utilizar diferentes algoritmos a los especificados originalmente.

Dado que la estructura general de los paquetes en cuestión (`javacard.security` y `javacardx.crypto`) esta definida en forma de framework, la misma es muy fácilmente extensible y, en lo que respecta a los usuarios finales de las clases, lo único que varía es la existencia de nuevas constantes en algunas clases que permiten generar claves, encriptar y desencriptar con los nuevos algoritmos. En realidad no se extendieron los paquetes de *Sun* directamente pues los mismos ya están cargados en la tarjeta y no es posible agregarle clases a los mismos una vez que ya están instalados sino que, como ya se dijo, se incluyeron nuevos paquetes que implementan al framework con las nuevas funcionalidades.

# 4.2. Aplicación de referencia

## 4.2.1. Introducción

La motivación inicial para el desarrollo de esta aplicación fue el hecho de ganar experiencia con el ambiente de desarrollo de las *Java Cards* en forma previa a comenzar el análisis e implementación de los algoritmos criptográficos. De esta forma se logró tener una idea más acabada de las características del ambiente lo que permitió, durante el estudio de la implementación los criptosistemas, poder saber claramente que era lo que se podía realizar en la tarjeta y qué puntos se debían tener en cuenta a la luz de los recursos limitados de la misma. Esta primer aplicación de referencia realizaba la autenticación utilizando un algoritmo de encriptación trivial que era en realidad la función identidad para cualquier clave que se le pasara.

Por otro lado, pensando en una fase posterior del proyecto, cuando se tocó el tema de la comparación de los diferentes criptosistemas que se implementaron, la aplicación de referencia pudo ser usada como patrón de referencia (de ahí su nombre) para comparar las performances relativas de los diferentes algoritmos en la práctica. Es decir que la aplicación proveyó un conjunto de operaciones criptográficas fijo bajo las cuales se comparó la eficiencia de los mencionados algoritmos. Por esta razón se trató de que la aplicación no presentara una tendencia particular hacia un tipo de operación buscándose específicamente que realizara el mismo número de encriptaciones que de desencriptaciones. Cabe aclarar que esta aplicación no incluye las operaciones de creación de claves en la tarjeta las cuales deberían ser analizadas independientemente dado que en general son operaciones suficientemente costosas como para justificar el ser estudiadas en forma separada.

Por último, esta aplicación permitió disponer, como producto final del proyecto completo, de una aplicación funcional, implementada en el ambiente de desarrollo de las *Java Cards*, la cual permite ver en funcionamiento las operaciones criptográficas más allá de los estudios presentados en este documento.

## 4.2.2. Características generales

Esta aplicación consiste, en líneas generales, en un sistema que permite la autenticación de usuarios ante un cierto agente exterior mediante el uso de un PIN y la tarjeta. En su diseño se tuvo en cuenta que la misma pudiera ser usada de forma de satisfacer la mayor parte de los criterios básicos que toda

aplicación de autenticación debería cumplir, según el *European Security Forum* [2.1]. Los mismos son los siguientes:

- Los usuarios se deben autenticar una vez sin importar cuantos sistemas necesitan acceder para cumplir su función.
- La autenticación debe ser relativamente sencilla para el usuario.
- Los usuarios sólo deben confirmar su identidad una vez por sesión.
- Los procesos de identificación deben ser efectivos (es decir, ser capaces de decidir si el que se registra es un usuario autorizado y en caso afirmativo saber si en realidad es quien dice ser)
- El proceso de identificación debe ser eficiente en términos de costo y tiempo empleado por usuario.

Desde el punto de vista del usuario el proceso de autenticación consiste únicamente en introducir su tarjeta en el lector e introducir su PIN en una consola. El sistema en sí está basado en una técnica de *desafío-respuesta* (challenge-response) en el que el sistema le enviará a la tarjeta uno o varios desafíos a los que la misma debe responder, y en caso de que todos sean correctamente contestados, se asumirá que el usuario que posee la tarjeta es realmente quien dice ser. Estos desafíos tendrán dos características fundamentales:

- Con alta probabilidad sólo podrán ser respondidos por la tarjeta correcta (cuantos más desafíos sean contestados correctamente menor será la probabilidad de error)
- La respuesta a uno de estos desafíos no le dará al sistema autenticador ninguna información sobre la forma en que la tarjeta los resuelve, sin importar la cantidad de desafíos planteados.

Dado que se utilizan algoritmos de clave pública este sistema de autenticación es considerablemente más seguro que el de comparar solamente el PIN escrito por el usuario con el que está en la tarjeta. La diferencia fundamental es que, en esta aplicación, toda la comunicación entre la tarjeta y la máquina a la que está conectada se realiza en forma encriptada evitando así que un atacante externo pueda obtener el PIN del cliente mediante la observación de la información que se transfiere.

Cabe aclarar que hay tarjetas que incluyen la funcionalidad de encriptar y/o firmar todos los APDUs que se intercambian entre la tarjeta y el lector, aunque esto generalmente se implementa mediante el uso de algoritmos de clave simétrica o RSA pero de ninguna otra forma.

Como beneficio agregado, este procedimiento de comunicación encriptada permitiría en principio implementar un sistema en que el usuario coloque su tarjeta en un lector físicamente seguro y que se autentique usando la misma desde diferentes máquinas de la red en que se encuentra el lector sin tener que introducir su tarjeta en cada máquina cada vez. Con esto se conseguirían las ventajas de la identificación de los usuarios por medio de la tarjeta sin necesidad de contar con un lector en cada máquina de la red. Además, este sistema permite lograr eficientemente algunos de los criterios básicos de los sistemas de autenticación mencionados anteriormente como ser el de autenticarse una sola vez aunque se necesite acceder a varios sistemas (*single sign-on*).

Este sistema de mezclar la autenticación por contraseña con el hecho de poseer un elemento de hardware (conocido en general como *token* en la bibliografía [2.1]) presenta algunas otras ventajas sobre los métodos tradicionales como ser:

- Facilita la administración al poder utilizar un solo token en lugar de conocer múltiples palabras clave; alcanza con introducir la tarjeta e indicar el PIN para autenticarse ante cualquier servicio.
- Aumenta la seguridad al necesitarse tanto el PIN como el token para poder personificarse como otro usuario.
- Es muy portable al ser independiente de una máquina específica. El usuario incluso podría autenticarse a través de un browser conectado a Internet.
- Al estar implementada sobre una tarjeta inteligente, esta es una solución muy versátil dado que las tarjetas son capaces de llevar múltiples aplicaciones las cuales pueden, en principio, ser agregadas o eliminadas durante el ciclo de vida de la misma. De esta forma la misma tarjeta que se usa para autenticarse podría ser usada también como tarjeta de

crédito, como billetera para almacenar dinero electrónico o cualquier otras aplicación imaginable.

- Dado que la clave privada queda almacenada en la tarjeta no existe el problema que algún troyano pueda espiar la clave, problema que si ocurre cuando las claves se almacenan en una PC.
- Relacionado con esto último está el hecho de que cualquier software que corre en una máquina dada puede eventualmente ser copiado o modificado por un usuario malicioso que tenga derechos de administración sobre la misma (en particular cualquier hacker va a tener estos privilegios en su propia máquina). Por esta razón la implementación en una tarjeta inteligente hace que los usuarios externos no tengan de por sí ningún derecho particular de acceso a la misma independientemente de los privilegios que tengan sobre la computadora a la que esta conectado el lector.
- El hecho de usar PKI (*Public Key Infrastructure*) asegura, como se dijo antes, que la clave privada nunca deja la tarjeta lo cual aumenta la seguridad del sistema.

Por otro lado las principales desventajas del sistema así implementado son las siguientes:

- Se necesita hardware especial de lectura, a la inversa de los sistemas basados solamente en contraseñas los cuales se pueden implementar exclusivamente en software.
- No existe un estándar aceptado universalmente con relación a estos lectores de tarjetas, por lo que el sistema pierde portabilidad (aunque claramente iniciativas como el OCF ayudan a mitigar este problema).
- El usuario debe transportar la tarjeta con él siempre que se desee autenticar.
- El sistema es más costoso por el hecho de tener que proveer una tarjeta a cada cliente y reponerla en caso de posibles pérdidas.
- La autenticación es en general más lenta por requerir encriptación y posiblemente, la respuesta a varios desafíos.

### 4.2.3. Funcionalidades

La aplicación de referencia consiste en un sistema de autenticación de usuarios y cuyo funcionamiento básico es el siguiente:

1. El usuario introduce su PIN[1] en la consola de la máquina ante la que desea autenticarse.
2. La máquina busca en su base de datos interna la clave pública del usuario asociado al PIN que se introdujo.
3. La máquina inicia entonces la comunicación con la tarjeta y le envía su clave pública a la misma.
4. A continuación esta máquina genera un string aleatorio, lo encripta con la clave pública de la tarjeta y se lo envía a la misma.
5. En este punto la tarjeta inteligente debe desencriptar este string usando su clave privada
6. La tarjeta vuelve a encriptar esta información usando la clave pública de la máquina cliente la cual fue recibida en el paso 3 y se lo envía de vuelta a la misma.
7. La máquina cliente debe desencriptar el string usando su clave privada y comparar el mismo con el string aleatorio que fue generado en el paso 4.
8. Si los dos valores coinciden la máquina cliente tiene 2 opciones: puede dar por válido el proceso de autenticación y proceder a iniciar el servicio requerido con el cliente o en su defecto puede volver al paso 4 y repetir el proceso tantas veces como quiera hasta asegurarse de que la tarjeta realmente posee la clave privada asociada a la clave pública registrada junto con el PIN introducido. Si, por el contrario, los dos valores no coinciden la autenticación se considera fallida.

Cabe aclarar que en estos casos la única utilidad del PIN es dar una representación más sencilla de la clave pública de la tarjeta de forma de que pueda ser recordada por el usuario, pero que en teoría ese

[1] Si bien a lo largo de este documento se habla del PIN asociado a un usuario, este no tiene por qué ser un número necesariamente, sino que podría ser cualquier cadena de caracteres.

podría indicar directamente su clave pública (i.e. la que se corresponde con la clave privada que contiene su tarjeta) a la máquina cliente y prescindir de esta forma de la tabla de conversión de PINs a claves públicas. Esto aumentaría la seguridad del sistema pues siempre se podría atacar a la base de datos en la máquina externa y obtener así las claves públicas asociadas a los PINs. Claro está que esta información sólo sería útil si el atacante pudiera crear una tarjeta falsa con una clave privada compatible con la clave pública que robó de la base de datos atacada.

El par de claves pública / privada de cada tarjeta debería ser también generado por la tarjeta misma, como forma de aumentar la seguridad del sistema. De esta forma se lograría que nadie (ni siquiera el propio usuario de la misma) conozca la clave privada que posee la tarjeta. Esta operación puede ser costosa, pero siempre que los recursos internos de la tarjeta sean suficientes para implementarla, se debería dejar que la misma cree sus claves, dado que esto elimina la necesidad de confiar en una tercera entidad para la generación de las mismas. Esta entidad debería asegurar que no almacenará en ninguna forma las claves creadas.

## 4.2.4. Diagrama de clases

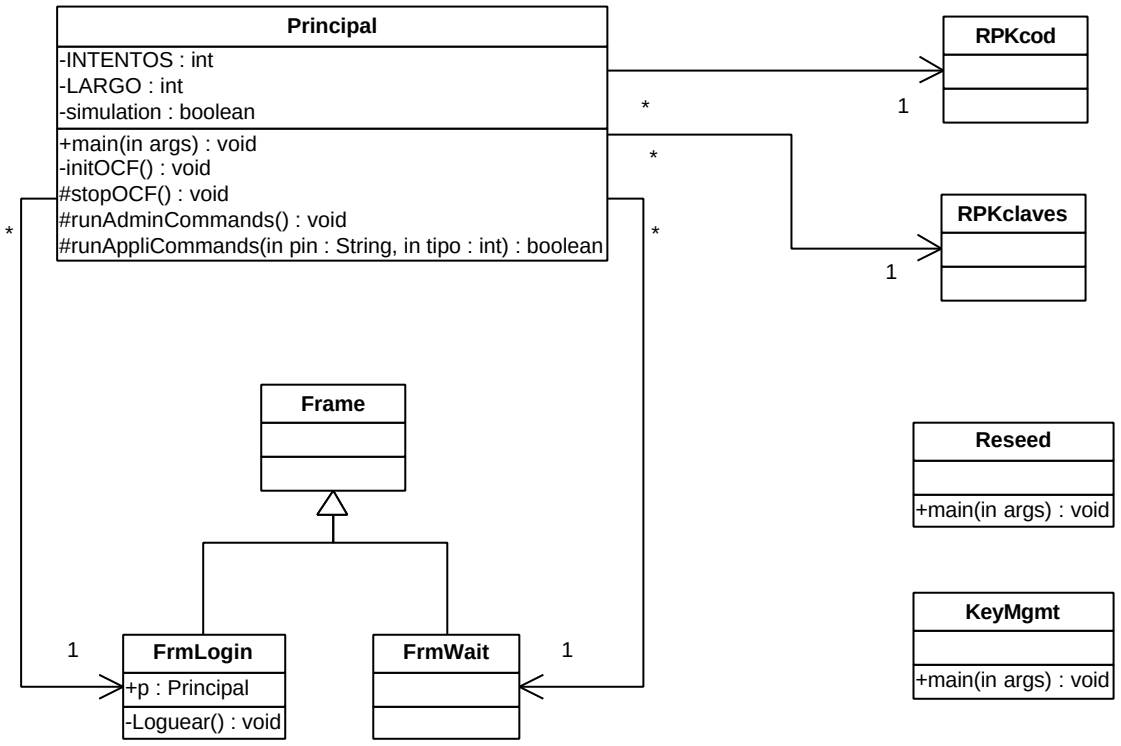



**Figura 3: Diagrama de clases de la aplicación de referencia**

## 4.2.5. Implementación

### 4.2.5.1. Clase Principal

Esta es la clase que contiene el método *main* de la aplicación y es la que maneja toda la comunicación con la tarjeta. Ella utiliza el *Open Card Framework* [11.1] y el *Visa Open Platform*[11.3] para inicializar la comunicación con la tarjeta, autenticarse contra la misma y enviarle los APDUs correspondientes.

Al ser ejecutada, esta clase recibe una serie de parámetros que regulan su comportamiento. El primero indica si la misma se correrá utilizando una tarjeta real (cuando se tiene el lector correspondiente conectado a la máquina) o si se ejecutará contra el emulador. Para usar el emulador el parámetro deberá ser –s, en otro caso podrá usarse cualquier otra letra. El segundo parámetro permite indicar directamente el PIN del usuario al ejecutar la aplicación, lo que evita el pasaje por la parte gráfica (ver 4.2.5.2). Este modo no tiene mucho uso en la práctica pero fue implementado a causa de que, al querer realizar mediciones de tiempo de ejecución de los diferentes algoritmos, interesaba que toda la corrida del programa se hiciera junta sin tener que esperar una entrada del usuario la cual agregaría una incertidumbre inaceptable a las mediciones. Si este parámetro no se indica se llamará en todos los casos a la interfase gráfica.

Además existe un tercer parámetro que también es usado en el caso de que no se quiera pasar por la interfase gráfica y que permite seleccionar el algoritmo criptográfico que se usará para realizar la encriptación y desencriptación (ver 4.3). Las opciones para este tercer parámetro son por lo tanto –ntru, –rsa y –rpk. En caso de no indicarse nada se tomará NTRU por defecto. Por último cabe aclarar que si el programa se llama sin ningún parámetro en absoluto se asumirá por defecto que la autenticación se realizará contra una tarjeta real y usando la interfase gráfica.

Esta clase lee el archivo de claves por completo al construirse y guarda todos los datos del mismo en una tabla de hash. De esta forma, una vez inicializada la misma, las búsquedas de las claves asociadas a los PIN que se le pasen se pueden realizar en forma bastante eficiente ($O(1)$ en caso promedio)

#### 4.2.5.2. Clase FrmLogin

Esta clase implementa la ventana gráfica donde el usuario introduce su PIN y selecciona el criptosistema bajo el cual se realizará la autenticación. Hereda de la clase `Frame` de Java.

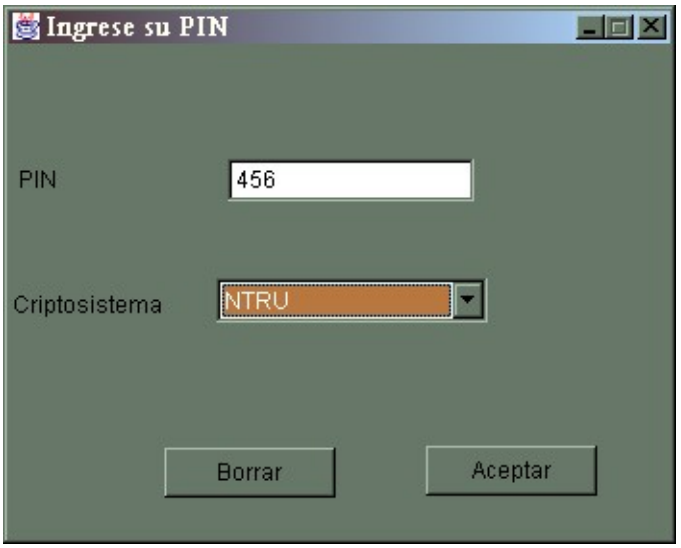



**Figura 4: Ventana gráfica de la aplicación de referencia**

#### 4.2.5.3. Clase FrmWait

Esta es la ventana que aparece luego de dar clic en el botón *Aceptar* de la ventana anterior indicado que se debe esperar un momento mientras se lleva a cabo la autenticación. Al igual que en el caso anterior hereda de la clase `Frame`.

#### 4.2.5.4. Clase Reseed

El generador de números aleatorios que utiliza la clase `Principal` para generar los valores para los desafíos es un objeto de la clase `SecureRandom` de java. Esta clase crea su propia semilla (*seed*) al ser instanciada y este es un proceso que lleva un tiempo sensible. A causa de esto la aplicación no puede crear un objeto nuevo en cada ejecución pues esto necesita demasiado tiempo. Para solucionar este problema la aplicación maneja un objeto de clase `SecureRandom` ya inicializado el cual se guarda en un archivo (*random.obj*) mediante el mecanismo de serialización de Java.

Con este método se evita el tiempo perdido en calcular le semilla pero, si las cosas se dejaran así, el generador de números aleatorios produciría siempre la misma secuencia fija de valores lo cual anula el propósito mismo del generador. Para solucionar este problema se creo la clase `Reseed` la cual, al ejecutarse, crea un nuevo objeto de clase `SecureRandom` y lo guarda en el archivo *random.obj* sustituyendo al anterior. De esta forma se logra la variación deseada de la semilla. Esta clase debería ser ejecutada regularmente por un administrador del sistema o eventualmente por un mecanismo automático.

#### 4.2.5.5. Clase KeyMgmt

Esta clase tiene como propósito administrar el archivo que guarda las claves asociadas a cada PIN. La misma no tiene interfase gráfica y, por línea de comando, acepta las siguientes opciones:

`KeyMgmt –reset`

Esta opción vacía el archivo de claves completamente borrando todo su contenido.

`KeyMgmt –get <PIN>`

Con este comando se devuelve la clave asociada a un PIN que se pasa como parámetro.

`KeyMgmt –add <Nombre_archivo>`

Esta opción permite agregar claves al archivo. La misma recibe el nombre (y posiblemente la ruta) de un archivo de texto a partir del cual se leerá la clave que se ingresará. El mismo deberá tener los siguientes datos:

```
<PIN> <Largo> <Clave>
```

que consisten en un número cualquiera que será el PIN, el largo de la clave (en bytes) y la clave misma que debe ser una serie de números entre 0 y 255 representando los bytes que conforman la clave pública. La cantidad de bytes debe ser al menos igual al valor indicado en `<Largo>`, en caso contrario se producirá el error “Archivo incorrecto”. Si se introducen bits extras además de los indicados en `<Largo>` los mismos serán ignorados. Los tres datos deberán encontrarse en el archivo separados por uno o más espacios en blanco o fines de línea.

#### 4.2.5.6. Auxiliares

Este es un paquete que contiene algunas clases necesarias para implementar la autenticación utilizando RPK.

En general la aplicación de referencia realiza los procesos de generar claves, encriptar y desencriptar utilizando las mismas clases exactas que corren en la tarjeta. Esto se hace así debido a que ellas, en su

mayor parte, contienen código Java estándar que puede ser ejecutado por la JVM de Windows. Existe otro porcentaje de código que utiliza clases específicas del API de las *Java Cards* el cual no puede ser ejecutado fuera de la tarjeta sin tener los archivos .class correspondientes a las mismas. Estos archivos necesarios se pueden encontrar entre los correspondientes al emulador de la tarjeta y pueden ser agregados al *classpath* para ser usados desde Windows.

El problema en el caso de RPK radica en que existe cierta operación del API de *Java Card* (*Util.setShort*) que al ser ejecutada desde Java (fuera de la tarjeta) produce siempre una excepción. Por esta razón se colocaron en este paquete copias de las clases que corren en la tarjeta pero cambiando este método por una implementación local diferente que se ejecuta sin problemas en Java.

#### 4.2.5.7. Applet en la tarjeta

Además de estas clases ya mencionadas la aplicación de referencia está formada por un componente que se ejecuta en la tarjeta. Este es un applet muy sencillo y es con el cual se comunicará la clase *Principal* ya descrita.

Se programaron versiones diferentes del mismo, uno para cada uno de los criptosistemas probados, aunque en esencia, en lo relativo a esta aplicación de referencia, cada uno de ellos recibe los mismos APDUs a saber:

- CLA = 0x90, INS = 0x24
  Con este APDU se envía a la tarjeta la clave pública del sistema que va a realizar la autenticación. Como la misma puede realizarse con diferentes clientes es necesario que cada uno le envíe la suya antes de comenzar. En caso de que la misma sea siempre igual (pues siempre se autentica contra el mismo cliente) alcanzaría con llamarla una vez para luego utilizar el valor ya cargado en la tarjeta.
  En cualquier caso su única función es copiar los datos recibidos a la memoria interna de la tarjeta y como retorno devuelve solamente un valor 0xFE en caso de éxito.

- CLA = 0x90, INS = 0x26
  Con este APDU se realiza la autenticación en sí. El mismo recibe el valor aleatorio generado por el cliente luego de ser encriptado por el mismo. Su función consiste en desencriptarlo con la clave privada de la tarjeta, volverlo a encriptar con la pública del cliente (que debió haber recibido con el APDU anterior) y devolver este valor encriptado.

- CLA = 0x90, INS = 0x20
  Este es el que inicia el proceso de generación de claves. Debe ser llamado antes de poder ejecutar el proceso de autenticación para la que la tarjeta genere su propio par de claves pública/privada. En teoría este APDU se invocará antes de poner a la tarjeta en producción y luego no volverá es ser llamado.

- CLA = 0x90, INS = 0x23
  Este APDU es el que devuelve la clave pública generada por la tarjeta mediante el envío de un APDU del tipo anteriormente descrito. Este APDU se utiliza en el momento de asociar un PIN a una clave pública con el fin de obtener esta última de la tarjeta. Este APDU tiene algunos otros relacionados dependiendo del criptosistema que se muestran a continuación.

- CLA = 0x90, INS = 0x28
  Este es un APDU que se usa con los criptosistemas RPK y RSA. En el caso de RPK se utiliza para devolver el contenido de la componente 1 del generador de mezclas mientras que el anterior se utiliza para devolver el componente mezclador en sí (ver apéndice IV). Por otro lado, para RSA, este APDU retorna el exponente de la clave privada siendo el anterior el que retorna el módulo (ver apéndice V).

- CLA = 0x90, INS = 0x29
  Este es exclusivo de RPK y devuelve el contenido de la componente 2 del generador de mezclas (ver apéndice IV).

Además de estos APDUs las diferentes versiones de este applet reciben algunos otros que permiten devolver la clave privada de la tarjeta, devolver la clave pública del cliente que se le ha transmitido a la tarjeta y encriptar y desencriptar valores pasados como parámetro. Estos existen para habilitar el testeo los diferentes componentes del proceso de autenticación en forma independiente y verificar su funcionamiento. En particular el APDU que devuelve la clave privada de la tarjeta presenta un riesgo de seguridad muy importante y no debería existir en una aplicación real de este sistema.

### 4.2.6. Limitaciones

- En la implementación realizada durante el transcurso del proyecto no se programó ni se probó el hecho de realizar la autenticación remota desde otra máquina en la misma red en que se encuentra el lector con la tarjeta del usuario. De todas formas se cree que dicha funcionalidad se debería poder integrar al sistema presentado sin necesidad de grandes esfuerzos extra de diseño o implementación.
- El archivo donde el programa de autenticación almacena las claves asociadas a cada PIN no se guarda encriptado en el disco. De todas maneras el mismo tampoco es un archivo de texto, por lo que sin conocer el formato que se usa para almacenarlo, no es trivial obtener las claves por simple observación. De todas formas, en un sistema real esta archivo deberá ser encriptado (probablemente con la propia clave pública del sistema de autenticación) o, en su defecto, deberá ser un archivo al que solo pueda acceder un usuario administrador del sistema.
- En la implementación actual la aplicación de referencia no posee una clave pública fija (por ejemplo guardada en un archivo) sino que se genera en cada corrida de la autenticación. Esto es una limitación dado que retarda un poco el proceso, aunque de todas maneras la mayor parte del tiempo invertido en el mismo se ocupa en el procesamiento dentro de la tarjeta. La razón para esta decisión es que permitió probar mejor el funcionamiento de los criptosistemas implementados dado que las diferentes corridas usan diferentes claves. Esta limitación es solo para la aplicación que corre en la PC; la tarjeta sí genera sus claves solamente una vez[2].
- Otra limitación de seguridad se encuentra en el archivo donde se guarda el generador de números aleatorios serializado. Al igual que antes este archivo no está encriptado y el acceso al mismo podría permitir predecir los próximos valores generados y de esta forma facilitar un ataque al sistema..
- En el caso particular de RSA, el cual se trata de un sistema determinista (o sea que la encriptación de un cierto texto plano siempre produce el mismo texto cifrado), se tiene otra debilidad ya que si alguien puede guardar todas las parejas desafío-respuesta que se produzcan podrá contestar adecuadamente a cualquier desafío que esté en su base de datos. Esto se podrá lograr siempre que el generador de números aleatorios genere más de una vez los mismos datos. En nuestra implementación se generan 512 bits de información aleatoria por lo que la probabilidad de que se generen exactamente los mismos datos más de una vez es baja (siempre que el generador usado sea bueno)

## 4.3. Criptosistemas Implementados

En esta sección se dará una explicación más completa de los criptosistemas estudiados que fueron realmente implementados en la tarjeta y utilizados por la aplicación de referencia comentada anteriormente.

Como ya se dijo, todas las implementaciones se realizaron siguiendo el API de las *Java Cards* agregando nuestras implementaciones dentro del framework definido por *Sun*. En este framework se definen dos paquetes que contienen las clases que implementan funciones criptográficas; ellos son `javacard.security` y `javacardx.crypto`.

[2] Existe una versión de la aplicación de referencia que utiliza RSA que no tiene esta limitación por razones que se explicarán luego en el documento.

Ambos paquetes están cargados por defecto dentro de las tarjetas que utilizamos, conteniendo una implementación parcial de las funcionalidades del framework, pero los mismos no fueron borrados de las mismas para sustituirlos por otros dado que esta operación es irreversible. Por esta razón fue necesario crear paquetes nuevos con diferente nombre a fin de poder cargarlos en la tarjeta junto con los ya existentes. De esta forma las clases criptográficas implementadas en este proyecto se encuentran en dos paquetes equivalentes a los anteriores llamados `jcard.security` y `jcardx.crypto`.

En el paquete `jcard.security` se agregaron subclases de la clase `Key` correspondientes a las claves de los nuevos criptosistemas y se implementaron las clases `KeyPair` y `KeyBuilder` con los mismos métodos que en el API y con nuevos códigos de tipo y largo de las claves para permitir generarlas. La idea es que la clase `KeyBuilder` devuelva interfases `Key` ya implementadas y la clase `KeyPair` las cargue con valores válidos que ella genera. Por otro lado, en el paquete `jcardx.crypto` se implementó la clase `Cipher` con los algoritmos correspondientes para realizar el cifrado y descifrado de la información. En este caso la clase abstracta `Cipher` posee un método estático que devuelve implementaciones de sí misma según el criptosistema elegido.

En ambos casos se crearon también clases sin acceso público que contienen las implementaciones de cada criptosistema y que serán comentadas más en detalle a continuación cuando se reseñe cada algoritmo en particular.

## 4.3.1. NTRU

Este fue el primero de los criptosistemas implementados durante el proyecto. El mismo fue seleccionado a causa de que, al utilizar claves basadas en polinomios en lugar de números muy grandes, (como por ejemplo RSA) se facilita su implementación en la tarjeta considerando que la misma no maneja tipos de datos numéricos primitivos de más de 16 bits. Además el mismo utiliza claves un poco más cortas que otros criptosistemas lo cual simplificó el manejo de memoria.

### 4.3.1.1. Problemas a resolver

Como se puede ver en el apéndice III, el algoritmo de NTRU no es realmente muy complejo de por sí. Los procesos de codificación y decodificación requieren únicamente multiplicaciones de polinomios en módulo y multiplicaciones de polinomios por constantes. Por otro lado, la generación de claves es un proceso un poco más complejo básicamente por el hecho de que se requiere construir un polinomio que sea invertible con respecto a dos polinomios distintos.

El método utilizado para resolver este problema consistió en sortear polinomios aleatorios y tratar de hallar su inversa mediante una variante del algoritmo de Euclides extendido (ver 4.3.1.3). Si el algoritmo falla en obtener cualquiera de las inversas necesarias se sortea otro polinomio con las características requeridas por el algoritmo y se vuelve a probar. Luego de obtenido el polinomio y la inversa se realizan al resto de las operaciones requeridas las cuales, al igual que en el caso de la encripción y desencripción, no revisten gran complejidad.

### 4.3.1.2. Diseño de la solución

Paquete `jcard.security`:

Para esta implementación se crearon las clases `NTRUPublicKey` y `NTRUPrivateKey` como interfases que heredan de la interfase `Key` y que agregan los métodos necesarios para trabajar con las claves de NTRU. De esta forma la clase `NTRUPublicKey` tiene métodos para trabajar con el polinomio *h* así como con los parámetros *P* y *Q* del criptosistema y la clase `NTRUPrivateKey` tiene métodos para cargar y recuperar el polinomio *f*, su inversa $F_p$ y al igual que antes, *P* y *Q*. Estas interfases están implementadas respectivamente por las clases `NTRUPubKey` y `NTRUPrivKey` que no son públicas por lo que no serán conocidas nunca por los usuarios finales (estos las usarán a través de las interfases)

Además se incluyeron en este paquete otra serie de clases auxiliares con diferentes funcionalidades:

- `NTRUgenClaves`
  Es la clase donde se encuentran los métodos específicos para generar las claves. La misma es llamada por `KeyPair` en el momento que necesita nuevas claves.

- `NTRUextras`
  Contiene otros métodos más generales que utiliza el criptosistema. Dichos métodos son usados tanto por las clases en ese paquete como por las que se encuentran en `jcardx.crypto` por lo que esta es una clase pública.

- `Polinomio`
  Es un simple arreglo con tope que se utiliza para representar polinomios. El tope indica la cantidad de coeficientes del polinomio.

- `Polinomios`
  Esta clase es la que contiene las operaciones que se pueden realizar sobre el tipo `Polinomio`. Las mismas incluyen entre otras: producto, división y el cálculo de inversas.

Paquete `jcardx.crypto`:

En este paquete se tiene la clase `NTRUCipher` que es la implementación de la clase abstracta `Cipher` para NTRU. Al igual que en el caso anterior esta no es la clase que realiza las operaciones criptográficas en sí, sino que las mismas se encuentran en otra, la cual es llamada por la primera. En este caso la clase que implementa estas operaciones es `NTRUcodDecod`. De esta forma se logran separar las clases que implementan la compatibilidad con el API de las que implementan el algoritmo en sí.

A continuación se muestra el diagrama de clases correspondiente a estos dos paquetes (en el siguiente diagrama no se muestran los parámetros formales de los métodos por razones de espacio y para no perder claridad, los mismos pueden verse en la documentación Javadoc adjunta):

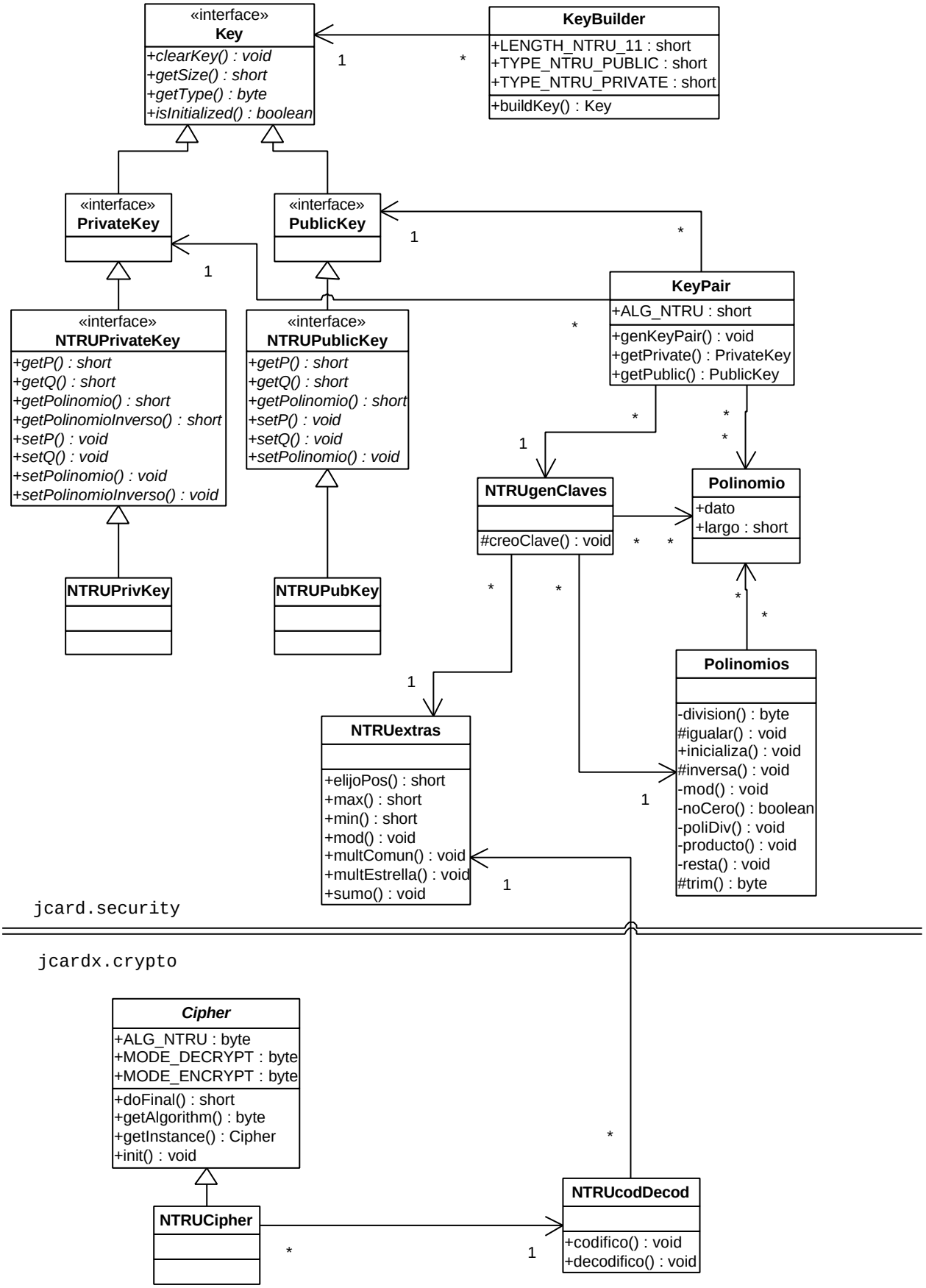



**Figura 5: Diagrama de clases de NTRU**

### 4.3.1.3. Algoritmos Usados

Como se dijo antes la encriptación y desencriptación usan operaciones simples de producto de polinomios y reducción en módulo que son explicadas con mayor profundidad cuando se describe el criptosistema en el apéndice III. Los algoritmos más complejos se usan en la parte de generación de claves en particular para la generación de las inversas de los polinomios.

Para esto se usó una adaptación para polinomios del algoritmo de Euclides extendido que en su versión original se utiliza para obtener inversas de números enteros. El seudo código del algoritmo usado es el siguiente, donde *pol1* es el polinomio a invertir, *pol2* es el polinomio módulo y *mod* es el módulo de los coeficientes del polinomio:

```
inversa (pol1, pol2, mod, resultado)
{
        t0 = 0
        t = 1
        n0 = pol2;
        q = ⌊n0/pol1⌋
        r = n0 – (q * pol1)
        while (r != 0) {
                temp = t0 – q * t
                x = ⌊temp/pol2⌋
                temp2 = temp – (x * pol2)
                t0 = t
                t = temp2
                n0 = pol1
                pol1 = r
                q = ⌊n0/pol1⌋
                r = n0 – (q * pol1)
        }
        if ((pol1 != -1) && (pol1 != (mod – 1)))
                pol1 no tiene inversa módulo pol2

        x = ⌊t/pol2⌋
        temp = t – (x * pol2)
        result = temp * pol1
}
```

Otro algoritmo que se pensó en usar por su rapidez y eficiencia fue el de la multiplicación de polinomios truncados usando una estrategia de divide y vencerás (*divide and conquer*). La multiplicación de dos polinomios de grado *n* insume en el orden de $n^2$ operaciones. El dividir los polinomios en 2 partes, para multiplicarlas por separado y luego juntarlas, puede reducir las operaciones a $¾n^2$ y si aplicamos éste método recursivamente las mejoras pueden ser aún mayores. Por una descripción más detallada de este algoritmo y su funcionamiento ver [4.1].

Los parámetros de entrada son tres polinomios (*a*, *b*, *c*), y dos números(*n*, *corte*). Los polinomios a ser multiplicados son *b* y *c* y el resultado de dicha multiplicación es *a*. El parámetro *n* es el grado de los polinomios *b* y *c*. El último parámetro, *corte*, es el que indica que tantas veces se procederá a dividir los polinomios, mientras *corte* sea menor que el grado *n* se procederá con la recursión.

```
polyMult(a, b, c, n, corte)
{
        if (n < corte){
                for (k=0; k<=(2*n-2); k++){
                        a[k]=0;
                        for (i=max(0,k-n+1); i<=min(k,n-1); i++)
                                a[k] += b[i]*c[(k-i)];
                }
        }
        else{
                n1 = n/2;
```

```
                n2 = n-n1;

                Polinomio a1 = new Polinomio(2*n1);
                Polinomio a2 = new Polinomio(2*n2);
                Polinomio a3 = new Polinomio(2*n2);
                Polinomio b1 = new Polinomio(n1);
                Polinomio b2 = new Polinomio(n2);
                Polinomio c1 = new Polinomio(n1);
                Polinomio c2 = new Polinomio(n2);
                Polinomio B = new Polinomio(n2);
                Polinomio C = new Polinomio(n2);

                //Escribo b como b=b1+b2*X^n1 y c como c=c1+c2*X^n1
                for (i=0; i<=(n1-1); i++){
                      b1[i]=b[i];
                      c1[i]=c[i];
                      b2[i]=b[(i+n1)];
                      c2[i]=c[(i+n1)];
                }
                if (n2==(n1+1)){
                      b2[(n2-1)]=b[(n-1)];
                      c2[(n2-1)]=c[(n-1)];
                }

                //B = b1 + b2 y C = c1 + c2;
                for(i=0; i<=(n1-1); i++){
                      B[i]=(b1[i]+b2[i]);
                      C[i]=(c1[i]+c2[i]);
                }
                if (n2==(n1+1)){
                      B[(n2-1)]=b2[(n2-1)];
                      C[(n2-1)]=c2[(n2-1)];
                }

                polyMult(a1,b1,c1,n1,corte);
                polyMult(a2,b2,c2,n2,corte);
                polyMult(a3,B,C,n2,corte);

                //a = a1 + (a3-a1-a2)*X^n1 + a2*X^(2*n1)
                for (i=0; i<a1.largo; i++){
                      a[i] = a1[i];
                      a[i+n1] += (a3[i]-a1[i]-a2[i]);
                }

                for (i=a1.largo; i<a2.largo; i++)
                      a[i+n1] += (a3[i]-a2[i]);

                for (i=0; i<a2.largo; i++)
                      a[i+2*n1] += a2[i];
          }
          if (((2*n-1) > N) && (N > 0))
                for (k=N; k<(2*n-1); k++)
                      a[(k-N)] += a[k];
    }//polyMult
```

### 4.3.1.4. Limitaciones

- La generación de claves es muy lenta en la tarjeta por el hecho de tener que sortear posiblemente muchos polinomios antes de encontrar uno válido y porque hay que realizar cálculos relativamente complejos en cada uno de ellos para saber si son correctos.
- A causa de esto se utilizó una versión muy reducida del algoritmo con polinomios de grado 11. Esta no es segura para su utilización en la práctica donde se recomienda usar al menos polinomios de grado 107.
- Se implementó la mejora al algoritmo de multiplicación de polinomios mencionada basada en *divide and conquer* pero la misma, si bien funciona en el emulador, no se pudo imple-

mentar en la tarjeta. El problema es que dicho algoritmo mejora la performance a costa de un consumo mayor de memoria y este costo, en nuestra implementación, era demasiado grande para la capacidad del hardware.

- Habría que investigar alguna forma mejor de generar las claves incluyendo métodos para detectar a priori si un cierto polinomio tiene inversa sin necesidad de realizar la propia inversión para determinarlo, pues el método utilizado actualmente es demasiado lento.

## 4.3.2. RPK

El segundo algoritmo en implementarse durante el proyecto fue el del criptosistema llamado RPK. El mismo fue seleccionado, por sobre todo, debido a que utiliza operaciones muy sencillas como ser el XOR y el manejo de *shift registers*. Por esta razón, aunque su implementación más eficiente sería en hardware, es un criptosistema que en teoría consume pocos recursos por lo que se adaptaba perfectamente a nuestras necesidades.

En segundo lugar, nuestra elección se basó en que la seguridad del criptosistema mencionado se sustenta en un tipo de problema matemático (problema del algoritmo discreto) distinto al del primer criptosistema implementado, NTRU.

### 4.3.2.1. Problemas a resolver

El algoritmo de RPK no resulta muy complejo de entender, sin embargo existen dos métodos (los cuales son el corazón del algoritmo) cuya implementación requiere un poco más de esfuerzo para comprenderse. Estos métodos son los encargados de, dado un cierto estado del generador de mezclas (mezclador de aquí en adelante), primero, avanzarlo una cantidad determinada de veces y segundo, dado un cierto número exponenciar el estado del mezclador a dicho número. Los principales procesos, la codificación, la decodificación y por supuesto la generación de claves, se basan en éstos dos métodos.

La exponenciación la podemos ver como el acto de avanzar el mezclador la distancia que existe entre el estado actual y el inicial una cierta cantidad de veces (el exponente). Así, al finalizar la exponenciación, el estado resultante estará a una distancia del estado inicial igual al exponente multiplicado por la distancia que existía entre el estado anterior y el inicial. Por información más detallada referirse al apéndice IV.

El método utilizado para resolver el problema de avanzar el mezclador consistió en descomponer en potencias de 2 al número de veces que se debía avanzar el mismo. Una vez realizada dicha descomposición se procede a avanzar el mezclador hasta la mayor potencia de la descomposición mientras se van guardando todos las estados correspondientes a las potencias de 2 intermedias. Al culminar con ésta tarea se efectúa la multiplicación[3] de los estados de las potencias halladas correspondientes a la descomposición.

El siguiente método, utilizado para resolver el problema de exponenciar un cierto estado del mezclador, es muy parecido al método recientemente explicado solo que en lugar de avanzar el mezclador un número de veces dado, lo que se hace es multiplicar por sí mismo el estado dado, tantas veces como la mayor potencia que la descomposición del exponente nos arroje y luego multiplicar por las potencias de dos intermedias correspondientes al exponente.

### 4.3.2.2. Implementación del generador de mezclas

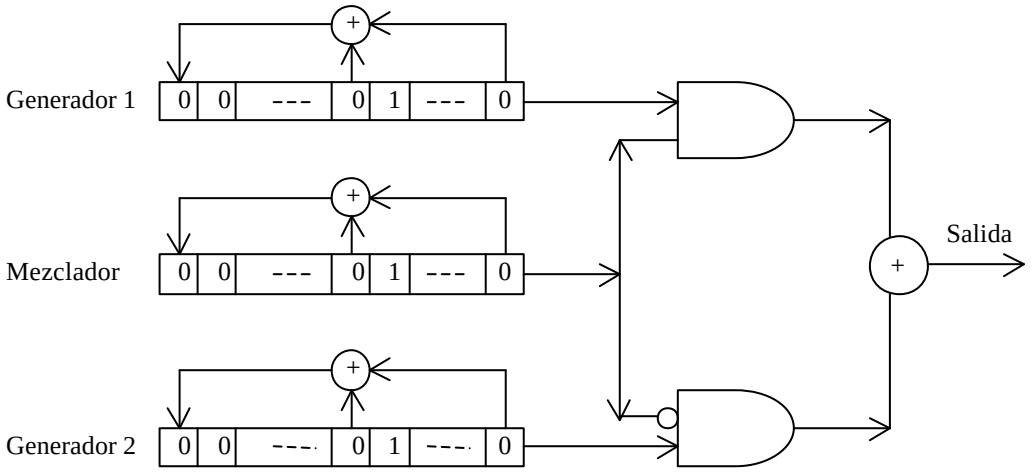
Para realizar la implementación se utilizó un generador de mezclas de tipo *Geffe*[5.1]. Este tipo de generadores se caracteriza por tener tres componentes separados, dos de los cuales brindan las posibles salidas del generador de mezclas mientras que el restante es el encargado de elegir cual de las dos

[3] La multiplicación entre dos estados A y B consiste en avanzar el mezclador la distancia que existe entre B y el estado inicial tomando como estado de partida A.

salidas posibles será la utilizada, es por ello que recibe el nombre de *generador mezclador* (mixer generator).

En nuestro caso implementamos un generador de largo 87 y otro de 89, mientras que el generador mezclador es de largo 127. Estos largos específicos fueron utilizados de acuerdo a lo que está planteado en [5.1].

A continuación se presenta un esquema de la representación del generador de mezclas implementado (recordar que aquí el signo de más representa un XOR, el círculo pequeño un NOT y el otro símbolo redondeado es un AND):




**Generador Geffe (de [5.1])**

Tanto los generadores como el mezclador, en su estado inicial, solamente cuentan con un número 1. Dicho número está situado en el lugar siguiente al que ocupa el sumador de cada *shift register* mientras que el resto de los lugares están ocupados por ceros

.

En nuestro caso el sumador del generador 1 se encuentra en el lugar 13 del *shift register*, de igual forma el sumador del generador 2 esta ubicado en la posición 38 mientras que el del mezclador se ubica en el lugar 30.

### 4.3.2.3. Diseño de la solución

Paquete `jcard.security`:

Para poder trabajar con las claves generadas por RPK se crearon las clases `RPKPublicKey` y `RPKPrivateKey` las cuales se comportan como interfases que heredan de la interfase `Key`. Así la clase `RPKPublicKey` cuenta con métodos para cargar y recuperar los estados del generador de mezclas, es decir, los estados del mezclador y de los generadores. De igual forma la clase `RPKPrivateKey` tiene métodos para el manejo de la clave privada la cual consiste de tres números cada uno de los cuales indica la cantidad de veces que hay que avanzar el mezclador, el generador 1 y el generador 2 a partir del estado inicial para llegar al estado correspondiente a la clave pública.

`RPKPubKey` es la clase encargada de implementar la interfase `RPKPublicKey` mientras que la clase `RPKPrivKey` se encarga de la implementación de la interfase `RPKPrivateKey`.

Las siguientes clases auxiliares también fueron incluidas en este paquete dado que su funcionalidad era necesaria:

- `RPKgenClaves`
  Es la clase donde se encuentran los métodos específicos para generar las claves. La misma es llamada por `KeyPair` en el momento que necesita nuevas claves.

- `RPKextras`
  Contiene otros métodos más generales que utiliza el criptosistema. Dichos métodos son usados tanto por las clases en ese paquete como por las que se encuentran en `jcardx.crypto` por lo que esta es una clase pública.

- `Mixture`
  Es la maquinaria en la que se basa el criptosistema. Contiene 3 arreglos que representan el generador de mezclas (mezclador, generador 1 y generador 2) así como también todas las operaciones necesarias para trabajar con ellos, setear y recuperar los estados, recuperar tamaño de los arreglos y por supuesto avanzar (ver glosario: avanzar) cada arreglo para de esa forma avanzar el generador completo.

Paquete `jcardx.crypto`:

En este paquete se tiene la clase `RPKCipher` que es la implementación de la clase abstracta `Cipher` para RPK. Al igual que en el caso anterior esta no es la clase que realiza las operaciones criptográficas en sí, sino que las mismas se encuentran en otra, la cual es llamada por la primera. En este caso la clase que implementa estas operaciones es `RPKcodDecod`. Así, al igual que fue mencionado anteriormente, se logran independizar las clases que implementan la compatibilidad con el API de las que implementan el algoritmo en sí.

A continuación se muestra el diagrama de clases correspondiente a estos dos paquetes (al igual que antes el mismo no muestra los parámetros de cada uno de los métodos):

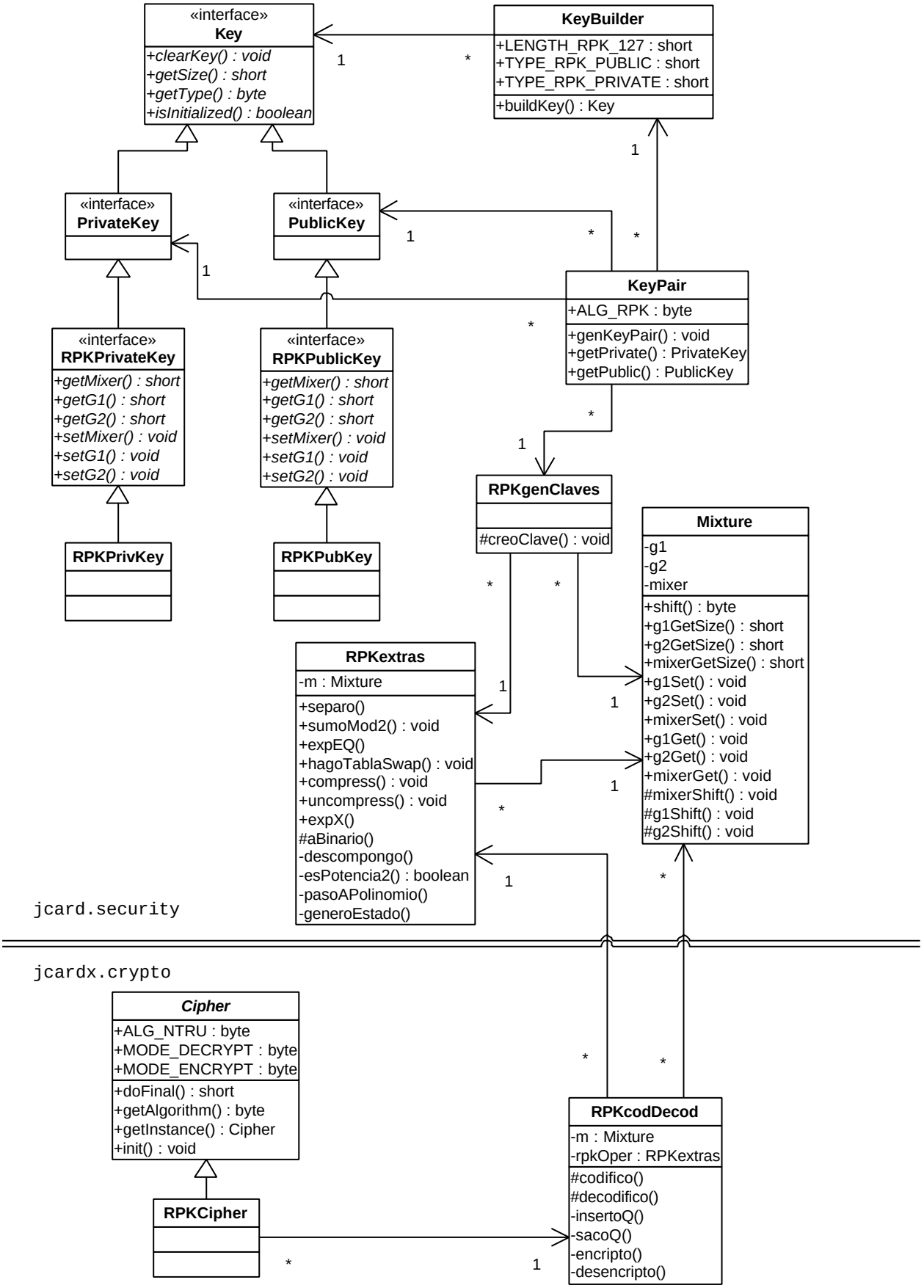



**Figura 6: Diagrama de clases de RPK**

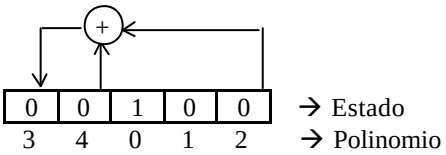
### 4.3.2.4. Algoritmos Usados

Al igual que el criptosistema NTRU, la encriptación y la desencriptación no revisten mayor complejidad, simplemente hacen uso de dos métodos básicos *expX* y *expEQ*. Estos son los métodos más complejos del criptosistema.

A continuación presentaremos un ejemplo y el seudocódigo del método encargado de, dado un cierto estado del generador de mezclas y un número, avanzar el generador dicha cantidad de veces, el nombre del método es *expX*.

Ejemplo:

Sea el siguiente generador de largo = 5,




Sea $D$ = 24 la cantidad de veces que hay que avanzar el generador y $E$ = (00100).
Descompongo $D$ en sus potencias de 2 → 24 = 16 + 8.
Avanzo el generador 16 veces (correspondiendo a la mayor potencia de la descomposición).
Guardo las potencias intermedias, es decir, 1, 2, 4 y 8.
La tabla de los estados guardados es la siguiente:

| Potencia | Estado |
|---|---|
| 1 | 00010 |
| 2 | 00001 |
| 4 | 01000 |
| 8 | 11010 |
| 16 | 10001 |

**Tabla 2: Estados calculados**

Ahora solamente nos queda por realizar la multiplicación de los estados correspondientes a las potencias 16 y 8. Dicha multiplicación nos da como resultado el estado (11001) el cual es el resultado de avanzar el estado $E$ $D$ veces consecutivas.

La multiplicación se realiza de la siguiente forma:
El estado correspondiente a avanzar el generador 8 veces es 11010 que lo podemos ver como el polinomio 01011 que es igual a decir $0.x^0 + 1.x^1 + 0.x^2 + 1.x^3 + 1.x^4$. Esto nos indica que debemos avanzar el generador tantas veces como la potencia de x que tenga como coeficiente 1, en este caso 1, 3 y 4.
Así es que debemos cargar el generador con el estado correspondiente a la potencia 16 y avanzarlo 1 vez para obtener el estado (11000).
Este proceso lo repetimos para 3 y 4 obteniendo los estados (11110) y (11111).
Por último sumamos con álgebra módulo 2 estos tres estados y llegamos al resultado de la multiplicación (11001).

Los parámetros de la función implementada son los siguientes:

D es el número de veces que hay que avanzar el generador.
gen es el encargado de diferenciar entre el mezclador, el generador1 y el generador2.
estadoI es el estado inicial.

```
expX(D, gen, estadoI){
        potencias = Descompongo D en sus potencias de 2;
        estado_aux = potencias[0];
        seteo_mixture(gen, estadoI);
```

```
        estado_potencias=calculo los estados de cada
        potencia(potencias);
        for (i=1; i<cantidad de potencias de 2; i++){
               estado_aux = Multiplico(estado_aux, estdo_potencias[i]);
        }
        return(estado_aux);
}
```

#### 4.3.2.5. Limitaciones

- La seguridad de la implementación es muy débil dado que la misma utiliza un generador de mezclas demasiado pequeño. La razón para esto fue que, en nuestra implementación, los tiempos de ejecución necesarios para correr una versión más segura son mayores que el tiempo máximo que admite la tarjeta para procesar un APDU exitosamente. Por esto las operaciones con la versión más segura no llegaban a completarse. El generador de mezclas que se usó finalmente tiene largos 2, 3 y 5 para los dos generadores y el mezclador respectivamente y los sumadores se encuentran en los lugares 0, 1 y 1 de cada uno de ellos.
- Por esta misma razón los valores aleatorios que deben ser sorteados son elegidos entre un grupo reducido de valores pequeños con el fin de reducir el tiempo de procesamiento.

### 4.3.3. RSA

En este caso no se realizó una nueva implementación del criptosistema sino que se utilizó la que ya está integrada en la tarjeta. Esto nos permitió, a la hora de realizar las comparaciones, tener también medidas de un algoritmo implementado por los propios fabricantes de la tarjeta para poder confrontarlo con los implementados durante el proyecto.

Lo que se hizo en este caso, en lugar de implementar nuevos paquetes, fue utilizar los que ya tiene incluidos la tarjeta y adaptar nuestra aplicación de referencia para que los utilizara.

#### 4.3.3.1. Problemas a resolver

El problema principal con este algoritmo fue que nuestra aplicación de referencia necesita también encriptar y desencriptar información fuera de la tarjeta, es decir que se debe tener acceso al código (o a los .class) del criptosistema para poder ser ejecutado externamente. En este caso claramente la dificultad radicaba en la imposibilidad de obtener (al menos por medios no destructivos) el código fuente que implementa RSA en la tarjeta.

Una solución a este problema podría haber sido la de implementar una nueva versión de RSA para ser usada externamente, aunque dicha medida alteraría un poco la idea inicial de probar una implementación completamente independiente a las nuestras con el fin de compararlas. La respuesta a esta cuestión fue la de utilizar las clases de Java que usa el emulador de la tarjeta para simular las operaciones criptográficas fuera de la misma. De esta forma se obtuvo una implementación funcional fuera de la tarjeta realizada también por los fabricantes de la misma.

Si bien esta fue la solución final empleada, requirió cierto esfuerzo extra dado que las clases del emulador de *Gemplus* están programadas de forma que las mismas *no* puedan ser usadas fuera del propio emulador. De esta forma, para lograr utilizarlas, hubo que recurrir a descompilar el código de las mismas a fin de entender como funcionaban y evitar sus medidas de protección para impedir usos externos.

#### 4.3.3.2. Virtudes y limitaciones

Como se verá mejor en la parte de comparaciones, al utilizar los algoritmos incluidos en la propia tarjeta, esta implementación de la aplicación de referencia es muy rápida, y ocupa menos memoria dado que el código ya está en la tarjeta y no es necesario cargarle paquetes extra (a excepción por supuesto del applet que realiza la autenticación)

En lo que respecta a la implementación externa, la misma no incluye ninguno de los algoritmos matemáticos más complejos necesarios para hacer funcionar el criptosistema (ej. los de generación de números primos grandes o el de exponenciación en módulo) sino que utiliza las implementaciones ya incluidas en la clase `BigInteger` que pertenece al API estándar de Java. Esto trajo algunos problemas dado que, para la generación de claves, dichas clases crean una instancia de la clase `SecureRandom` la cual, como se dijo anteriormente, tarda un tiempo sensible en obtener su semilla. Por esta razón, en este caso también se realizaron mediciones con una variación de la aplicación de referencia que en lugar de crear sus claves públicas en cada ejecución las lee cada vez de un archivo en el disco.

## 4.4. Comparaciones

Como resultado directo del estudio e implementación realizada de los diferentes criptosistemas surgió la necesidad de compararlos con el fin de obtener una apreciación de sus ventajas y desventajas relativas. Las comparaciones se hicieron midiendo el tiempo que tardaba la autenticación utilizando la aplicación de referencia, así como viendo la cantidad de memoria que consumió en la tarjeta la implementación de los mismos. Además se investigaron las vulnerabilidades y seguridad de cada uno de ellos

### 4.4.1. En la aplicación de referencia

Para cada criptosistema se realizaron una serie de corridas de prueba las cuales fueron cronometradas. Se hicieron en total 20 corridas en cada caso, 10 en los que la autenticación fue correcta y 10 en los que la misma falló. Todas las pruebas fueron realizadas en la misma máquina (Pentium III 550 MHz) corriendo Windows NT y su tiempo fue cronometrado usando las facilidades del programa *Matlab*.

Al realizar las mediciones se tuvo en cuenta el hecho de tener la menor cantidad posible de procesos corriendo en el sistema, de modo de tratar de minimizar la competencia de otros procesos con el de autenticación y así obtener valores más confiables.

Para cada algoritmo se calcularon tres intervalos de confianza al 95%: uno para las autenticaciones correctas, otro para las incorrectas y el tercero para la unión de ambos casos. Como se mencionó anteriormente para el caso de RSA se hicieron dos grupos de corridas, uno en el que el cliente externo generaba la clave cada vez y otro en el que las mismas eran leídas de un archivo.

Los resultados obtenidos fueron los siguientes:

| | NTRU | RPK | RSA (claves) | RSA (sin claves) |
|---|---|---|---|---|
| Autenticación Correcta | [16.894, 17,080] | [63.494, 64.792] | [23.614, 24.830] | [13.183, 13.241] |
| Autenticación Incorrecta | [16.878, 17.088] | [63.466, 64.806] | [23.836, 24.572] | [12.208, 12,268] |
| Total | [16.831, 17.048] | [63.778, 64.662] | [23.893, 24.533] | [12.491, 12.959] |

**Tabla 3: Intervalos de confianza del tiempo de ejecución de los diferentes criptosistemas**

Aquí se puede ver que los mejores resultados son obtenidos por RSA cuando no se generan las claves cada vez con un tiempo promedio menor a los 13 segundos. NTRU también logra buenos tiempos, sólo un poco mayores a los de RSA. Es claro que la instanciación de la clase `SecureRandom` al gene-

rar las calves de RSA en la PC afecta notoriamente los tiempos casi duplicando el tiempo que tarda el proceso de autenticación. Por otro lado la implementación por lejos más ineficiente es la de RPK que tarda más de un minuto lo cual la hace realmente inútil para cualquier aplicación práctica.

Para ver las medidas concretas y los cálculos realizados ver el apéndice VII.

### 4.4.2. Memoria

Otra de las medidas realizadas fue la del espacio que ocupa la implementación de los diferentes criptosistemas en la tarjeta, ya sea con respecto al código mismo así como también al espacio necesario para instanciar las clases y ejecutar el programa. Junto con esto se consideró también el espacio ocupado por la pequeña applet con la que interactúa la aplicación de referencia para ejecutar las operaciones necesarias en la tarjeta y recibir los resultados.

| | NTRU | RPK | RSA |
|---|---|---|---|
| Código binario (bytes) | 5826 | 8512 | 1038 |
| Programa corrido (bytes) | 11995 | 15650 | 4278 |

**Tabla 4: Memoria ocupada por cada implementación**

En este caso se puede ver que la implementación de RSA requiere mucha menos memoria a causa de que la misma no necesita cargar código de criptografía pues el mismo ya se encuentra en el API de *Java Card* pre-instalado en la tarjeta. Por otro lado, las implementaciones realizadas íntegramente por nosotros ocupan una cantidad sensiblemente mayor de memoria y entre ellas RPK ocupa alrededor de 4 k más que NTRU tanto antes como después de instanciar las clases.

### 4.4.3. Seguridad

Como último punto se evaluaron las diferencias entre los distintos criptosistemas con respecto a la seguridad de cada uno de ellos. Esto permite darle una cierta perspectiva a las medidas de tiempo mostradas en 4.4.1 dado que los tiempos por si solos no proporcionan ninguna información de peso.

En esta sección se analizaron las características del criptosistema en sí mismo y también las de las implementaciones particulares realizadas por nosotros.

#### 4.4.3.1. Seguridad del criptosistema

NTRU:

Como se explica con más detalle en el apéndice III, este criptosistema es considerado seguro ante los ataques conocidos cuando el valor de $N$ es mayor a un valor aproximado de 100. Los autores recomiendan, basados en resultados empíricos que se utilicen estos valores:

| | $N$ | $p$ | $q$ | Largo clave privada (bits) | Largo clave pública (bits) |
|---|---|---|---|---|---|
| Seguridad moderada | 107 | 2 | 64 | 340 | 642 |
| Seguridad alta | 167 | 3 | 128 | 530 | 1169 |
| Seguridad muy alta | 503 | 3 | 254 | 1595 | 4024 |

**Tabla 5: Parámetros recomendados para NTRU**

RPK:

Considerando los ataques conocidos en la actualidad, este criptosistema se considera seguro cuando el componente más largo del generador (ver apéndice IV) mide por lo menos 607. Se estima que claves

de largo 521 o menor pueden llegar a ser quebradas en unos pocos años, por lo que no pueden ser usadas para encriptar información a largo plazo.

RSA:

Siendo este uno de los criptosistemas más difundidos en la actualidad, se han realizado muchas investigaciones con respecto a su seguridad y a formas de atacarlo. Actualmente se considera que para obtener un buen nivel de seguridad se deben usar claves de al menos 1024 bits, aunque para aplicaciones que no requieren gran seguridad se podrían utilizar claves de 512 bits. De hecho en 1999 se tardó más de 7 meses en quebrar por factorización una de estas claves.

#### 4.4.3.2. Seguridad de la implementación

NTRU:

La implementación de este algoritmo no presenta niveles realistas de seguridad a causa de que usa largos de claves demasiado cortos. La razón para esto, como ya se comentó, es que la generación de claves de mayor largo es demasiado ineficiente con el método de generación implementado en la tarjeta.

RPK:

Nuestra implementación de RPK en la tarjeta tiene seguridad bastante limitada ya que no es más que una prueba del algoritmo con valores muy pequeños. Como ya se dijo, su tiempo de ejecución con niveles de seguridad adecuados es demasiado extenso y no puede ser corrido en la tarjeta utilizada.

RSA:

Esta es la implementación más segura y más rápida de las comparadas aquí. Como se comentó anteriormente sus 512 bits la dotan de una seguridad moderada pero aceptable en algunas aplicaciones prácticas.

# 5. Conclusiones

El presente trabajo permitió obtener, como primer objetivo, un panorama general del estado del arte en cuanto a sistemas de criptografía de clave pública y su adecuación para ser implementados en dispositivos de bajos recursos. Esto nos condujo, como segundo objetivo, a tratar de implementar y testear en las propias tarjetas algunos de estos criptosistemas y, posteriormente, plantearnos el obtener una cierta medida comparativa de algunos de ellos.

La experiencia adquirida nos permite indicar que la implementación de algoritmos medianamente complejos en el entorno de sistemas de bajos recursos, es una tarea bastante difícil, a causa de la falta de potencia de procesamiento y memoria disponible. Esto dificultó la implementación de los algoritmos criptográficos aunque los mismos en sí no sean tan complicados de implementar en ambientes con recursos computacionales más holgados.

Con respecto a las propias tarjetas, lo más significativo que pudimos concluir fue que su programación es una tarea ardua fundamentalmente debido a que es difícil encontrar los errores de programación en las aplicaciones que corren en las mismas. Para llevar adelante esta labor se contó con el kit de desarrollo de *Gemplus* y su emulador de la tarjeta, que si bien fueron de importante ayuda, todavía están muy lejos de los *debuggers* más simples para ambientes de programación normales. En particular el emulador no es muy confiable dado que hay mucho código que corre en él pero no en la tarjeta, es decir, el emulador no cumple completamente con su función.

Por otra parte, la implementación específica de la JCVM tampoco ayudó mucho, dado que, como la misma no dispone de recolección de basura y Java no permite devolver la memoria pedida, se debe solicitar toda la memoria necesaria al construir las clases. Esto dificulta la programación, oscurece el código e impide lograr mayores ahorros de memoria.

Desde un punto de vista más general hay que tener en cuenta que, dados los escasos recursos con que cuentan actualmente las tarjetas y la necesidad de obtener implementaciones de algoritmos criptográficos veloces y eficientes que corran sobre las mismas, dichas implementaciones en la práctica se llevan a cabo utilizando ayudas especiales en hardware (como es el caso de la tarjeta usada con su coprocesador de RSA) dado que es muy complicado lograr buenos tiempos de ejecución (manteniendo la seguridad) de otra manera. Por otro lado la programación en sí de la tarjeta se limita en general a applets más sencillas que solamente hacen uso de las operaciones criptográficas ya programadas.

En lo que refiere a la utilidad general del proyecto consideramos que, en primer lugar, el mismo puede servir como una referencia básica de los algoritmos criptográficos que se utilizan en tarjetas inteligentes o que están pensados para funcionar en ambientes de bajos recursos. Además se estudió el API de criptografía de las *Java Cards* así como la implementación del mismo que posee el modelo particular de tarjeta utilizado durante el proyecto.

En segundo lugar, se dejaron disponibles implementaciones de algunos de los criptosistemas estudiados, las cuales, más allá de la mayor o menor eficiencia de su funcionamiento en la tarjeta, pueden servir de base para futuras mejoras de los mismos o su aplicación en otras tareas o proyectos, ya sean también sobre tarjetas inteligentes o en cualquier otra área en la cual la criptografía tenga relevancia.

En lo que respecta a nosotros, al llegar al final del proyecto nos encontramos con que el mismo nos ha dejado alguna experiencia en el desarrollo de aplicaciones para ambientes de bajos recursos, lo que implica tomar en consideración temas como el consumo de memoria y la eficiencia de los algoritmos implementados, lo cual muchas veces no se tiene en cuenta cuando se programan aplicaciones para entornos convencionales. Además de esto, adquirimos conocimientos sobre la programación en *Java Cards* así como su funcionamiento y la forma de comunicación de las aplicaciones externas con las mismas.

Como último punto, pero no menos importante, también tuvimos la oportunidad de investigar diferentes algoritmos criptográficos viendo sus ventajas y debilidades y al mismo tiempo lograr comprender su funcionamiento con el fin de intentar obtener implementaciones eficientes de los mismos.

# 6. Trabajo futuro

El trabajo que se podría hacer como continuación de este proyecto o a partir del mismo es diverso, ya que nosotros tuvimos que restringirnos a algunos temas específicos por razones obvias de tiempo y alcance del proyecto.

Un punto inicial sería que se podría continuar con la investigación de los diferentes criptosistemas estudiados aquí, ya sea buscando nuevos algoritmos o implementaciones mejoradas de los mismos. Relacionado con esto estaría la investigación de nuevas formas de ataque a dichos algoritmos, así como la búsqueda de nuevos criptosistemas que puedan ser implementados apropiadamente en sistemas de bajos recursos.

Asimismo, nosotros nos concentramos en algoritmos de cifrado, mientras que es muy razonable también la investigación en otro tipo de algoritmos, como ser los de firma digital, que también tienen una aplicabilidad práctica muy importante en el ambiente de las tarjetas inteligentes.

En lo que respecta a la aplicación de referencia, la misma no deja de ser una prueba de concepto explorando la comunicación de los programas externos con la tarjeta e intentando mostrar un uso posible de los algoritmos implementados en el proyecto. Esta aplicación podría ser extendida de muchas maneras, una de las cuales podría ser, por ejemplo, el permitir el uso de la misma a través de la red para permitir autenticaciones remotas (a sitios web, por ejemplo) a partir del uso de una tarjeta inteligente en la máquina del cliente.

# Apéndice I: ECC

## Introducción a las curvas elípticas

Dado un cuerpo $K$ y siendo $\overline{K}$ su clausura algebraica se define el plano proyectivo $P^2(K)$ como el conjunto de clases de equivalencia en $K^3/\{0,0,0\}$ (o sea ternas de valores de $K$ distintas de la $\{0,0,0\}$) donde $\{x_1, y_1, z_1\}$ está relacionado con $\{x_2, y_2, z_2\}$ si y sólo si existe un $u$ perteneciente a $K^*$ (léase todos los elementos de $K$ con inverso multiplicativo) tal que $x_1 = u.x_2$, $y_1 = u.y_2$ y $z_1 = u.z_2$

Dentro de este conjunto se define una ecuación de Weierstrass como:

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$$

donde $a_1$, $a_2$, $a_3$, $a_4$ y $a_6 \in \overline{K}$ . Una curva elíptica (también llamada curva cúbica no singular) se define entonces como el conjunto de todas las soluciones de una ecuación de Weierstrass. A una solución cualquiera se le llama un punto de la curva elíptica. Toda curva tiene una solución en la clase de equivalencia de $\{0, 1, 0\}$ a la que se le llama punto en el infinito y se representa por $O$. Por más información sobre este tema ver [7.2].

Cuando todos los $a_i$ pertenecen a $K$, se dice que la curva está definida sobre $K$ y se nota como $E(K)$ al conjunto de puntos de la curva cuyas coordenadas son elementos de $K$ junto con $O$.
A partir de esta ecuación se puede hacer un cambio de variables con

$$x = \frac{X}{Z} \quad y = \frac{Y}{Z}$$

llamadas coordenadas affine y obtener entonces la ecuación anterior pero con la forma:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

cuyas soluciones pertenecen a $\overline{K} \times \overline{K}$ junto con $O$.

Cuando la característica del cuerpo $K$ es diferente de 2 o 3 la ecuación que representa una curva puede ser modificada por ciertos cambios de variables que la simplifican enormemente.

Por ejemplo, si la característica es distinta de 2 se puede efectuar el siguiente cambio de variables:

$$(x, y) \rightarrow (x, y - \tfrac{1}{2}(a_1x + a_3))$$

a partir del cual, luego de varias operaciones, la ecuación queda de la forma:

$$y^2 = x^3 + \tfrac{1}{4}b_2x^2 + \tfrac{1}{2}b_4x + \tfrac{1}{4}b_6$$

donde se tiene que:

$$b_2 = a_1^2 + 4a_2$$
$$b_4 = 2a_4 + a_1a_3$$
$$b_6 = a_3^2 + 4a_6$$

Hay que notar que si la característica del cuerpo fuera dos, para cualquier $k$ del cuerpo se tendría que $2.k = 0$ por lo que la división entre dos no estaría definida.

En el caso de que la característica sea también diferente de 3 se puede hacer este cambio de variables:

$$(x, y) \rightarrow (\frac{x - 3b_2}{36}, \frac{y}{216})$$

Que permite obtener la ecuación:

$$y^2 = x^3 - 27c_4 x + 54c_6$$

con:

$$c_4 = b_2^2 - 24b_4$$
$$c_6 = b_2^3 + 36b_2 b_4 - 216b_6$$

En este caso, al igual que en el anterior, si la característica del cuerpo fuera 3, todos los múltiplos de tres (en particular 36 y 216) serían equivalentes al 0 por lo que no se podría dividir.

En conclusión la curva puede escribirse como:

$$y^2 = x^3 + ax + b$$

Para definir la estructura de grupo sobre estas curvas se tiene que tener una suma de puntos de una curva elíptica *E* la cual se define de la siguiente manera:

1. $P + O = P$ y $O + P = P$
2. Si $P \neq O$ y $Q \neq O$, sea *l* la línea que conecta a *P* y *Q* (si P ≠ Q) o la tangente a la curva en *P* (si *P* = *Q*). Entonces *R* será el tercer punto de intersección de *l* con *E* (contando las multiplicidades) Esto quiere decir que, o hay un tercer punto en la intersección de *l* y *E* distinto de *P* y *Q* o *l* es tangente a la curva en *P* (o *Q*) y *R* = *P* (o *R* = *Q*).
    a. Si $R = O$ entonces $P + Q = O$
    b. Si no, sea *l'* la línea que conecta a *R* y *O*. Entonces *P* + *Q* se define como el tercer punto en la intersección de *l'* y *E*, contando las multiplicidades.

Se puede probar que (*E*,+) con esta definición forma un grupo abeliano con identidad *O*.

Como ejemplo se tiene la Figura 7, que muestra una curva definida sobre los números reales, en la que el punto en el infinito se encuentra en cada línea vertical. Aquí se muestran tres ejemplos de sumas que son los siguientes:

1. $P + Q$ donde $P \neq Q$. La línea que conecta a *P* y a *Q* es *l*. El tercer punto en la intersección de *l* con la curva *E* es *R*, el cual es diferente de *O*. La línea que conecta a *R* y a *O* es *l'* y el tercer punto de intersección de *l'* con la curva (además de *O* y *R*) es el que equivale a *P* + *Q*.

2. $S + S = 2S$. Entonces la línea que se debe trazar es *m* que es la tangente a *E* en *S*. El tercer punto de la intersección (contando la multiplicidad 2 del punto *S*) es *T*. Entonces la línea que conecta a *T* y *O* es *m'* y el tercer punto de intersección con la curva es 2*S*.

3. $T + S$. Como antes la línea que conecta a *T* y a *S* es *m*. Como la línea es tangente a la curva en S este punto tiene multiplicidad 2 y es el tercer punto de corte. Por lo tanto se debe trazar una recta vertical en *S* cuya tercera intersección con *E* es el punto *T* + *S*.

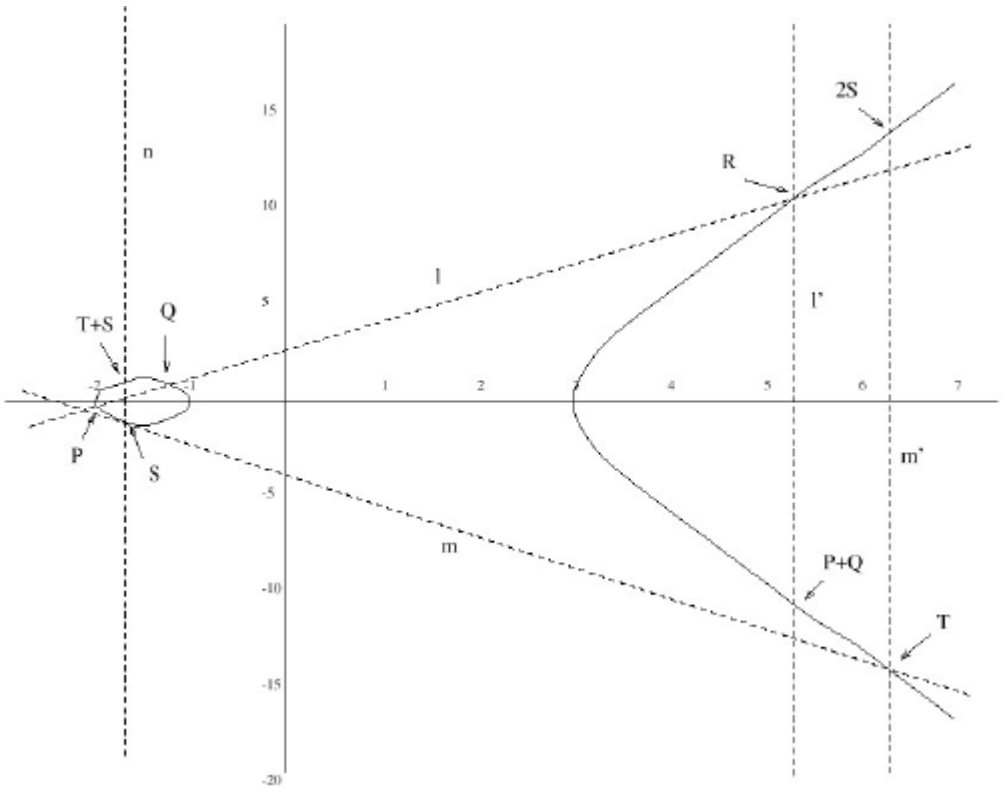



**Figura 7: Ejemplos de sumas en curvas elípticas (de [7.2])**

La suma también se puede definir analíticamente en función de las coordenadas. Para cuerpos con característica distinta de 2 o 3 se tiene que si $P = (x_1, y_1)$, $Q = (x_2, y_2)$ entonces $P + Q = (x_3, y_3)$ donde:

$$-P = (x_1, -y_1)$$

y en otro caso:

$$x_3 = l^2 - x_1 - x_2$$
$$y_3 = l(x_1 - x_3) - y_1$$
$$l = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & si\ P \neq Q \\ \dfrac{3x_1^2 + a}{2y_1} & si\ P = Q \end{cases}$$

Por otra parte también se puede definir la multiplicación escalar sobre las curvas elípticas de la manera obvia o sea:

$$mP = \begin{cases} P + \ldots + P & si\ m > 0 \\ O & si\ m = 0 \\ (-m)(-P) & si\ m < 0 \end{cases}$$

Y a partir de esto, dado que es más fácil calcular $mP$ dados los dos valores que hallar $m$ una vez hecho el producto, se tiene toda la estructura matemática necesaria para definir el logaritmo discreto en curvas elípticas.

Nótese que, en este caso, la operación que se utiliza como logaritmo es la que sería la división en esta estructura definida en las curvas elípticas.

A continuación se mostrarán dos criptosistemas basados en este problema.

# ElGamal para curvas elípticas

Este criptosistema es una variación del sistema de ElGamal [8.1] clásico pero adaptado para curvas elípticas.

Para usar este criptosistema se debe elegir un número $p$ primo de forma que $q = p^m$ para algún entero $m$. Entonces se toma una curva elíptica $E(\mathrm{GF}_q)$ y un punto $P$ perteneciente a $E$ y de orden $n$. Se dice que el orden de un punto es $n$ si $n$ es el menor número que cumple que $nP = O$. Estos números deben ser seleccionados de forma que el problema del logaritmo discreto no sea tratable para $P$.

La clave pública consiste entonces en $E(\mathrm{GF}_q)$, $P$, $n$ y $Q_U = l_U P$ para cada usuario $U$. La clave privada de este usuario sería el valor $l_U$. La característica de $E(\mathrm{GF}_q)$ es $p$ por lo que si $p > 3$ por lo visto anteriormente la curva elíptica puede ser representada por dos valores $a$ y $b$ pertenecientes a $\mathrm{GF}_q$.

Encriptación: Suponiendo que $A$ quiere mandar a $B$ un mensaje $M \in E(GF_q)$ los pasos son los siguientes:

1. $A$ genera un número aleatorio $k$ perteneciente a {2,3, ..., n-2} y calcula $R = kP$
2. $A$ calcula $S = M + kQ_B$, siendo $Q_B$ la clave pública de $B$
3. $A$ envía $(R, S)$ a $B$

Nótese que $k$ no puede ser 0, 1 o $n$–1 pues en estos casos $kQ_B$ sería igual a $O$, $P$, y $–P$ respectivamente por lo que podría resultar muy sencillo obtener $M$ a partir de $S$.

Desencriptación:

$B$ obtiene el mensaje como $M = S – l_B R$

Esto funciona porque:

| | |
|---|---|
| $S – l_B R = S – l_B kP$ | (def. de $R$) |
| $S – l_B R = S – kl_B P$ | (conmutativa) |
| $S – l_B R = S – kQ_B$ | (def. de clave pública) |
| $S – l_B R = M + kQ_B – kQ_B$ | (def. de $M$) |
| $S – l_B R = M$ | (resta) |

En este caso se puede ver que el texto encriptado tiene el doble de largo que el texto plano pues se deben dar dos puntos de la curva para encriptar uno.

# Criptosistema de Menezes-Vanstone

Uno de los problemas del criptosistema anterior es que el espacio de mensajes consiste solamente en los puntos de la curva el cual es un subconjunto relativamente pequeño de $\mathrm{GF}_q \times \mathrm{GF}_q$. Por el contrario, el que se comentará a continuación [7.5], puede usar como espacio de mensajes el conjunto completo de $\mathrm{GF}_q \times \mathrm{GF}_q$ sin necesidad de transformar los mensajes en puntos de una curva.

El setup del método es exactamente igual que el del anterior por lo que no se repetirá.

Encriptación: Si $A$ desea mandar a $B$ un mensaje $M = (m_1, m_2)$ perteneciente a $\mathrm{GF}_q /\{0\} \times \mathrm{GF}_q /\{0\}$ debe hacer lo siguiente

1. $A$ genera un número aleatorio $k$ perteneciente a $\{2, 3, ..., n – 2\}$ y calcula $R = kP$

2. *A* calcula $(x_1, y_1) = kQ_B$
3. *A* computa
$$x_2 = m_1 x_1 \pmod q$$
$$y_2 = m_2 y_1 \pmod q$$
4. Envía $(R, (x_2, y_2))$

Desencriptación: *B* recibe lo que le manda *A* y calcula

1. $(x_1, y_1) = l_B R$
2. Entonces computa el mensaje original $(m_1, m_2)$ como:
$$m_1 = x_2\, x_1^{-1} \pmod q$$
$$m_2 = y_2\, y_1^{-1} \pmod q$$

Y funciona porque:

$m_1 = x_2\, x_1^{-1}$ $\quad$ $m_2 = y_2\, y_1^{-1}$
$m_1 = m_1 x_1 x_1^{-1}$ $\quad$ $m_2 = m_2 y_1 y_1^{-1}$
$m_1 = m_1$ $\quad$ $m_2 = m_2$

Y además el $x_1$ que calcula *B* es el mismo $x_1$ que usó *A* pues:

$l_B R = l_B k\, P = k\, l_B\, P = kQ_B$

Con el mismo argumento se prueba que el $y_1$ que calcula *B* es el mismo que tenía *A* y $m_2$ es el mensaje correcto.

El problema con este criptosistema es que los textos cifrados parciales dan información sobre el texto completo, en particular si alguien descubre $m_2$ puede calcular $m_1$ a partir del texto cifrado completo $(x_2, y_2)$ de la siguiente manera:

$$y_1 = y_2\, m_2^{-1} \pmod q$$

además $x_1$ cumple (por ser la componente en *x* de un punto de una curva elíptica):

$$y_1^2 = x_1^3 + ax_1 + b \pmod q$$

la cual tiene a lo sumo 3 soluciones que se pueden calcular y de esta forma obtener:

$$m_1 = x_2 x_1^{-1} \pmod q$$

Para evitar este problema se puede recurrir al método de usar solamente $m_1$ como mensaje con la contra de que ahora tendremos un factor de expansión del mensaje de 4 mientras que el del método anterior era 2. Este método de usar sólo $m_1$ como mensaje se conoce como *Elliptic Curve Encryption System*

# Disminución de la información intercambiada

Cuando se está trabajando en cuerpos con característica mayor que 3 y un emisor *S* tiene que enviar un punto de la curva definida por:

$$y^2 = x^3 + ax + b$$

la forma trivial de hacerlo es enviar los dos valores (o sea *x* e *y*) Sin embargo, si el receptor del mensaje conoce los coeficientes *a* y *b* hay un método para disminuir el ancho de banda necesario a costa de realizar un cálculo extra (ver [7.2]).

Suponiendo que *S* mande sólo la coordenada *x*, el receptor podría calcular $y^2$ a partir de la ecuación de la curva. Esto deja dos posibilidades para la otra coordenada que son *y* junto con *–y* (módulo *q*) De esta forma *S* sólo tendría que enviar un bit más para indicar cual de las dos opciones es la correcta en lugar de transmitir la coordenada *y* completa.

En el caso de las tarjetas inteligentes probablemente sea más conveniente no utilizar este método pues en general será más interesante reducir el tiempo requerido en cálculos que disminuir la información intercambiada con el lector de tarjetas.

# Seguridad / Ataques

La principal razón del atractivo de ECC es que no existe un algoritmo sub-exponencial conocido capaz de resolver el problema del logaritmo discreto en una curva elíptica apropiadamente elegida. El mejor algoritmo conocido para resolver el problema matemático subyacente en ECC (*ECDLP*: Elliptic Curve Discrete Logarithm Problem) toma un tiempo completamente exponencial.

A continuación presentamos algunos de los ataques conocidos al *ECDLP* según lo publicado en [7.6].

## Problema del logaritmo discreto de curvas elípticas (*ECDLP*)

Como ya se comentó anteriormente, el *ECDLP* es el siguiente: dada una curva elíptica *E* definida sobre un cuerpo finito $F_q$, un punto *P* perteneciente a $E(F_q)$ de orden *n*, y un punto $Q = lP$ donde $0 \leq l \leq n-1$, el problema es determinar *l*.

## Ataques conocidos al *ECDLP*

1) Búsqueda exhaustiva: En este método solo se computan múltiplos sucesivos de *P*: *2P*, *3P*, *4P*,.... hasta que se obtiene Q. Puede llevar hasta *n* pasos como peor caso.

2) Algoritmo de Pohlig-Hellman: Este algoritmo explota la factorización de *n*, el orden del punto *P*. El algoritmo reduce el problema de recuperar *l* al problema de recuperar *l* modulo cada uno de los factores primos de *n*. Luego *l* puede ser recuperado usando el Teorema Chino del Resto.

3) Algoritmo Paso-Bebé Paso-Gigante: Este algoritmo es una limitación tiempo-memoria de la búsqueda exhaustiva. Requiere almacenamiento para alrededor de $\sqrt{n}$ puntos y su tiempo de corrida esta en el entorno de $\sqrt{n}$ pasos en el peor caso.

4) Algoritmo de Pollard Rho: Este algoritmo es una versión aleatoria del algoritmo anterior. Su tiempo de corrida es muy parecido, $\sqrt{\frac{pn}{2}}$ pasos pero el algoritmo es superior al anterior ya que necesita una cantidad menor de almacenamiento. Se ha demostrado que este algoritmo se puede acelerar para que tenga una performance de $\frac{\sqrt{pn}}{2}$ pasos.

5) Algoritmo paralelizado de Pollard Rho: Se ha demostrado que cuando el algoritmo de Pollard Rho se corre en paralelo en *r* procesadores el tiempo esperado de corrida del algoritmo esta cerca de $\sqrt{\frac{pm}{2r}}$ pasos.

# Comparaciones

Aquí presentamos algunos datos comparativos entre tres criptosistemas [7.6]:

| | RSA1024 | ECC168 | NTRU263 |
|---|---|---|---|
| Expansión del mensaje | 1 – 1 | 2 – 1 | Aprox. 4.5 – 1 |
| Tamaño del bloque de texto plano (bits) | 1024 | 160 | 416 |
| Tamaño de la clave pública (bits) | 1024 | 169 | 1841 |

**Tabla 6: Comparación de ECC con otros criptosistemas**

# Apéndice II: GGH

## Definiciones

Previo a comenzar a explicar este algoritmo se debe introducir el concepto de "defecto ortogonal" (*orthogonality defect*) de una matriz *B* que se define como:

$$\text{orth} - \text{defect}(B) = \frac{\prod_i \|b_i\|}{|\det(B)|}$$

donde $\|b_i\|$ es la norma Euclidiana de la i-ésima columna de *B*.

Otra noción importante que debe ser introducida aquí es la del lattice dual. Si *B* es la base de un lattice *L*(*B*) (donde pensamos en *B* como en una matriz de $n \times n$ cuyas columnas generan el lattice) se dice que el lattice dual al original es aquel que es generado por las filas de la matriz $B^{-1}$

De esta forma el "defecto ortogonal dual" (*dual-orthogonality defect*) de una matriz *B* se define como el defecto ortogonal de la matriz $B^{-1}$ considerando las filas en lugar de las columnas. O sea que, aplicando propiedades básicas de las matrices se tiene:

$$\text{dual} - \text{orth} - \text{defect}(B) = \prod_i \|\hat{b}_i\| . |\det(B)|$$

donde las $\hat{b}_i$ son las filas de la matriz $B^{-1}$

Por más información a este respecto ver [3.1].

## Generación de claves

Para generar las claves se eligen dos matrices *B* y *R* donde *B* tiene un alto defecto ortogonal dual y *R* tiene un bajo defecto ortogonal dual. *B* será la clave pública y *R* la privada. Se elige además un número s real positivo que medirá el "tamaño" del vector de error.

Cuanto más grande es s más seguro es el sistema pues más alejado está el vector "encriptado" del original, pero si es muy grande cabe la posibilidad de que no se pueda desencriptar con la clave privada. La selección de este valor dependerá mucho de la aplicación del criptosistema dado que se debe decidir si la alta seguridad compensa la posibilidad de tener mensajes que no puedan ser desencriptados.

Se puede demostrar que para que la desencriptación no falle nunca se debe cumplir que $s < 1/(2r)$ donde *?* es la máxima norma $L_1$ de las filas de $R^{-1}$

Además se tiene que para que la probabilidad de error al desencriptar sea menor que e se debe cumplir que $s \leq (l\sqrt{8\ln(2n/e)})^{-1}$ donde $\frac{l}{\sqrt{n}}$ es la máxima norma $L_8$ de las filas $R^{-1}$ y se supone que *R* es una matriz de $n \times n$ (ver prueba en [3.1])

Para generar la matriz *R* se propone generar una matriz aleatoria *R'* con coeficientes uniformemente distribuidos en {+*l*, –*l*} (supuestamente el valor de *l* no afecta mucho la seguridad del sistema por lo que puede ser pequeño) y luego calcular *R* = *R'* + *k*.*l* Cuanto más grande sea *k* menor será el defecto ortogonal dual, por lo que en principio se podrán tomar valores mayores para el parámetro s. El pro-

blema es que un atacante siempre puede intentar reducir la base pública para disminuir su defecto ortogonal dual. Experimentalmente los autores [3.1] recomiendan $k = l.\sqrt{n}$

En cuanto a la matriz *B* se procede a mezclar la base *R* de forma de aumentar su defecto ortogonal dual. De forma que el algoritmo LLL (ver glosario) no pueda recuperar la base original. Para lograr esto se proponen dos formas:

1) Sumarle a cada vector de la base una combinación lineal de los restantes. En la práctica se tomaron coeficientes en {–1, 0, 1} para que los vectores no crecieran mucho (y por lo tanto el espacio necesario para almacenar la clave). Se deben hacer alrededor de 2n combinaciones.
2) Multiplicar *R* por varias matrices unimodulares[4] $T_i$ de forma que $B = R \,.\, T_1 \,.\, T_2 \,.\, .... \,.\, T_n$
   Las matrices $T_i$ son elegidas como el producto de dos matrices $L_i$ y $U_i$ las cuales son triangulares inferior y superior respectivamente y con entradas ±1 en la diagonal (de forma que su determinante sea ±1). En la práctica se eligieron las entradas distintas de cero de las matrices de forma que pertenecieran al conjunto {–1, 0, 1} y se vio que había que hacer al menos cuatro multiplicaciones para que LLL no pudiera recuperar la base original.

# Encriptado

Se elige un vector *v* perteneciente al lattice. Se plantea elegir un vector con entradas entre $-n$ y $+n$ con el número *n* elegido de forma que se obtengan enteros de aprox. 8 bits. No se sabe si el tamaño de estos números afecta la seguridad del sistema o no, pero se desea que quepan en un byte para disminuir el espacio necesario para almacenar las claves. Luego se determina un vector e con entradas +s o –s con probabilidad ½ cada una (también se puede usar $\pm\lceil s \rceil$ si no se quieren utilizar valores reales para s pero la probabilidad con que se deben elegir cambia). El texto cifrado c se halla como:

$$c = Bv + e \qquad (1)$$

Otro problema aparte que se debe considerar es el de cómo representar los bits a encriptar en los vectores *v* de forma que ver el texto cifrado no aporte ninguna información sobre el texto plano original. Sería un error intentar usar todos los bits de las entradas del vector *v* para contener información sobre el texto original, pues un atacante podría obtener una estimación de cada una de estas entradas usando la base pública y calculando $B^{-1}c$ que por (1) es igual a $v + B^{-1}e$. En este caso, si alguna de las filas de $B^{-1}$ tiene norma euclidiana pequeña el "error" que le agrega a *v* el vector $B^{-1}e$ puede ser pequeño (incluso podría llegar a tener alguna entrada menor que ½ en valor absoluto por lo que un atacante obtendría la entrada correcta de *v* por simple redondeo) En teoría las filas de $B^{-1}$ no deberían tener norma pequeña pues esta matriz tiene un alto defecto ortogonal, pero puede darse el caso de que alguna fila tenga norma pequeña y las otras norma muy grande por lo que el defecto ortogonal de toda la matriz siga siendo alto.

Por estas razones, los datos a encriptar sólo deben ser colocados en los bits de menor orden de las entradas de v y sólo en aquellas que no correspondan a columnas de $B^{-1}$ con norma euclidiana pequeña. Si hay muchas columnas en $B^{-1}$ con norma pequeña se podrían igual utilizar para representar un solo bit usando el XOR del bit menos significativo de las entradas de v correspondientes a cada una de ellas.

# Desencriptado

Se representa el texto cifrado *c* como una combinación lineal de las columnas de *R* y se redondean los coeficientes al entero más cercano para obtener un vector del lattice. La representación de este vector según las columnas de *B* es el vector *v*.

[4] Una matriz unimodular es aquella que tiene entradas enteras y determinante igual a ±1 Todas las bases de un lattice tienen el mismo determinante a menos del signo por lo que la multiplicación de la base por estas matrices no afecta el lattice generado por el resultado del producto.

La cuenta que hay que hacer es esta:

$$v = T \,.\, round(R^{-1}c) \qquad \text{donde} \quad T = B^{-1}R$$

Con $R^{-1}c$ se halla la combinación lineal de la base $R$ que determina c (pues si $R^{-1}c = x$, entonces $Rx = c$) Entonces se redondea y al multiplicar por $R$ se obtiene el vector del lattice generado por los coeficientes (redondeados) y al multiplicar por $B^{-1}$ se obtiene la representación del vector según la base $B$.

Los autores demuestran que para que el desencriptado tenga éxito se debe cumplir que $round(R^{-1}e) = 0$ (ver demostración en [3.1]) o sea que las entradas de $R^{-1}e$ deben estar entre +½ y –½

Para hacer más eficientes estas operaciones en la práctica, en lugar de guardar la base privada $R$, se pueden almacenar otras dos matrices que serían $R^{-1}$ y la matriz unimodular $T$, las cuales se utilizan explícitamente en el cálculo de v, el cual puede ser hecho en orden cuadrático. Usando este sistema se necesitan menos cálculos pero el doble de espacio de almacenamiento ($2.n^2$)

En el caso de la matriz $T$, esta tiene entradas enteras que con los parámetros usados en la práctica no serán muy grandes, pero la matriz $R^{-1}$ tendrá en general entradas reales por lo que entra en juego la representación interna que se utilizará para guardar estas entradas. En un sistema de bajos recursos no se podrá tener una representación exacta de estos números por lo que sólo se guardarán los primeros $l$ bits más significativos. De esta forma se tendrá un error de a lo sumo $2^{-l}$ en cada entrada de la matriz. Esta pérdida de exactitud no perjudica la seguridad del sistema (pues afecta la precisión de la clave privada) pero aumenta la probabilidad de error de desencriptación. El resultado es que ahora las entradas de la matriz $R^{-1}e$ deberán estar entre ($½ – n$) y ($–½ + n$) donde $n$ será mayor cuanto menor sea $l$ (prueba en [3.1])

Una propiedad adicional de este criptosistema es que, si bien hay una cierta probabilidad de que la desencriptación falle, también es posible detectar la ocurrencia de un error con muy alta probabilidad. La idea es que, si uno utiliza un vector de error $e$ con entradas ±s y encripta un vector $p$ como $c = p + e$, entonces es trivial comprobar si un vector $p'$ que se desencriptó corresponde con el p original mediante la obtención el vector $e' = c – p'$ y comprobando que todas sus entradas sean +s o –s.

Por otro lado esta propiedad también permite que, a pesar de ser un criptosistema aleatorio, un atacante que conozca la clave pública ($B$, s) pueda testear si un vector encriptado $c$ corresponde a un vector original $m$ calculando $c – Bm$ y comprobando si las entradas de este vector están en {+s, –s}

# Historia

Inspirado en un resultado de Ajtai [3.4] quien presentó una función de una sola vía (es decir, difícil de invertir) que fue el primer trabajo donde se introdujo la idea de usar el problema SVP (*Shortest non-zero Vector Problem*) como problema difícil en criptografía. Ajtai no presentó una forma fácil de invertir la función (o sea lo que sería una clave privada) por lo que dicho método no puede usarse para plantear algoritmos de clave pública. Sin embargo se puede probar que invertir la función es tan difícil como resolver el peor caso del SVP.

Es parecido al criptosistema de McEliece que usa códigos lineales aunque no está basado en él. En ambos se utiliza el método de multiplicar una matriz por un vector y agregar al resultado un ruido aleatorio, aunque el dominio y el álgebra de las operaciones son diferentes. Por más información sobre esto último ver [3.3]

# Seguridad

## Selección de parámetros

Se recomienda en [3.1] elegir un lattice de dimensión 250-300 para evitar los métodos de reducción más efectivos conocidos hasta el momento, los cuales funcionan hasta dimensiones de alrededor de 100. Si se desea un menor nivel de seguridad pero claves bastante menores se puede usar un lattice de dimensión 120. Con respecto al parámetro s, y considerando el caso de n=120 y una probabilidad de error de $10^{-5}$ resulta que debe ser menor o igual a 2,5.

Para s se toma en general un valor de 3 para que la probabilidad de desencriptación correcta se mantenga relativamente alta y a la vez obtener un buen nivel de seguridad.

Se describirán a continuación algunos ataques posibles contra este criptosistema, algunos de los cuales son bastante generales para todos los algoritmos basados en lattices y otros (incluido el que permitió quebrar el sistema) son particulares para este caso.

## Ataque "Round-off"

Consiste simplemente en usar la base pública $B$ en lugar de la base privada en el algoritmo de desencriptación. Dado que para encriptar se calcula $c = Bv + e$ la idea es calcular:

$$B^{-1}c = v + B^{-1}e$$

y hacer una búsqueda exhaustiva por el vector $d = B^{-1}e$. Para mejorar el ataque la idea es que, en vez de usar la base $B$ directamente, se va a utilizar el algoritmo LLL para obtener una base con menor defecto ortogonal dual (problema SBP – *Smallest Basis Problem*) y esa será la que se usará en el ataque.

En la práctica, con dimensiones de hasta 80 el algoritmo LLL puede encontrar bases suficientemente buenas para descubrir el vector d esencialmente en O(1) (o sea que se descubre el vector directamente haciendo la operación mencionada arriba) Para dimensiones mayores la base que devuelve el algoritmo es cada vez menos útil y por dimensiones de alrededor de 100 el uso de este sistema es más ineficiente que la búsqueda trivial por fuerza bruta del vector de error e (recordar que esta búsqueda está acotada a valores entre +s y –s)

## Ataque por encajamiento

Este ataque es heurístico y se basa en "encajar" los $n$ vectores de la base pública $B$ y el punto $c$ en un nuevo lattice de dimensión $n$+1 cuya base es de la forma:

$$\begin{pmatrix} \vdots & \vdots & \vdots & & \vdots \\ c & b_1 & b_2 & \cdots & b_n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 0 & 0 & & 0 \end{pmatrix}$$

La idea es que, como ambos lattices tienen el mismo determinante y casi la misma dimensión, se espera que los vectores más cortos de ambos tengan más o menos el mismo largo. Entonces, si $v$ es el vector más cercano a $c$ en el lattice original, se puede ver que $(c - v, 1)$ pertenece al lattice ampliado (pues se genera con el opuesto de los coeficientes que generan $v$ en el lattice original y 1 multiplicando a la columna que contiene a $c$) y debería ser corto si $v$ está cerca de $c$. Si este vector en particular resulta ser el más corto, se podrá hallar y entonces a partir de él se tendrá el valor de $v$.

A diferencia de los ataques anteriores, este sistema no permite acotar una búsqueda, sino que sólo se puede ver si funciona o no. Los autores del criptosistema afirman que el ataque funciona y es factible de ser aplicado hasta dimensiones de 110 o 120.

Usando esta técnica Shnorr logró desencriptar textos cifrados de hasta dimensión 150 [3.6], aunque los autores del criptosistema indican que no es útil para dimensiones mayores de 120. Mientras que Nguyen logró descifrar un texto encriptado con un lattice de dimensión 200 [3.2] en unos pocos días, pero no se han logrado avances en dimensiones mayores.

# Ataque de Nguyen

Este ataque, descubierto por Phong Nguyen [3.2], aprovecha la forma en que se encripta en este criptosistema (mediante la suma de un vector de error) para obtener información parcial sobre los vectores originales a partir de los vectores con el vector de error adicionado. Usando este método se lograron desencriptar textos completos en dimensiones hasta 350, mientras que se desencriptaron en parte textos encriptados en dimensión 400.

Esto es suficiente para transformar el criptosistema en no práctico pues las claves del mismo deberían tener más de $8.400^2$ bits (aproximadamente 1 Mb) usando 8 bits por entrada en la matriz.

La idea del ataque es aprovechar la estructura de la ecuación que define el proceso de encriptado ($c = Bm + e$) y tratar de reducirla módulo algún número $n$ adecuado para de esta manera hacer desaparecer el vector $e$ (hacerlo igual a 0 mod $n$) y de esta manera obtener alguna información sobre $m$ (no se obtendrá $m$ pero se intentará obtener $m$ mod $n$)

Una idea es tomar la ecuación módulo s de modo que la ecuación quede:

$$c = Bm \pmod{s}$$

pues las entradas de $e$ son +s o –s. La idea que se usa definitivamente en este ataque es tomar la ecuación módulo 2s y definir un vector s de la forma (s, s, ... , s) para lograr que:

$$e + s = 0 \pmod{2s}$$

(pues sus entradas pueden ser 2s o 0). Entonces la ecuación que se usa para el ataque es

$$c + s = Bm \pmod{2s}$$

y la idea es obtener $m$ (mod 2s) lo cual da cierta información sobre el vector original y permite quebrar el sistema resolviendo un problema más sencillo que el original.

Este sistema es efectivo porque se puede probar que en la práctica esta ecuación sólo tiene pocas soluciones diferentes para $m$ (y con probabilidad no despreciable tiene una sola)

En el caso general de una ecuación $y = Bx \pmod{n}$ dos soluciones distintas difieren por un elemento del kernel de $B$. Esto es así pues un elemento $x$ pertenece al kernel de $B$ sii $Bx = 0 \pmod{n}$, entonces si $z$ es solución de la ecuación ($y = Bz \pmod{n}$) entonces $x + z$ también lo es pues:

$$B(x + z) = Bx + Bz = 0 + y = y \pmod{n}$$

Entonces todas las soluciones de la ecuación pueden ser encontradas a partir del kernel de $B$ y su número es igual al cardinal del mencionado kernel de $B$. Si la matriz $B$ en invertible módulo $n$ entonces la solución es única y se puede hallar como:

$$x = yB^{-1} \pmod{n}$$

Nguyen demostró que hay una considerable probabilidad de que la matriz sea invertible (es decir que la solución sea única) Las probabilidades por él calculadas en [3.2] para varios módulos son las siguientes:

| Módulo 2s | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| % | 28.9 | 56.0 | 28.9 | 76.0 | 16.2 | 83.7 | 28.9 | 56.0 | 22.0 |

**Tabla 7: Probabilidad de que *B* sea invertible**

Por lo que para el caso de s = 3 (2s = 6) que se recomienda en el paper de GGH la probabilidad es de 0.162.

De igual manera se intentó ver para cada módulo s cuales eran las probabilidades de que el kernel de *B* tuviera una cierta dimensión, y a partir de un teorema también demostrado por Nguyen se construyó la siguiente tabla:

| Modulo s | 2 | | | 3 | | | 5 | | |
|---|---|---|---|---|---|---|---|---|---|
| Dimensión del kernel | 1 | 2 | =3 | 1 | 2 | =3 | 1 | 2 | =3 |
| % | 57.8 | 12.8 | 0.5 | 42.0 | 2.0 | <0.1 | 23.8 | 0.2 | <<0.1 |

**Tabla 8: Probabilidades de diferentes dimensiones del kernel de *B***

Se sostiene que estos resultados son más o menos constantes para matrices con las dimensiones de interés en este estudio.

Para el caso particular de 2s = 6 los resultados son los siguientes:

| Cardinal del kernel | 1 | 2 | 3 | 4 | 6 | 9 | 12 | 18 | 36 | Otros casos |
|---|---|---|---|---|---|---|---|---|---|---|
| % | 16.2 | 32.4 | 12.1 | 7.2 | 24.3 | 0.6 | 5.4 | 1.1 | 0.3 | 0.6 |

**Tabla 9: Probabilidades para 2s = 6**

Donde se muestra que es muy improbable que el núcleo tenga cardinal mayor a 12 mientras que los casos más probables son 2, 6, 1 y 3. Se puede ver entonces que las soluciones posibles del sistema son en general pocas.

Suponiendo que se sabe $m$ (mod 2s) ($m_{2s}$ de aquí en adelante) esta información se pude utilizar para simplificar grandemente el proceso de encontrar el texto plano a partir de la base pública. La idea es que, partiendo de la ecuación original:

$$c = Bm + e$$

se puede restar $B.\, m_{2s}$ de los dos lados para obtener:

$$c - B.\, m_{2s} = B(m - m_{2s}) + e$$

y dado que las entradas del vector $m - m_{2s}$ son todas de la forma $m'.2s$ (pues se le están restando sus restos módulo 2s) se puede dividir por 2s de ambos lados para así obtener:

$$\frac{c - B m_{2s}}{2s} = B m' + \frac{e}{2s}$$

por lo que se vuelve a tener un problema de hallar el vector más cercano a $\frac{c - m_{2s} B}{2s}$ donde en este caso el vector de error $\frac{e}{2s}$ tiene entradas que están en $\{+½, -½\}$ por lo que es claramente menor que el vector e original y el problema CVP es más sencillo de resolver.

Una vez resuelto este problema simplificado, es trivial resolver el problema original dado que alcanza con multiplicar el vector de error por 2s y así obtener el vector de error original y restarlo al vector encriptado *c* para obtener *Bm*:

$$c - e - B.\, m_{2s} = B.m - B.m_{2s}$$
$$c - e = B.m$$

y al multiplicar por $B^{-1}$ se obtiene el mensaje original ($B^{-1}$ siempre existe pues *B* es una base y sus vectores son LI)

Ya se comentó que en general se tendrán varias soluciones posibles para $m_{2s}$ por lo que se deberán resolver varias instancias de este problema CVP más sencillo que el original. Como hay una instancia independiente del problema para cada uno de los $m_{2s}$, los cálculos necesarios se pueden paralelizar realizándose en diferentes máquinas hasta que una de ellas encuentre el texto original (ya se comentó que es muy fácil chequear si un texto plano dado es una desencriptación correcta o no de un texto cifrado).

Es claro que este ataque fallará en el caso de que las entradas del vector de error no sean exactamente +s o –s sino que sean números enteros ubicados entre estos dos valores. En este caso estos vectores resultarán ser menores a los que se usaban con el primer caso haciendo el criptosistema más débil a los otros tipos de ataques ya mencionados (como el de encajamiento) El problema es que incluso con estos métodos el problema puede ser resuelto hasta con lattices de dimensiones alrededor de 200 y si bien puede ser seguro en dimensiones mayores ya se comentó que el tamaño de las claves lo hacen no solamente inaplicable a ambientes de bajos recursos, sino que no lo hacen competitivo con otros criptosistemas muy conocidos.

# Apéndice III: NTRU

## Notación

El criptosistema NTRU depende de 3 parámetros enteros ($N$, $p$, $q$) y 4 conjuntos $L_f$, $L_g$, $L_f$, $L_m$ de polinomios de grado $N$–1 con coeficientes enteros. No es necesario que $p$ y $q$ sean primos, pero asumiremos que gcd($p$,$q$) = 1 y que $q$ siempre será bastante mayor que $p$. Es importante que gcd($p$,$q$) = 1 ya que aunque en principio NTRU funciona sin esta restricción, en la práctica el tener que el gcd($p$,$q$) > 1 reduce la seguridad del sistema. Esto se explicará más adelante.

Trabajaremos en el anillo $R = Z[x]/(x^N - 1)$ de polinomios con coeficientes enteros reducidos módulo un polinomio de grado $N$ (por una excelente referencia matemática sobre esto ver [1.5]) Un elemento $F \in R$ será escrito como un polinomio o un vector,

$$F = \sum_{i=0}^{N-1} F_i x^i = [F_0, F_1, \ldots, F_{N-1}]$$

Usamos $\otimes$ para denotar la multiplicación en $R$. Esta multiplicación esta dada explícitamente como un producto de polinomios en el anillo R

$$F \otimes G = H \quad \text{con} \qquad H_k = \sum_{i=0}^{k} F_i G_{k-i} + \sum_{i=k+1}^{N-1} F_i G_{N+k-i} = \sum_{i+j \equiv k \pmod N} F_i G_j$$

Esta operación es la multiplicación habitual de polinomios cuyos resultados son reducidos módulo el polinomio ($x^N$–1). De hecho la implementación real de este producto (dadas las características particulares que tiene) puede ser realizada mediante un producto normal de polinomios para luego reemplazar la potencia de $x^N$ por 1, la potencia de $x^{N+1}$ por $x$ y así sucesivamente con todas las potencias de $x$ que tengan el exponente mayor que $N$–1.

Ejemplo (tomado de [4.15]):

> Supongamos $N$ = 3, y tomemos los polinomios $F = 3x^2 - x + 2$ y $G = -x^2 + 2x + 1$ reducidos módulo $x^3 - 1$
>
> Entonces,
>
> $F \otimes G = (3x^2 - x + 2)(-x^2 + 2x + 1)$
>
> $F \otimes G = -3x^4 + 7x^3 - x^2 + 3x + 2$ (reemplazando)
>
> $F \otimes G = -3x + 7 - x^2 + 3x + 2$
>
> $F \otimes G = -x^2 + 9$
>
> Que es lo mismo que se obtiene como resto al dividir $-3x^4 + 7x^3 - x^2 + 3x + 2$ entre $x^3 - 1$ (el cociente en este caso es $-3x + 7$ )

Cuando realizamos una multiplicación módulo (por ejemplo) $q$, queremos decir reducir los coeficientes módulo $q$ además de la reducción de los polinomios en sí. Esto es así pues los coeficientes de los polinomios pertenecen a los enteros módulo $q$.

*A tener en cuenta:* En principio, el cálculo del producto $F \otimes G$ tomaría $N^2$ multiplicaciones. Sin embargo, para el típico producto usado por NTRU, $G$ o $F$ tienen coeficientes pequeños, por lo que su multiplicación es muy rápida. Por otro lado, si N es demasiado grande, entonces sería más rápido usar

el método *Fast Fourier Transforms* (ver glosario) para calcular el producto en el orden de ($N \log N$) operaciones. La información pasada así como la que viene a continuación puede ser profundizada en [4.1].

# Creación de claves

Para crear una clave NTRU, la persona **A** escoge 2 polinomios $f$ y $g$ tales que $f \in L_f$ y $g \in L_g$. El polinomio $f$ debe satisfacer el requerimiento adicional de tener inversos módulo $q$ y módulo $p$. Si se eligen bien los parámetros, esto se cumplirá para la mayoría de las elecciones de $f$, y el cálculo de estos inversos se realiza fácilmente usando una modificación del algoritmo de Euclides (ver 4.3.1.3). Denotaremos estos inversos con $F_q$ y $F_p$, esto significa:

$$F_q \otimes f = 1 \pmod q \qquad \text{y} \qquad F_p \otimes f = 1 \pmod p$$

Entonces la persona **A** calcula lo siguiente:

$$h = F_q \otimes g \pmod q$$

La clave pública de **A** es el polinomio $h$. La clave privada de **A** es el polinomio $f$, aunque en la práctica **A** también almacenará $F_p$ ya que de esta forma se estará evitando volver a calcularlo en el momento de la desencriptación.

# Encriptación

Supongamos que la persona **B** (el encriptador) quiere mandarle un mensaje a la persona **A** (el desencriptador). **B** comienza eligiendo un mensaje $m$ del conjunto de textos planos $L_m$. Seguidamente escoge un polinomio $f \in L_f$ y utiliza la clave pública de **A** ($h$) para calcular:

$$e = p f \otimes h + m \pmod q$$

Este será el mensaje encriptado que **B** le enviará a **A**.

Aquí se puede ver uno de los casos extremos de por qué se pide que gcd($p$,$q$) = 1. Si se diera que $p|q$[5], entonces el mensaje encriptado $e$ satisface que $e=m \pmod p$ por lo que la encriptación sería completamente insegura.

# Desencriptación

Supongamos que **A** recibió el mensaje $e$ de **B** y quiere desencriptarlo usando su clave privada $f$. Para realizar esto eficientemente, **A** debería haber calculado el polinomio $F_p$.
Para desencriptar $e$, la persona **A** calcula,

$$a = f \otimes e \pmod q$$

donde escoge los coeficientes de $a$ en el intervalo que va desde $-q/2$ hasta $q/2$. Ahora tratando a $a$ como un polinomio con coeficientes enteros. **A** recupera el mensaje calculando,

$$F_p \otimes a \pmod p$$

[5] Se dice que $p|q$ ($p$ divide a $q$) si existe un número entero $n$ tal que $q = pn$

# Por que funciona la desencriptación

El polinomio $a$ que **A** calcula satisface:

$$\begin{aligned} a &= f \otimes e \pmod q \\ &= f \otimes (p f \otimes h + m) \pmod q && \text{e} = \text{p } f \otimes \text{ h} + \text{m} \pmod q \\ &= (f \otimes p f \otimes h) + (f \otimes m) \pmod q && \text{distributiva} \\ &= (f \otimes p f \otimes F_q \otimes g) + (f \otimes m) \pmod q && h = F_q \otimes g \pmod q \\ &= (p f \otimes g \otimes f \otimes F_q) + (f \otimes m) \pmod q && \text{conmutativas} \\ &= (p f \otimes g) + (f \otimes m) \pmod q && f \otimes F_q = 1 \pmod q \end{aligned}$$

Consideremos este último polinomio $p f \otimes g + f \otimes m$. Si se eligen apropiadamente los parámetros, se puede asegurar que (casi siempre) todos sus coeficientes caen entre $-q/2$ y $q/2$, por lo tanto no cambia si sus coeficientes son reducidos módulo $q$. Esto significa que cuando **A** reduce los coeficientes de $f \otimes e$ módulo $q$ en el intervalo que va de $-q/2$ hasta $q/2$, recupera exactamente el polinomio:

$$a = p f \otimes g + f \otimes m \qquad \text{en } \mathrm{Z[x]}/(\mathrm{x}^N - 1)$$

Reducir $a$ módulo $p$ le da a **A** el polinomio $f \otimes m \pmod p$ (pues los coeficientes del primer sumando son múltiplos de $p$ y por lo tanto se reduce a 0), y la multiplicación por $F_p$ recupera el mensaje $m \pmod p$

*Importante:* Para valores apropiados de los parámetros, existe una probabilidad extremadamente alta de que el procedimiento de desencriptación recupere el mensaje original. Sin embargo, algunas elecciones de parámetros pueden causar fallas en la desencriptación, por lo que se deberían incluir unos pocos bits de chequeo en cada bloque del mensaje. La causa más usual de falla en la desencriptación es que el mensaje no esta apropiadamente centrado. Es decir que sus coeficientes no se encuentran exactamente entre $-q/2$ y $q/2$. En este caso la persona **A** podría ser capaz de recobrar el mensaje escogiendo los coeficientes de $a = f \otimes e \pmod q$ en un intervalo apenas distinto, por ejemplo desde $-q/2+x$ hasta $q/2+x$ para algún valor (positivo o negativo) de $x$. Si ningún valor de $x$ funciona, entonces se dice que tenemos un *error de hueco* (gap failure) y que el mensaje no puede ser desencriptado tan fácilmente. Para parámetros con valores bien escogidos, este tipo de falla ocurrirá tan difícilmente que en la práctica puede ser ignorado.

# Espacios de muestras

Los conjuntos $L_i$ presentados anteriormente se conocen como los espacios de muestras y son definidos en forma detallada a continuación:

El espacio de mensajes $L_m$ consiste en todos los polinomios módulo $p$. Asumiendo que $p$ es impar, es más conveniente tomar,

$$L_m = \left\{ m \in R : m \text{ tiene coeficientes entre } \tfrac{-(p-1)}{2} \text{ y } \tfrac{p-1}{2} \right\}$$

Para describir los otros espacios de muestras, vamos a usar conjuntos de la forma

$$L(d_1, d_2) = \left\{ F \in R : F \text{ tiene } d_1 \text{ coeficientes iguales a } 1,\ d_2 \text{ coeficientes iguales a } -1 \text{ y el resto igual a } 0 \right\}$$

Con esta notación, escogemos 3 enteros positivos $d_f$, $d_g$, $d_f$, y seteamos

$$L_f = L(d_f, d_f - 1), \quad L_g = L(d_g, d_g), \quad \text{y} \quad L_f = L(d_f, d_f)$$

$L_f$ es el espacio en que se elige el polinomio $f$, $L_g$ es el espacio en que se elige el polinomio $g$ y $L_f$ es el conjunto de donde se selecciona el polinomio aleatorio $f$.

La razón por la cual no seteamos $L_f = L(d_f, d_f)$ es porque queremos que $f$ sea invertible, y un polinomio que satisface $f(1) = 0$ nunca puede ser invertible.

# Elección apropiada de los parámetros

Se habló varias veces de que los parámetros del criptosistema deben ser elegidos en forma apropiada para que la desencriptación funcione con alta probabilidad (aún considerando mensajes no apropiadamente centrados pero que igual pueden ser desencriptados con pequeñas correcciones).

Expresada matemáticamente esta propiedad puede escribirse como:

$$\left| pf \otimes g + f \otimes m \right|_\infty < q$$

Donde la norma infinito se define en este caso como:

$$\left| F \right|_\infty = \max_{1 \le i \le N} \{F_i\} - \min_{1 \le i \le N} \{F_i\}$$

Los autores sostienen empíricamente [4.1] que la propiedad anterior es válida en general si los parámetros cumplen:

$$\left| f \otimes m \right|_\infty \le q/4 \qquad \text{y} \qquad \left| pf \otimes g \right|_\infty \le q/4 \qquad (1)$$

Y a partir de evidencia experimental proponen usar valores de $N$=107, $N$=167, y $N$=503 dependiendo del nivel de seguridad deseado (siendo $N$ el grado del polinomio por el que se dividen los elementos del anillo)

# Seguridad

## Ataques de Fuerza Bruta

Un atacante puede recuperar la clave privada probando todas las posibles $f \in L_f$ y comprobando si $f \otimes h \pmod q$ tiene entradas pequeñas. El fundamento de ese ataque consiste en que como $h$ cumple que:

$$h = F_q \otimes g \pmod q$$

al multiplicar $f$ por $h$ se obtiene el polinomio $g \pmod q$ el cual debe tener entradas solamente en –1, 1 o 0. De igual manera se pueden probar todas las $g \in L_g$ y comprobar si $g \otimes h^{-1} \pmod q$ tiene entradas pequeñas. Esa operación debe dar como resultado el polinomio $F_q^{-1}$ que es igual a $f$ por lo que también tendrá valores que sólo podrán ser 0, 1 y –1 (este segundo método es un poquito más costoso pues hay que invertir $h$, pero esto sólo se hace una vez)

De forma similar un atacante puede recuperar un mensaje probando todas las posibles $f \in L_f$ y comprobando si:

$$e - pf \otimes h \pmod q$$

tiene entradas pequeñas. En este caso el polinomio que se obtendrá será el propio mensaje $m$ por lo que sus entradas deberán encontrarse entre $-(p-1)/2$ y $(p-1)/2$

En la práctica $L_g$ será más pequeño que $L_f$ (pues $d_f$ se elige siempre mayor que $d_g$) por lo que la seguridad de la clave esta determinada por $\#L_g$ y la seguridad de cada mensaje esta determinada por $\#L_f$.

Sin embargo, existe un ataque "Encuentro en el Medio" (*meet-in-the-middle*) el cual corta el tiempo de búsqueda por la raíz cuadrada. La idea de este ataque es que uno divide $f$ en dos, digamos que $f = f_1 + f_2$, y entonces compara $f_1 \otimes e$ contra $-f_2 \otimes e$, buscando $(f_1, f_2)$ de tal forma que los coeficientes correspondientes tengan aproximadamente el mismo valor (ver [4.6])

Por esta razón, a partir de cálculos de conteo, se puede obtener que el nivel de seguridad esta dado por:

$$\begin{pmatrix}\text{Seguridad}\\ \text{de la clave}\end{pmatrix} = \sqrt{\#L_g} = \frac{1}{d_g!}\sqrt{\frac{N!}{(N-2d_g)!}}$$

$$\begin{pmatrix}\text{Seguridad}\\ \text{del mensaje}\end{pmatrix} = \sqrt{\#L_\varnothing} = \frac{1}{d_\varnothing!}\sqrt{\frac{N!}{(N-2d_\varnothing)!}}$$

Por lo tanto, para obtener un nivel de seguridad de (digamos) $2^{80}$ uno debería escoger $f, g, f$ de conjuntos que contengan alrededor de $2^{160}$ elementos.

# Ataques de Multi-Transmisión

Si **B** envía un solo mensaje $m$ varias veces usando la misma clave pública pero diferentes $f$'s, entonces el atacante **C** será capaz de recuperar una gran parte del mensaje.

Supongamos que **B** transmite $e = p f_i \otimes h + m \pmod q$ con $i = 1, 2, ..., r$. Entonces **C** puede calcular $(e_i - e_1) \otimes h^{-1} \pmod q$, con lo que recupera $p f_i - p f_1 \pmod q$. Sin embargo $p$ y los coeficientes de los $f$'s son en general tan pequeños que **C** recupera exactamente $f_i - f_1$, y desde aquí **C** podrá estudiar los valores que obtiene de forma de intentar recuperar varios de los coeficientes de $f_1$. Aún si $r$ es pequeño (4 o 5), **C** podría recuperar suficientes coeficientes de $f_1$ como para probar todas las posibilidades de los coeficientes restantes por medio de fuerza bruta, con lo que lograría recuperar $m$.

Hay que acotar que aunque **C** desencriptara un mensaje de esta forma, la información no le servirá para desencriptar mensajes subsecuentes que usen otro valor de $f$.

# Ataques basados en el lattice

Existen 2 tipos de ataque al lattice, ataque a la clave privada y ataque al mensaje $m$. Estos ataques están basados en la reducción del lattice que consiste en encontrar uno o varios vectores pequeños en un lattice dado. En teoría, el vector más pequeño puede ser encontrado por medio de una búsqueda exhaustiva, pero en la práctica esto no es posible si la dimensión del lattice es grande.

El algoritmo LLL de Lenstra–Lenstra–Lovász, con varias mejoras introducidas por Schnorr y otros, es capaz de encontrar vectores relativamente pequeños en tiempos polinomiales, pero aún al LLL le tomará mucho tiempo encontrar el vector más pequeño si se da el caso de que el mismo no es mucho más pequeño que su largo esperado más probable.

*Ataques a la clave privada:*
Se considera la matriz de $2N \times 2N$ compuesta por 4 bloques de $N \times N$,

$$\left(\begin{array}{cccc|cccc} a & 0 & \cdots & 0 & h_0 & h_1 & \cdots & h_{N-1} \\ 0 & a & \cdots & 0 & h_{N-1} & h_0 & \cdots & h_{N-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a & h_1 & h_2 & \cdots & h_0 \\ \hline 0 & 0 & \cdots & 0 & q & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & q & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & q \end{array}\right)$$

donde $a$ es un parámetro que será definido a continuación. Sea $L$ el lattice generado por las filas de la matriz; su determinante es $q^N a^N$ (pues es una matriz triangular).

Como la clave pública es $h = f^{-1} \otimes g$, el lattice $L$ contendrá el vector $T=(a\, f, g)$, con el cual queremos decir el vector de largo $2N$ formado por los $N$ coeficientes de $f$ multiplicados por $a$, seguido de los N coeficientes de $g$. A partir de una técnica heurística, se propone en [4.1] que el largo esperado del vector más corto de un lattice aleatorio de dimensión $N$ será más grande (pero no mucho más) que:

$$s = \sqrt{\frac{Naq}{pe}}$$

Una implementación del algoritmo de reducción de lattices tendrá la mejor chance de localizar $T$, o algún otro vector cuyo largo este cerca de $T$, si el atacante elige $a$ de tal forma de maximizar la relación $s/|T|_2$. O sea que lo que se busca es lograr que el vector T sea bastante más pequeño que el largo esperado del vector más pequeño facilitando así su obtención mediante el uso del algoritmo de LLL. En esta relación la norma 2 de un vector se define como:

$$|F|_2 = \left( \sum_{i=1}^{N} (F_i - \overline{F})^2 \right)^{\frac{1}{2}}$$

donde $\overline{F}$ es el promedio de las $F_i$. Dicho de otra forma $|F|_2 / \sqrt{N}$ es la desviación estándar de los coeficientes de $F$.

Como se intenta minimizar la relación, los coeficientes constantes de $s$ no se considerarán en los pasos siguientes. De hecho el único coeficiente variable de $s$ cuando se está atacando un criptosistema ya definido es $a$.

Entonces, elevando al cuadrado la relación $s/|T|_2$, vemos que el atacante elige $a$ para maximizar

$$\left( \frac{\sqrt{a}}{|T|_2} \right)^2 = \frac{a}{|(af, g)|_2^2} =$$

$$\frac{a}{a^2 |f|_2^2 + |g|_2^2} = \left( a |f|_2^2 + a^{-1} |g|_2^2 \right)^{-1}$$

Entonces el mínimo se logra eligiendo $a = |g|_2 / |f|_2$. (Nótese que $|g|_2$ y $|f|_2$ son cantidades públicas aunque los polinomios específicos no lo sean, pues dependen solamente de $d_f$, $d_g$ y N los cuales son todos valores públicos)

Se puede ver que el valor de $a$ es el que se mencionó calculando la derivada de la expresión obtenida anteriormente y viendo que la misma equivale a:

$$\frac{|g|_2^2 - |f|_2^2 a^2}{\left(a^2 |f|_2^2 + |g|_2^2\right)^2}$$

por lo que $|g|_2/|f|_2$ es una de las raíces de la misma y es donde se encuentra el máximo (en $-|g|_2/|f|_2$ se encuentra su mínimo)

Cuando se elige $a$ de esta manera, se define una constante $c_h$ que equivale a $|T|_2 / s$.

Por lo tanto $c_h$ es la relación del largo del vector buscado con el largo del vector más corto esperado. Cuanto más chico sea $c_h$, más fácil será encontrar el vector buscado. Realizando algunas sustituciones y sabiendo que:

$$|T|_2 = 2a|f|_2|g|_2$$

llegamos a que:

$$c_h = \sqrt{\frac{2pe|f|_2|g|_2}{Nq}}$$

Para un par $(f, g)$ dado usado para inicializar el criptosistema, $c_h$ puede ser visto como una medida de cuan separado está el lattice asociado de un lattice aleatorio. Si $c_h$ esta cerca de 1, entonces $L$ se asemejará a un lattice aleatorio y a los métodos de reducción de lattices se les hará muy difícil encontrar un vector pequeño en general, y un $T$ en particular. A medida que $c_h$ disminuye, a los algoritmos de reducción de lattices se les simplificará el trabajo para encontrar $T$. Basados en la evidencia obtenida, los autores comentan que el tiempo requerido aparenta ser (por lo menos) exponencial en $N$, con una constante en el exponente proporcional a $c_h$.

*Ataques a un mensaje:*

Un ataque de lattice también puede ser dirigido contra un mensaje en particular $m$. Aquí el problema de lattice asociado es muy similar al del ataque a la clave privada, y el vector buscado será de la forma $(a\,m, f)$. Como anteriormente, el atacante tiene que definir el lattice usando $a = |f|_2 / |m|_2$ para minimizar el tamaño del vector que se va a buscar. Esto nos lleva al siguiente valor:

$$c_m = \sqrt{\frac{2pe|m|_2|\varnothing|_2}{Nq}}$$

Esta constante $c_m$ nos brinda una medida de la vulnerabilidad de un mensaje individual contra un ataque de lattice, similar a la forma en que $c_h$ lo hacía para un ataque de lattice sobre $h$. Un mensaje encriptado es más vulnerable si $c_m$ es pequeño, y se vuelve menos vulnerable a medida que se acerca a 1. Para hacer que los ataques sobre $h$ y $m$ sean igual de difíciles queremos tomar $c_m \approx c_h$, o equivalentemente:

$$|g|_2 |f|_2 \approx |f|_2 |m|_2.$$

Para fijar ideas vamos a tomar el caso en que $p$=3. Para $p$=3, un mensaje $m$ promedio consistirá de $N$/3 coeficientes de 1's, 0`s y –1's, por lo tanto $|m|_2 = \sqrt{2N/3}$

De forma similar, $f$ consistirá de $d$ coeficientes 1's, –1's y el resto 0's por lo que $|f|_2 = \sqrt{2d}$ Por lo tanto querremos setear:

$$|f|_2|g|_2 \approx \sqrt{2N/3}.\sqrt{2d} \approx \sqrt{\frac{4Nd}{3}}$$

Esta relación, junto con las expresadas en (1), se puede utilizar para realizar una elección apropiada de los parámetros.

# Valores reales de los parámetros

Basados en experimentación en computadoras los autores [4.1] recomiendan tres distintos grupos de parámetros que proporcionan diferente nivel de seguridad pero que siempre respetan la restricción de que la probabilidad de error al desencriptar es menor que $5\times10^{-5}$.

Seguridad moderada:

| N | $p$ | $q$ | $d_f$ | $d_g$ | $d_f$ | Largo de la clave privada | Largo de la clave pública | Seguridad de la clave | Seguridad de un mensaje | $c_h$ | $c_m$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 107 | 3 | 64 | 15 | 12 | 5 | 340 bits | 642 bits | $2^{50}$ | $2^{26.5}$ | 0.257 | 0.258 |

**Tabla 10: Seguridad moderada**

Seguridad alta:

| N | $p$ | $q$ | $d_f$ | $d_g$ | $d_f$ | Largo de la clave privada | Largo de la clave pública | Seguridad de la clave | Seguridad de un mensaje | $c_h$ | $c_m$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 167 | 3 | 128 | 61 | 20 | 18 | 530 bits | 1169 bits | $2^{82.9}$ | $2^{77.5}$ | 0.236 | 0.225 |

**Tabla 11: Seguridad alta**

Seguridad muy alta:

| N | $p$ | $q$ | $d_f$ | $d_g$ | $d_f$ | Largo de la clave privada | Largo de la clave pública | Seguridad de la clave | Seguridad de un mensaje | $c_h$ | $c_m$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 503 | 3 | 256 | 216 | 72 | 55 | 1595 bits | 4024 bits | $2^{285}$ | $2^{170}$ | 0.182 | 0.160 |

**Tabla 12: Seguridad muy alta**

# NTRU comparado con diferentes criptosistemas

| | NTRU | RSA | McEliece | GGH |
|---|---|---|---|---|
| **Velocidad de encriptación**[6] | $N^2$ | $N^2$ | $N^2$ | $N^2$ |
| **Velocidad de Desencriptación**[7] | $N^2$ | $N^3$ | $N^2$ | $N^2$ |
| **Clave Pública** | N | N | $N^2$ | $N^2$ |
| **Clave Privada** | N | N | $N^2$ | $N^2$ |
| **Expansión de Mensajes** | Varía | 1–1 | 2–1 | 1–1 |

**Tabla 13: Comparaciones de NTRU contra otros criptosistemas (de [4.1])**

[6] Encriptar en NTRU solamente requiere sumas y shifts, no multiplicaciones.
La encriptación en RSA es de orden $N^2$ a menos que se usen exponentes de encriptación pequeños.

[7] Asintóticamente la encriptación y desencriptación de NTRU son de orden $N.\log(N)$ con FFT.

# Apéndice IV: RPK

## Generación de claves

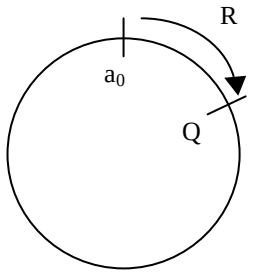
En el sistema RPK, una clave privada es equivalente a una n-tupla de números los cuales especifican la cantidad de veces que arbitrariamente cada uno de los componentes del *mixture generator* (generador de mezclas) serán avanzados. Estos pueden ser interpretados como distancias (medidas en número de pulsos de reloj) dentro de la salida periódica de cada componente, comenzando desde un punto conocido y fijo de partida.

La clave publica correspondiente a una clave privada es el estado final del generador de mezclas que resultaría si cada componente fuera pulsado un número de veces dado por la parte correspondiente de la clave privada.

Un generador de mezcla consiste en un único generador binario seudoaleatorio, llamado el *generador mezclador*, cuyas salidas o estados son usados sucesivamente para elegir, de una forma sin memoria, salidas de miembros de un conjunto de otro generador de componentes binarios.

## Encriptado

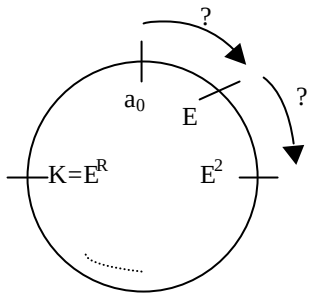
Cuando la persona **B** decide encriptar un mensaje **P** que solo pueda ser desencriptado por la persona **A** (usando su clave privada), lo primero que debe hacer es generar una clave de inicialización verdaderamente aleatoria (*) **R** que tan solo será usada durante el proceso para encriptar **P** (la manera de elegir éste número de forma verdaderamente aleatoria será tratado más adelante). Seguidamente **B** computa una clave **Q** a partir de **R**. Esto quiere decir que **Q** representa el estado de los generadores de componentes en el momento **R**, comenzando desde el estado inicial $\mathbf{a_0}$. Es entonces cuando **B** incluye **Q** en el cabezal del texto codificado, para ser transmitido sin encriptar.




Generador de mezclas

(*) verdaderamente aleatoria significa que se debe usar por ejemplo hardware especializado para ésta tarea. Esto se debe a que si es posible predecir cual será la clave elegida toda la seguridad del sistema se verá amenazada. Es por eso que no se puede confiar en los métodos corrientes de generación de números aleatorios.

Para proseguir con el proceso de encriptación, **B** carga los generadores de componentes con el estado inicial **E** (clave publica de la persona **A**) y nuevamente usa la clave de inicialización **R** para computar un estado final de inicialización del generador (estado **K**) exponenciando la clave publica de **A** , tomando como exponente **R**.




exponenciar = pulsar el reloj = avanzar los generadores de componentes a través de sus estados.

El estado **K** es usado como un estado final de inicialización del generador a partir del cual se comenzará a crear el texto encriptado. La persona **B** genera el cuerpo del texto encriptado **C** usando la salida obtenida al pulsar el generador de mezclas comenzando desde el estado **K**, operando con dicha salida y combinándola con los bits del texto sin encriptar **P**.

Aunque la técnica de combinación más simple solamente involucra un XOR bit a bit entre el texto plano (sin encriptar) y la salida del generador de mezclas, este acercamiento no es el mejor.

Otro enfoque posible es el de crear una tabla de intercambio de posiciones. La idea es simple, se trata de crear una tabla que contenga permutaciones a ser realizadas en el texto. Esta tabla será construida utilizando la salida del generador de mezclas. Para realizar las permutaciones se puede ver al texto como un arreglo de bits o como un arreglo de bloques de bytes a ser permutados.

Luego de haber obtenido el texto permutado, se continúa usando la salida del generador de mezclas para realizar un simple XOR (o cualquier función adecuada) con dicho texto y así llegar al texto cifrado final. Esta técnica es muy usada ya que el texto cifrado continúa manteniendo su largo original.

Ejemplo del uso de la tabla de permutaciones:

> Supongamos que tenemos el texto ABCD y el arreglo [0, 1, 2, 3] (el largo del arreglo es igual al número de bloques que contenga el texto).
> Ahora usando la salida del generador de mezclas obtenemos 2 enteros módulo 4 que representarán posiciones en el arreglo. La permutación consistirá en intercambiar el lugar entre el entero obtenido y el resultado de (entero + 1).
> Sean 2 y 1 los enteros, por lo tanto las permutaciones serían el lugar 2 por el 3 y el 1 por el 2:
>
> [0, 1, 2, 3] → [0, 1, 3, 2] → [0, 3, 1, 2]
> [0 1 2 3] [0 1 2 3] [0 1 2 3]
>
> La tabla de permutaciones la obtenemos tomando de a dos posiciones consecutivas.
> En nuestro ejemplo la tabla es la siguiente: (0 → 3) y (1 → 2)
> Con lo que llegamos a que el texto ya permutado resulta ser DCBA.

Este método tiene una propiedad que lo hace muy favorable, dicha propiedad es que para deshacer las permutaciones simplemente se vuelven a realizar los intercambios marcados en la tabla, es decir, si aplicamos nuevamente este algoritmo obtenemos el texto sin permutar.

En resumen, el proceso de encriptación involucra los siguientes pasos, cada uno de los cuales es alcanzado usando el generador de mezcla y sus componentes:

- Generar una clave aleatoria de inicialización R y usarla para exponenciar el estado inicial, con lo que se genera una clave Q la cual es incluida en el cabezal, precediendo el cuerpo principal del texto cifrado.

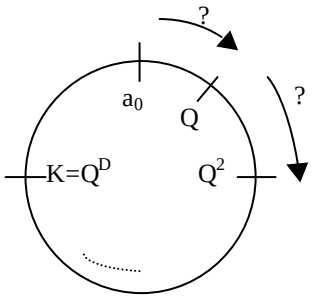
- Nuevamente usar R para exponenciar la clave pública E, con lo que se genera un estado final de inicialización del generador K.

- Comenzando del estado K, correr el generador de mezclas para obtener el string de salida y combinarlo con el texto plano P para obtener el cuerpo principal del texto cifrado C.

# Desencriptado

Para desencriptar el texto cifrado, la persona **A** usa el estado dado por la clave pública **Q** contenida en el cabezal del mensaje para computar el estado del generador que corresponde a $\mathbf{Q^D}$, donde el exponente es su clave privada **D**. El proceso de exponenciar **Q** a la **D** es hecho usando el mismo tipo de proceso que se uso para exponenciar **E** a la **R** durante la encriptación. Observamos que el estado resultante es **K**, ya que **Q** representa el estado del generador después de **R** pulsaciones comenzando

desde el estado inicial $\mathbf{a_0}$ y el estado después de $\mathbf{R^D}$ pulsaciones es exactamente **K**, como se acotó anteriormente.




$$E^R = (a_0{}^D)^R = K = (a_0{}^R)^D = Q^D$$

La persona **A** ahora ya puede correr el generador de mezclas tomando como estado inicial **K** para obtener el string necesario para invertir el proceso de combinación usado durante la encriptación. Ya que el generador de mezclas comienza desde el estado **K** para tanto la encriptación como la desencriptación, el string de salida será idéntico en los dos casos.

En resumen los pasos específicos para desencriptar son los siguientes:

- Usando la clave privada, exponenciar el estado Q contenido en el cabezal del texto cifrado para computar el estado final de inicialización K. El estado del generador de mezclas será dado por K.

- Para cada bloque del cuerpo del texto cifrado, correr el generador de mezclas para obtener el string de salida y combinarlo con el texto cifrado (usando un XOR o la función inversa de la que se uso para encriptar) para invertir el proceso de encriptado.

Por información más detallada sobre la encriptación y desencriptación referirse a [5.1]

# Seguridad

Selección de parámetros: Se recomienda para aplicaciones que requieran gran nivel de confiabilidad, es decir, que la información con la que están trabajando esté bien protegida, que el largo del componente más largo del generador debería ser de por lo menos 607. Esta recomendación actualmente se refleja en las distintas aplicaciones comerciales que implementan RPK. Esto se debe a que el criptosistema RPK se basa en el problema del algoritmo discreto para el cual el método mas eficiente para lograr resolverlo es mediante el algoritmo de D. Coppersmith [4.16]. Este algoritmo ha estimado empíricamente de acuerdo a la tecnología con que se cuenta en el momento que claves de largo 521 pueden llegar a ser quebradas dentro de pocos años. El valor de N=607 debería ser adecuado para la mayoría de las aplicaciones comerciales, mientras que niveles mas bajos pueden satisfacer perfectamente aplicaciones solo con medidas de protección transitorias.

Se describirán a continuación algunos posibles ataques contra este criptosistema basados en lo documentado en [5.2].

## Ataques a la clave final de inicialización del generador (K)

Ataque Directo Basado en el Conocimiento de Q y E: La seguridad del sistema en términos de esfuerzo computacional requerido para obtener la clave final de inicialización del generador a partir del conocimiento de la clave pública Q y la clave pública E está relacionada con la seguridad del sistema de distribución de claves Diffie-Hellman. Esto es, la clave final de inicialización del generador K representa el polinomio $E^R=(x^D)^R=K(x)=(x^R)^D=Q^D$

Por lo tanto, si es posible computar K a partir únicamente de E y Q entonces puede ser posible computar $(x^R)^D$ a partir de $x^D$ y directamente $x^R$ o debe ser posible computar D o R.

Computar D o R significa resolver el problema del logaritmo discreto. Computar directamente $(x^R)^D$ a partir de $x^D$ y $x^R$ sin conocer R o D es equivalente a un ataque satisfactorio al sistema de distribución de claves Diffie-Hellman, y no se conoces ningún ataque de ese tipo en general.

Ataque Criptoanalítico Convencional: RPK es un criptosistema de clave pública no determinístico debido a que la clave de inicialización aleatoria R es escogida independientemente de forma verdaderamente al azar cada vez que se encripta un texto. Esto quiere decir que no existe una correspondencia 1 a 1 entre R y la salida específica del generador usada para encriptar, aunque el mismo texto sea encriptado más de una vez el texto cifrado será distinto de forma aleatoria. Por consiguiente no puede existir un ataque directo contra una clave que es directamente comparable con un ataque de "texto-elegido" en un criptosistema convencional, en el sentido que uno no puede ganar información a cerca de un R usado para encriptar cierto texto de otro R usado en otra ocasión.

Sin embargo, podemos discutir la seguridad del sistema desde el punto de vista de un ataque contra la salida del generador y el procedimiento de combinación que es parecido a un ataque de "texto-elegido". Este tipo de ataque es uno en el que el criptoanalista tiene acceso a todas las claves públicas y tiene a su disposición un sistema criptográfico, incluyendo acceso directo a la salida del generador el cual puede usar para generar pares de texto plano - texto cifrado. Esto significa que el criptoanalista puede estudiar cualquier tipo de secuencias de salidas, comenzando desde cualquier estado inicial.

Generando un gran número de porciones de salidas (fragmentos de búsqueda) y correlacionando cada uno de ellos con un texto cifrado desconocido, el criptoanalista puede tratar de descubrir solapamientos los cuales pueden ser detectados mediante análisis analítico. La probabilidad de detectar solapamientos depende del largo de los mensajes y de la velocidad a la cual el generador pueda ser corrido, pero análisis probabilísticos muestran que la probabilidad de algún solapamiento es extremadamente chica.

Por ejemplo, asumiendo que el generador es capas de ser pulsado a 1000 GB por segundo ($2^{40}$ bits por segundo), que el largo promedio del texto plano es de 1 GB ($2^{30}$ bits) y que un solapamiento con un fragmento de búsqueda puede ser efectivamente detectado instantáneamente (en 0 segundos) con una correlación usando fragmentos de búsqueda de solo $2^{10}$ bits de largo, entonces en el caso del generador más chico del tipo Geffe (ver 4.3.2.2) el tiempo esperado para encontrar un texto cifrado en particular esta en el orden de los $2^{240}$ segundos.

El análisis probabilístico también muestra que, bajo las mismas consideraciones, la probabilidad de algún solapamiento correspondiente a mensajes cuya clave fue elegida al azar es insignificante, por lo que un ataque de "texto conocido" basado en este acercamiento es inútil.

El único ataque restante posible es un ataque en cada texto cifrado tratando de descubrir la clave final de inicialización usada cuando fue encriptado. En tal tipo de ataques, un criptoanalista trata de determinar el estado del generador basado en el conocimiento de cierta parte de la salida usada para encriptar el texto.

# Aplicaciones prácticas

InvisiMail: InvisiMail es un agregado de seguridad basado en estándares para software de correo electrónico POP3/SMTP/IMAP4. InvisiMail se integra de forma transparente y sencilla para el usuario con cualquier protocolo de correo electrónico para Internet.

Communicado: Communicado Data Ltd., es una compañía del Reino Unido pionera en redes de datos satelitales, provee sistemas para rápido acceso a Internet vía satélites, una capacidad muy útil para compañías que requieren de un gran ancho de banda.

Más aplicaciones pueden ser encontradas en [5.4].

# Comparación por la complejidad de su implementación entre RPK y RSA

La figura 8 muestra curvas cualitativas típicas de "tiempo de procesamiento/complejidad" de RPK y RSA+3DES versus "bit/tasa de bits" del texto plano/cifrado con el mismo nivel de seguridad, para RPK el largo de la clave es de 607 bits mientras que para RSA es de 512 bits.

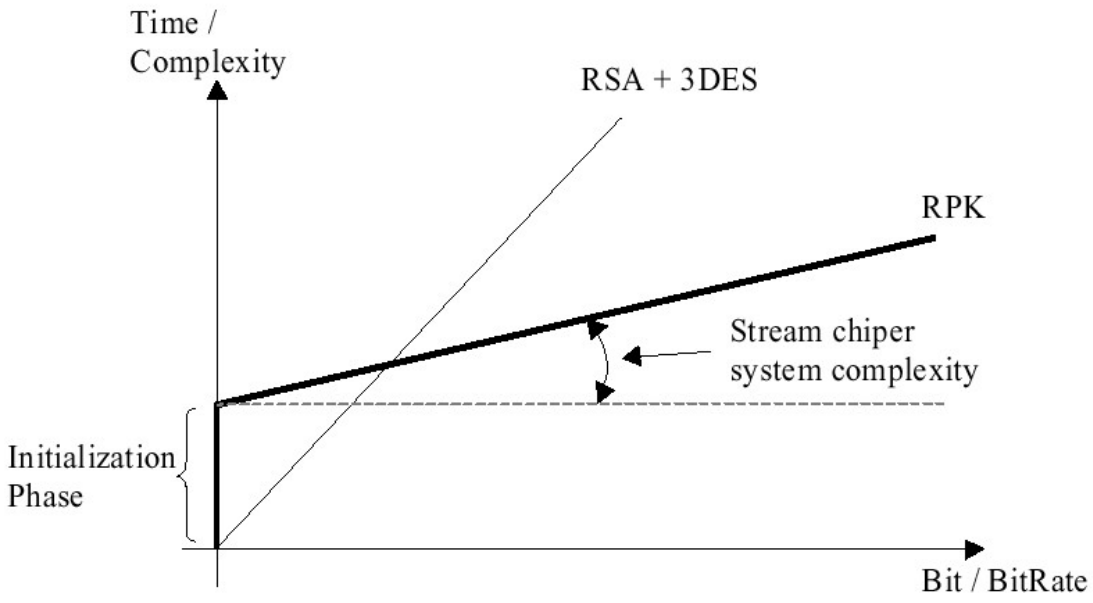



**Figura 8: Comparación entre RPK y RSA (de [5.3])**

Como se puede notar en el caso de RPK luego de un cierto tiempo de inicialización de la máquina, ésta puede encriptar y desencriptar muy eficientemente. De hecho, una vez que la máquina ha sido inicializada, el único factor que determina la capacidad de la tasa de bits de un sistema de RPK es el sistema de mezclado del texto plano con la salida del generador.

Un nivel de seguridad extremadamente alto puede ser garantizado ya que claves privadas muy largas, como por ejemplo de largo 1279 bits, no afectan la performance simplemente aumentan el tiempo de inicialización. La figura ilustra el hecho que la pendiente de la curva de RPK solo depende de la complejidad de f(Texto plano, Texto cifrado). La figura también nos muestra que el sistema RPK es menos apropiado en términos de complejidad para mensajes cortos. De hecho, el tiempo de inicialización es más largo que el tiempo de encriptación/desencriptación de RSA+3DES, pero con el aumento del largo del mensaje el sistema RPK también aumenta su conveniencia (ver [5.3]).

# Apéndice V: RSA

## Generación de claves

La generación de claves (ver [1.4]) requiere inicialmente de dos números primos grandes, que llamaremos *p* y *q* cuyo producto llamaremos *n*. Además se deben generar dos números enteros *a* y *b* tales que:

$$ab \equiv 1 \mod j(n)$$

$j(n)$ corresponde a la llamada función de Euler evaluada en *n* y que da la cantidad de números enteros positivos *m,* tales que cumplen:

$$1 \leq m < n \quad \text{y} \quad \gcd(m, n) = 1$$

es decir que da la cantidad de números primos con *n*. La fórmula para calcular la función de Euler es conocida (ver [1.6]) y en este caso $j(n) = (p-1)(q-1)$

Una vez hechos estos cálculos se tiene que la clave privada está formada por *p, q* y *a* mientras que *n* y *b* son públicos. Está claro que si se logra factorizar *n* se obtendrán inmediatamente los valores privados *p* y *q*.

## Encriptación y desencriptación

Para encriptar un mensaje *m* se debe calcular:

$$m^b \mod n$$

mientras que para desencriptar un mensaje encriptado *e* lo que se debe hacer es:

$$e^a \mod n$$

Para ver que la desencriptación funciona (según [1.4]), hay que tener en cuenta previamente que si:

$$ab = 1 \mod j(n)$$

entonces

$$ab = tj(n) + 1 \quad \text{con} \quad t \geq 0 \qquad (1)$$

y además existe una propiedad que indica que:

$$b^{j(n)} \equiv 1 \mod n \qquad (2)$$

Entonces se tiene que:

$$\left(m^b\right)^a \equiv m^{tj(n)+1} \mod n \qquad \text{por propiedad (1)}$$

$$\equiv \left(m^{j(n)}\right)^t m \mod n \qquad \text{por propiedades de la exponenciación}$$
$$\equiv 1^t m \mod n \qquad \text{por propiedad (2)}$$
$$\equiv m \mod n$$

y la desencriptación funciona.

# Seguridad

La seguridad del criptosistema puede ser evaluada desde diferentes puntos de vista. El más básico es el que concierne a la posibilidad de factorizar el valor de *n*, mientras que hay otros que solamente son posibles ante un mal uso del sistema (las llamadas *fallas de protocolo*) y aún otros que solo son posibles cuando se usan valores pequeños para los exponentes (pero que son relevantes en el mundo de los bajos recursos, donde el ahorro de memoria es siempre fundamental). Por último, con relación a este tema existe el problema de los ataques físicos a las implementaciones de los algoritmos, los cuales están comentados en el apéndice VI.

## Factorización

Como se dijo, este criptosistema basa su seguridad en el hecho de que no se conocen algoritmos eficientes para factorizar enteros. En realidad existen algunos métodos rápidos para lograr esto pero, o bien dependen de la forma particular de los enteros a factorizar o solo funcionan bien con números relativamente pequeños.

Dentro los métodos llamados de propósito general, en los que el ni el tamaño de la forma de los factores son relevantes se destacan el llamado *Quadratic Sieve* que fue creado en 1981 por Carl Pomerance y una variación posterior del mismo introducida por Pollard el 1988 y que recibe el nombre de *Number Field Sieve*.

El algoritmo de *Quadratic Sieve* es uno de los más eficientes conocidos (dentro de los de propósito general) y los estudios indican que la complejidad de el algoritmo es $o\left(e^{\sqrt{\ln(n)\cdot\ln(\ln(n))}}\right)$. Los estudios de todas maneras no son muy rigurosos dado que el algoritmo en si es muy complejo para ser estudiado matemáticamente con todo detalle.

Una ventaja importante de este algoritmo es que es fácilmente paralelizable, y de hecho en 1994 fue usado para factorizar un número de 129 bits correspondiente a una clave de RSA en 8 meses mediante el uso de máquinas conectadas a Internet. Por otro lado con el uso del *Number Field Sieve* se logró factorizar un número de 130 bits 15% más rápidamente que usando *Quadratic Sieve*, aunque este método tiene problemas para números mayores dada la gran cantidad de espacio de almacenamiento que necesita.

Posteriormente, en 1999 [6.3] un grupo de investigadores logró factorizar una clave de RSA de 512 bits utilizando el *Number Field Sieve*. Se utilizaron 35.7 años de CPU en total utilizando 292 estaciones de trabajo y una máquina CRAY que utilizó 224 horas de CPU y 3.2 Gb de RAM en realizar los cálculos necesarios. El tiempo total que llevó fue de 7.4 meses.

## Fallas de protocolo

El problema más conocido a este respecto es el que tiene que ver con el uso del mismo módulo *n* para más de una clave. Esto en principio parecería una buena idea dado que solo se requiere el cálculo de un *n* y entonces se generan muchas claves a partir de él con solo dar un par $a_i$, $b_i$ para cada usuario *i*. La cuestión es que se puede probar que cualquier usuario *U* que posea un exponente privado $a_U$ podrá

usarlo para factorizar $n$ y a partir de esto para obtener cualquier $a_i$ a partir del $b_i$ público. (ver prueba en [6.1]).

Otro problema de mal uso es el llamado *ataque de transmisión* (broadcast attack). Ese ataque es explotable en el caso que alguien desee enviar un mensaje a $k$ personas y por lo tanto proceda a encriptar exactamente el mismo mensaje $k$ veces con las claves de las $k$ personas y los envíe por un canal inseguro. En algunos casos, si un atacante es capaz de recuperar todos los mensajes cifrados puede recuperar el mensaje original siempre que $k \geq b$ [6.1].

Una forma de evitar este ataque sería agregar alguna información a los mensajes de forma que estos sean todos diferentes. Por ejemplo, en lugar de mandar el mismo mensaje $M$ a todos los receptores se podría mandar un mensaje del tipo $M_i = i2^m + M$ a cada receptor $i$ (donde se está suponiendo que $M$ tiene $m$ bits de largo). Esto se conoce como realizar un *relleno* (padding) del mensaje. Sin embargo se puede probar que el ataque es igualmente factible en este caso, así como en cualquier otro donde se aplique un polinomio fijo al mensaje antes de la encripción. Para inutilizar realmente el ataque se debe utilizar un relleno aleatorio.

Existe otro problema con RSA que puede ser explotado y que se produce cuando una de las partes encripta y envía dos mensajes relacionados usando el mismo módulo. En este caso de dice que dos mensajes $M_1$ y $M_2$ están relacionados cuando $M_1 = f(M_2) \bmod n$ donde $f$ es un polinomio conocido con coeficientes en $Z_N$. Nuevamente, esto funciona solamente si el exponente $b$ es pequeño.

Una extensión a este ataque [6.1] indica que si quien envía los mensajes utiliza un relleno aleatorio que consista solamente en agregar algunos bits al principio o al final de un mensaje, esto es suficiente para hacer que dos mensajes estén relacionados. Es decir: si un atacante logra interceptar un mensaje que $X$ le envía a $Y$ y evitar que llegue forzando a $X$ a reenviarlo varias veces (con un *relleno* diferente) podrá obtener el texto plano correspondiente al mismo siempre que $b$ sea pequeño.

## Exponente privado pequeño

Para reducir el tiempo necesario para desencriptar, sería deseable utilizar un valor pequeño para $a$ en lugar de un número aleatorio, con el fin de disminuir el tiempo de ejecución del algoritmo de exponenciación modular.

Lamentablemente se ha probado [6.4] que si se utiliza un valor de $a$ suficientemente pequeño el criptosistema puede quebrarse completamente. Numéricamente el teorema indica que si $n = pq$ con $q < p < 2q$ y si $a < \frac{1}{3}n^{1/4}$ el valor de $a$ puede ser calculado en tiempo lineal. Esto quiere decir que si $n$ tiene 1024 bits $a$ deberá tener alrededor 256 bits para tener total seguridad de que este ataque no sea posible.

Una medida que se puede tomar para evitar este ataque sin necesidad de aumentar el tamaño de $a$ es la de aumentar en su lugar el tamaño del exponente público $b$. Se puede probar que si $b > n^{1.5}$ el ataque anterior no es posible. De esta forma lo único que se logra es disminuir el tiempo necesario para desencriptar aumentando a su vez el tiempo necesario para encriptar. Esto tal vez no sea de una gran ayuda en general, pero dependiendo de la aplicación particular (por ejemplo una en la que el cifrado se realizara fuera de la tarjeta y esta se utilizara solamente para decodificar) podría ser de mucha utilidad.

Otra medida posible que se puede tomar es utilizar la variante de RSA que trabaja con el teorema chino del resto. Esta versión no es vulnerable a este ataque aunque si lo es ante ataques físicos por generación de fallas en la tarjeta como se comenta en el apéndice VI.

## Exposición parcial de la clave

Esta vulnerabilidad permite a un atacante descubrir un exponente privado $a$ completo a partir de algunos de sus bits. Esto será posible siempre que la clave pública sea relativamente pequeña. Lo que se

prueba [6.1] es que si $b < \sqrt{N}$ un atacante podrá recuperar la clave $a$ completa a partir de los $\lceil \frac{1}{4} \rceil$ bits menos significativos de $a$ en tiempo lineal en $b \log_2 b$

## Exponente público pequeño

Esta es una posibilidad que ayuda también a disminuir el tiempo que requiere la encriptación. Si bien el menor $b$ posible es 3 para evitar los ataques mencionados anteriormente se recomienda que el valor no sea menor a $2^{16} - 1 = 65537$ (en particular este es el valor que utiliza la tarjeta empleada en este proyecto) Este número tiene la característica que permite que la desencriptación utilice solamente 17 multiplicaciones en lugar de las alrededor de 1000 que se necesitan cuando se utiliza un $b$ aleatorio.

A diferencia del caso anterior, los ataques conocidos a criptosistemas con exponente público pequeño (que se basan en general en el algoritmo LLL) [6.1] están muy lejos de quebrar el sistema completamente.

# Apéndice VI: Ataques contra algoritmos criptográficos implementados en tarjetas inteligentes

## Introducción

En la criptografía tradicional los algoritmos que se estudian son analizados exclusivamente desde un punto de vista matemático abstracto. Si un atacante no logra encontrar un algoritmo que logre resolver cierto problema considerado intratable en orden sub-exponencial y no es posible para el mismo probar todas las combinaciones de claves por fuerza bruta se concluye que el criptosistema es seguro. Estas conclusiones son válidas siempre y cuando no se consideren las implementaciones particulares de cada algoritmo.

Por otro lado, toda implementación práctica de un algoritmo criptográfico se realiza sobre un aparato físico particular con características propias que no están cubiertas por el modelo matemático general. Estas características podrían ser explotadas por un atacante que pueda manipular el dispositivo con el objetivo de alterar el algoritmo que corre en el mismo y obtener información secreta a partir de él.

Durante mucho tiempo estos dispositivos eran grandes y no eran accesibles por cualquier persona mientras que los métodos para atacarlos eran complicados y costosos. Pero con el advenimiento de los dispositivos portátiles, como las tarjetas inteligentes, que corren algoritmos criptográficos estos se han hecho accesibles para un público sustancialmente mayor; mientras que se han desarrollado ataques que se basan en información fácilmente accesible como el tiempo de ejecución de los algoritmos y la energía consumida por el dispositivo los cuales pueden ser llevados a cabo a muy bajo costo.

Hay muchas formas diferentes de atacar los algoritmos criptográficos cargados en un dispositivo portable (en este caso nos centraremos en las tarjetas inteligentes). Muchos de ellos son idénticos a los que se pueden utilizar para atacar criptosistemas implementados en un sistema informático general, pero hay otros que son exclusivos para el entorno de las tarjetas inteligentes. Incluso hay ataques imprácticos en general que resultan muy efectivos al atacar este tipo de sistemas de bajos recursos y hay algunos, que más allá de la cantidad de recursos del sistema, son efectivos a causa de que el atacante puede disponer del dispositivo criptográfico para analizarlo en detalle.

Por estas razones la resistencia de las implementaciones de los algoritmos y de los dispositivos en que serán ejecutados este tipo de algoritmos actualmente son seriamente consideradas tanto en la industria como en ambientes académicos.

En forma general los ataques se pueden dividir en dos tipos que llamaremos los agresivos y los no-agresivos. El criterio para diferenciarlos se basa en si la tarjeta es destruida durante el ataque o no. En general los métodos agresivos tienen como finalidad entender con detalle el funcionamiento de la tarjeta con el fin de desarrollar métodos no-agresivos eficientes y fácilmente repetibles los cuales son, por su simpleza, los más peligrosos (pues se pueden aplicar rápidamente y a gran escala)

A continuación se detallarán algunos los diferentes ataques posibles y las medidas que se pueden tomar contra ellos, cuya efectividad se mide por la penalización extra en tiempo y costo que su presencia impone al atacante en cuestión.

A este respecto IBM [10.7] clasifica a los posibles atacantes en tres categorías:

Clase 1. Personas inteligentes, sin conocimiento completo del sistema y que poseen equipamiento electrónico de complejidad moderada. Ellos probablemente traten de explotar alguna debilidad conocida del sistema antes de intentar descubrir una nueva.

Clase 2. Personas con educación y herramientas especializadas quienes poseen un conocimiento bastante completo del funcionamiento interno del sistema

Clase 3. Organizaciones con dinero que pueden reunir equipos de personas calificadas con conocimientos especializados y complementarios (por ejemplo grupos de atacantes de tipo 2) Pueden realizar un análisis con detalle del sistema y diseñar nuevos y sofisticados ataques.

La mayoría de las técnicas que se mostrarán a continuación podrán en general ser aplicadas por atacantes de tipo 2, aunque algunas requerirán más infraestructura y se necesitará un atacante de clase 3 y algunas otras son suficientemente sencillas para ser llevadas a cabo por atacantes de tipo 1.

# Ataques al CAD

Suponiendo que no se pueda atacar a la tarjeta directamente, por carecer de los conocimientos de electrónica o de las herramientas necesarias, el punto más vulnerable a ataques es el determinado por la interfase entre la tarjeta, el CAD y la máquina a la que este está conectado.

Como defensa inicial es claro que la tarjeta debe ser resistente al envío de instrucciones ilegales y combinaciones de bits no especificadas pues este sería el modo de ataque más sencillo. Siguiendo por este camino, una tarjeta sin soporte criptográfico alguno siempre va a poder ser atacada, dado el suficiente tiempo para probar. La razón para esto es que un atacante que pruebe enviar secuencias de bits más o menos aleatorias a la tarjeta, tarde o temprano va a dar con los comandos correctos para efectuar las operaciones que desea (por ejemplo recargar una tarjeta telefónica o que contenga dinero electrónico)

Estos tipos de ataques, no son muy prácticos en la actualidad ya que cualquier tarjeta que maneje valores tendrá siempre soporte criptográfico que exija la autenticación del lector antes de enviar comandos y posiblemente también encripte la información intercambiada.

# Análisis de consumición de energía

Este tipo de ataques fue uno de los primeros desarrollados pero también, junto con el mencionado anteriormente, es de los menos poderosos en la práctica. Consiste básicamente en interpretar las medidas de consumo de energía por el dispositivo durante la ejecución de operaciones criptográficas. El objetivo fundamental del ataque es tratar de averiguar que instrucciones particulares se están ejecutando en cierto momento así como sus posibles entradas y salidas a partir del consumo de energía del dispositivo. Es claro que para tratar de conseguir esto el atacante necesita conocer exactamente la implementación específica, instrucción por instrucción, del algoritmo que se está atacando.

Un ejemplo típico de un algoritmo pasible de ser atacado por este método es el de exponenciación conocido como *square and multiply* muy usado en la mayoría de las implementaciones de RSA y otros algoritmos criptográficos que requieren exponenciaciones modulares. Por ejemplo un algoritmo de este tipo implementado con este seudocódigo:

Entrada: base $b$, exponente $e = (e_1, ..., e_l)_2$ y módulo $m$
Salida: $c = b^e \bmod m$

1) c := 1
2) for $i$ := 1 to $l$ do
3) $c := c^2 \bmod m$
4) if $e_i = 1$ then $c := c * b \bmod m$ end if
5) end for

podría ser atacado mediante la observación de la energía consumida en cada iteración del mismo. De esta forma se podría llegar a detectar si se está ejecutando el paso 4) o no y por lo tanto se obtendrían los bits del exponente privado *e*.

Las medidas que se pueden tomar para impedir este tipo de ataques [10.4] incluyen evitar la ejecución de instrucciones condicionales basadas en los datos privados, proteger el código específico de la implementación de forma que sea inaccesible a los atacantes, y también ejecutar ciertas instrucciones aleatoriamente para enmascarar el código real ejecutado. Por ejemplo en el algoritmo comentado antes una solución simple consistiría en calcular siempre el producto del paso 4) y guardarlo sólo si $e_i = 1$.

# Ataques de tiempo de ejecución

En forma similar al caso de la corriente consumida, el tiempo de ejecución de ciertos algoritmos criptográficos muestra pequeñas diferencias dependientes de los datos de entrada que reciben. Ese tipo de variaciones se debe a instrucciones condicionales, falla del caché, manejo de casos especiales y optimizaciones varias. Estos ataques son implementables a bajo costo ya que lo único que se requiere es un sistema que permita medir tiempos con relativa exactitud. Además, en el entorno particular de las tarjetas inteligentes, las cuales poseen un sistema operativo muy simple y que no es multitarea, se podrá medir con mucha mayor exactitud el tiempo que tardó una cierta operación que si se lo hiciera por ejemplo en una PC común con un sistema operativo multitarea (es decir no DOS) que posiblemente esté corriendo muchos procesos a la vez.

Al igual que en el caso anterior el atacante requerirá tener un conocimiento detallado del algoritmo así como de su implementación y las razones por las cuales se presentan las diferencias de tiempo apreciadas.

Siguiendo con el ejemplo del algoritmo de *square and multiply* presentado anteriormente un ataque de este tipo se podría realizar de la siguiente manera [10.4]:

Se define $t(y, b_1, \ldots, b_k)$ como el tiempo que tarda la ejecución de las $k$ primeras iteraciones del algoritmo dada una entrada $y$ y un exponente privado cuyos primeros $k$ bits son $b_1, \ldots, b_k$. Suponiendo que el atacante ya averiguó los primeros $k$ bits del exponente puede calcular $t(y, b_1, \ldots, b_k)$ y al restar este tiempo del tiempo total de ejecución del algoritmo obtiene el tiempo que tardan en ejecutarse las $l - k$ iteraciones restantes del loop.

En ese momento el atacante realiza una serie de corridas del algoritmo con diferentes entradas asumiendo como hipótesis estadística que $b_{k+1} = 1$. Entonces separa las entradas en dos conjuntos, aquellos en los que se tardó más en ejecutar el paso 4) en la iteración $k$ (conjunto $Y_0$) y aquellos en los que tardó menos (conjunto $Y_1$) Es claro que la diferencia entre ambos es subjetiva y cuanto mayor sea la misma mayor será la debilidad del algoritmo ante este tipo de ataques. Entonces el siguiente paso consiste en tomar muestras aleatorias de ambos conjuntos para definir dos variables aleatorias $T(Y_0)$ y $T(Y_1)$ y realizar la siguiente observación: si $b_{k+1} = 0$ no debería haber ninguna diferencia en promedio entre el tiempo que tardaron las $l - k$ últimas iteraciones sin importar que la entrada fuera elegida de $T(Y_0)$ o de $T(Y_1)$, es decir estas dos variables aleatorias tendrás propiedades estadísticas similares. Sin embargo si $b_{k+1} = 1$ entonces estas variables tendrán propiedades diferentes y el tiempo promedio de la ejecución de la últimas iteraciones será mayor en un caso que en otro.

Como extensión a este método se tiene otro más general que no necesita partir las entradas de acuerdo a las demoras del tiempo de ejecución. La idea es que q partir de los bits $b_1, \ldots, b_k$ que son conocidos un atacante podría calcular:

$$T_b(Y) = t(Y, b_1, \ldots, b_l) - t(Y, b_1, \ldots, b_k, b) \text{ donde } b \in \{0,1\}$$

El primer término es el tiempo de ejecución completa del algoritmo mientras el segundo depende del siguiente bit desconocido $b$ del exponente privado. Si se asume entonces que las diferentes iteraciones del algoritmo de exponenciación son independientes, se espera que la varianza de $T_b(y)$ descienda

cuando se acierte el valor de $b$ y crezca en caso contrario. Eligiendo entonces entradas aleatorias $y \in Y$ se podría aproximar la varianza de $T_b(y)$ y así calcular el siguiente bit de la clave privada.

La medida más obvia para defenderse ante este tipo de ataques consiste en evitar que los algoritmos cambien su tiempo de ejecución dependiendo de las entradas, aunque esto es un proceso complicado y posiblemente costoso en términos de eficiencia. Llegando a un extremo, la ejecución del algoritmo deberá ser lineal sin realizar optimizaciones de performance lo que resultará en que el tiempo de ejecución será siempre el del peor caso. Además, como el tiempo de ejecución depende de otros factores además del algoritmo en sí (excepciones de hardware, interferencia con otros procesos, diferentes instrucciones de máquina, etc.) una implementación segura en una plataforma puede ser insegura en otra.

Otra posibilidad es transformar el algoritmo agregando otro parámetro que modifique los valores intermedios calculados y las operaciones que se realizan pero sin afectar el resultado final. El valor de este parámetro deberá ser elegido aleatoriamente en cada ejecución del programa y su valor no debe ser conocido por nadie fuera del dispositivo.

# Examen microscópico (*microprobing*)

Este es el método de ataque agresivo por excelencia y consiste en aplicar ingeniería reversa sobre el hardware de la tarjeta examinando sus circuitos usando un microscopio óptico especial.

El proceso consiste [10.8] en eliminar el recubrimiento plástico del chip mediante el uso de algún tipo de ácido y posteriormente analizar una por una las diferentes capas de circuitos integrados que conforman la tarjeta. Cuando se termina el análisis de una capa esta se destruye para dejar visible la siguiente y continuar investigando. De esta forma, un atacante con conocimientos de electrónica y un microscopio potente puede detectar fácilmente los diferentes componentes de una tarjeta como ser la ROM, RAM, EEPROM, ALU, etc.

Una vez que se tiene el mapa completo del chip se pueden intentar extraer los contenidos de la ROM y la EEPROM mediante el uso de instrumentos que emulan las señales que envía el procesador de la tarjeta y las comunicaciones con el lector o mediante la reescritura del código como se menciona en la siguiente sección.

Es claro que este método de ataque es muy sofisticado y requiere conocimientos avanzados, tiempo para analizar cuidadosamente el chip e instrumentos electrónicos caros de los que no dispone cualquier persona. Esto puede hacer el método no factible para personas comunes, aunque si para organizaciones que dispongan de recursos para invertir en este tipo de ataques. Al respecto se considera que, dado el suficiente tiempo e inversión, cualquier dispositivo supuestamente resistente a ataques podrá ser completamente analizado.

Las medidas de defensa que se pueden tomar ante el examen microscópico consisten en hacer más complicado el diseño del microchip, por ejemplo mediante la utilización de instrucciones no estándar en el procesador o mezclando los cables del bus para hacerlos más intrincados y dificultar su estudio. Otros métodos más sofisticados pueden ser la utilización de elementos aleatorios en el clock del chip y el procesamiento en paralelo logrando así introducir un elemento de no-determinismo en el procesamiento interno del dispositivo.

También se pueden diseñar soluciones más radicales como circuitos que destruyan el chip ante eventos anormales. Estos circuitos podrían activarse ante la detección de variaciones en la luz o en la temperatura, aunque los mismos también podrían ser detectados previamente por el atacante y ser desconectados antes de que puedan entrar en acción. Hay que considerar además, que a medida que el tamaño de los procesadores disminuye el uso de microscopios ópticos será cada vez más inapropiado necesitándose instrumentos que utilicen longitudes de onda más cortas (ej. microscopios electrónicos) cuyo costo pondrá momentáneamente estas técnicas fuera del alcance de muchos atacantes.

# Reescritura de chips y ataque a compuertas lógicas

Cuando la implementación de la lógica de la tarjeta es familiar para el atacante se puede obtener información (y en particular claves privadas) a partir de la misma mediante el ataque a determinadas compuertas lógicas o rescribiendo ciertas localizaciones de la memoria [10.2].

Un ejemplo de este tipo de ataques se puede aplicar a algunas tarjetas que poseen un fusible que se quema al terminar su ciclo de testeo y que inhabilita ciertos algoritmos de prueba los cuales pueden, por ejemplo, leer secuencialmente todos los datos de la memoria o efectuar otras operaciones no válidas para una tarjeta ya puesta en uso. En este caso sólo se necesita cerrar el circuito que fue cortado por el fusible para poder acceder a todas estas funciones "ocultas".

En el caso de la reescritura de chips se pueden usar microscopios con cortadores laser (*laser cutter microscopes*) para sobrescribir los bits de la ROM o de la EEPROM. El segundo caso es más favorable dado que los bits se pueden cambiar tantas veces como se quiera dado que la memoria es regrabable. La idea de estos ataques es alterar el código de los algoritmos para simplificarlos, por ejemplo cambiando saltos condicionales a saltos incondicionales o directamente eliminar algunas instrucciones de los mismos para inutilizarlos o facilitar la obtención de la información secreta.

Un ejemplo de esto [10.2] podría ser un algoritmo que copia el contenido de una sección de la memoria de la tarjeta al puerto serial. Si su código es algo del tipo:

```
1  b = dirección_origen
2  a = largo_respuesta
3  if (a == 0) goto 8
4  emitir (*b)
5  b = b + 1
6  a = a – 1
7  goto 3
8  ...
```

entonces, si se puede alterar el código cambiando el loop de la línea 3 o el decremento en la línea 6 se podría lograr que el algoritmo devuelva todo el contenido de la memoria.

Los microscopios necesarios para este tipo de ataques se pueden encontrar en los laboratorios de biología celular en muchas universidades de los países desarrollados por lo que pueden estar al alcance de los estudiantes sin supervisión y además pueden ser adquiridos por algunos miles de dólares.

# Generación de fallas

Un conocimiento detallado de la lógica interna de una tarjeta permite llevar a cabo este tipo de ataques no-agresivos que consisten en inducir errores en las operaciones del chip con el fin de realizar operaciones no permitidas o engañar a protocolos criptográficos. Estas últimas técnicas son las que se comentarán con más detalle.

Las fallas que se mencionan aquí se provocan sometiendo a la tarjeta a ciertos efectos físicos como ser ionización o radiación de microondas. Es importante resaltar que lo que se busca aquí no es alterar el código de los programas y hacer que fallen, sino que la idea es que estos corran pero manejando valores equivocados. Se asume además que el atacante dispone físicamente del dispositivo y puede repetir el experimento de generación de fallas tantas veces como quiera.

A continuación se mostrarán algunos ataques basados en generación de fallas que se pueden aplicar a los criptosistemas más populares y algunas medidas que se pueden tomar para evitarlos.

# Ataques a RSA

A lo largo de toda esta sección se supondrá que $n = pq$ es el módulo bajo el cual se harán los cálculos (siendo $p$ y $q$ dos números primos) mientras se asumirá que $e$ es el exponente público y $d$ el exponente privado. A partir de estas definiciones se tiene que el texto cifrado $c$ correspondiente a un mensaje $m$ se obtiene como:

$$c \equiv m^e \bmod n$$

Además el exponente privado $d$ se notará como:

$$d_{t-1}\ d_{t-2} \ldots d_i \ldots d_1\ d_0$$

donde los $d_i$ son valores binarios concatenados y $t$ es la cantidad de bits con que se representa $d$. Por otro lado se definirá el siguiente símbolo:

$$c_i = c^{2^i} \bmod n \quad \text{con} \quad i = 0,1,2,\ldots,t$$

De esta forma, dado un mensaje en texto plano $m$ el mismo puede ser expresado como el siguiente producto de $t$ valores módulo $n$ (ver apéndice V):

$$m \equiv c^d \bmod n \equiv c_{t-1}^{d_{t-1}} \cdots c_i^{d_i} \cdots c_1^{d_1}\ c_0^{d_0} \bmod n$$

El primer ataque posible [10.1] se basa en inducir un error en algún $c_i$ aleatorio del texto cifrado. A ese valor se le llamará $c_i$' y, suponiendo para simplificar que se logra producir un error en un solo de ellos, al desencriptar el dispositivo devolverá un valor que será:

$$m' \equiv c_{t-1}^{d_{t-1}} \cdots c'^{d_i}_i \cdots c_1^{d_1}\ c_0^{d_0} \bmod n$$

De esta forma el atacante tendrá $m$ y $m$' y, asumiendo de nuevo por simplicidad que $m$ tiene inversa módulo $n$, podrá dividirlos y simplificar los términos iguales en el numerador y el denominador para obtener que:

$$\frac{m'}{m} \equiv \frac{c'^{d_i}_i}{c_i^{d_i}} \bmod n$$

Por lo tanto este cociente equivaldrá a $\frac{c'_i}{c_i} \bmod n$ si $d_i = 1$ o será igual a 1 si $d_i = 0$. De esta forma el atacante podría computar por adelantado todos los valores posibles para $\frac{c'_i}{c_i} \bmod n$ (hay $t^2$ valores posibles) y compararlos con el cociente entre los mensajes para ver si coinciden. Se asume aquí que los $c_i$ tienen inversa módulo $n$ y los cocientes pueden ser calculados, aunque en general no todos los $c_i$ cumplirán esta propiedad complicando el análisis.

En caso de encontrar una coincidencia se podrá obtener (suponiendo que los valores del cociente de las $c_i$ son todos diferentes) el $i$ para el que se produjo el error y se sabrá que $d_i = 1$. Repitiendo este experimento varias veces se puede obtener suficiente información sobre el exponente privado como para hacer factible un ataque por fuerza bruta.

Es claro que en esta descripción se hicieron muchas simplificaciones dado que en realidad es muy difícil producir tan sólo un error en uno solo de los valores de las $c_i$ y además los errores que se produzcan se propagarán a los $c_i$ restantes. Se han desarrollado modelos más realistas que toman estas cuestiones en cuenta y que también son efectivos a costa de tener que comparar muchos más valores hasta encontrar una coincidencia.

Otro ataque factible [10.1], muy similar a este anterior consistiría en lograr una falla en uno de los bits de *d* (de nuevo se toma uno solo por simplicidad) El método se basa en encriptar un mensaje arbitrario *m* y luego producir el error en *d* al desencriptarlo. En este caso el mensaje obtenido es:

$$m' \equiv c_{t-1}^{d_{t-1}} \cdots c_i^{d'_i} \cdots c_1^{d_1} c_0^{d_0} \bmod n$$

y al igual que antes el atacante puede calcular:

$$\frac{m'}{m} \equiv \frac{c_i^{d'_i}}{c_i^{d_i}} \bmod n$$

Aquí los resultados posibles de este cociente son $\frac{1}{c_i} \bmod n$ si $d_i = 1$ o $c_i \bmod n$ si $d_i = 0$ (suponiendo de nuevo que $c_i$ tenga inversa) Entonces lo que resta es comparar el cociente obtenido con los $c_i$ y sus inversas, y al encontrar una se podrá determinar el *i* y por lo tanto el valor de uno de los bits de *d*. De esta forma, al igual que antes, se pueden obtener los suficientes bits de *d* para realizar un ataque por fuerza bruta. De hecho se puede ver por conteo probabilístico que si se realizan $t \log t$ testeos fallidos se tiene una probabilidad mayor que ½ de obtener todos los bits de *d*.

El tercer tipo de ataque que consideraremos es más factible de ser implementado en la práctica pues requiere de menos asunciones y simplificaciones que los anteriores. Por otro lado sólo funciona cuando las exponenciaciones de RSA están implementadas mediante el uso del teorema chino del resto. El uso de este teorema se traduce en el hecho de que, cuando se desea desencriptar un mensaje *c* dado, se calcula $c^d \bmod p$ y $c^d \bmod q$ y luego se combinan ambos resultados para obtener $c^d \bmod n$.

En este ataque [10.1] lo que se intenta lograr es provocar un error de tal manera que el cálculo del mensaje encriptado sea correcto módulo *p*, pero incorrecto módulo *q* (o viceversa) Dado que la mayor parte del tiempo de cálculo de la encriptación ocurrirá en la realización de estas dos operaciones, sólo se necesita saber en que momento se realiza alguna de ellas y provocar cualquier tipo de error que afecte la salida (no es importante cuantos bits se afecten, sólo interesa que el resultado no sea correcto) Una vez que se logra provocar el error deseado lo que se tiene es que el texto *m'* desencriptado incorrectamente cumple que:

$$m'^e \equiv c \bmod p$$
$$m'^e \neq c \bmod q$$

Por esta razón se tiene que *p* divide a $m'^e - c$ pero que *q* no divide a $m'^e - c$ y dada esta propiedad es fácil obtener el valor de *p* pues el mismo está dado por:

$$\gcd(n, m'^e - c)$$

lo cual puede ser fácilmente obtenido por el algoritmo de Euclides extendido en forma similar el utilizado en la implementación de NTRU (ver 4.3.1.3).

Las medidas posibles que se pueden tomar contra este tipo de ataques son diversas y se pueden clasificar en general en dos tipos básicos a saber: las que se basan en chequear los cálculos para detectar errores y las que se basan en "ocultar la información" o sea agregar algún elemento aleatorio en las operaciones de modo de que, aunque un atacante pueda forzar una falla, los datos que obtengan no le sean útiles por desconocer el elemento aleatorio involucrado en las operaciones. Algunas de estas protecciones son las siguientes (de [10.1]):

- Calcular la salida del algoritmo de desencriptación dos veces y comparar los resultados obtenidos. Sólo en caso de ser idénticos se enviará la salida al exterior. La desventaja básica de este método es que duplica el tiempo de procesamiento, lo cual puede ser inaceptable en muchas aplicaciones.

- En algunas aplicaciones de bajos recursos donde la clave pública $e$ es pequeña puede ser más eficiente volver a encriptar el mensaje en texto plano obtenido ($m'^e$) para ver si coincide con el texto cifrado original en lugar de desencriptar el texto cifrado dos veces y comparar (como se proponía en el método anterior)

- Para algunas aplicaciones de firma digital se puede implementar que la tarjeta elija un string aleatorio y lo concatene al mensaje original antes de firmarlo. O sea que el cálculo que se realiza es $(m \mathbin{+\!\!+} s)^d \bmod n$ (donde ++ es la concatenación de strings) El largo de este string deberá ser conocido de antemano por las dos partes para que quien compruebe la firma pueda saber que parte del mensaje fue la que se generó aleatoriamente. Como el atacante ignora que se está agregando el string, nunca obtendrá resultados útiles al comparar $m$ y $m'$. Este sistema evita los ataques indicados anteriormente pero no permite el uso de funciones de hash para obtener un resumen del mensaje por lo que las firmas quedarán aproximadamente del mismo tamaño que los textos originales firmados.

- En el caso de que la clave pública sea grande y no se desee o no sea factible repetir la desencriptación, se puede utilizar el siguiente método para calcular $c^d \bmod n$:
  Primero la tarjeta genera un número aleatorio $r$ y calcula $r^d \bmod n$. Para lograr mayor eficiencia esto puede hacerse de antemano antes de recibir el valor de $c$.
  Entonces se calcula $(rc)^d \bmod n$ y finalmente $\frac{(rc)^d}{r^d} \bmod n$. Si no se produjo ningún error el cociente calculado será igual a $c^d \bmod n$ pero en caso de haberse producido un error el resultado enviado hacia afuera quedará "enmascarado" por $r$.
  A modo de ejemplo, en el segundo tipo de ataque descrito arriba si, el atacante tiene que $d_i$ es 0 y $d'_i$ es 1 el valor de $\frac{m'}{m}$ será igual a $r^{d'} c_i \bmod n$ y desde que $r$ es desconocido para el atacante este valor no tiene ninguna utilidad para él.

# Ataques a ElGamal

Para definir el criptosistema de ElGamal [8.1] se debe elegir un número primo $p$ y un valor $\alpha$ que tiene que ser un elemento primitivo de $Z_p^*$. Entonces, a partir de estos datos se elige un valor $a$ a partir del cual se define un número $\beta$ que es congruente con $\alpha^a$ módulo $p$. En este caso $a$ será la clave privada y el resto de los valores serán públicos.

Cuando se desea encriptar un mensaje $x$ con este método se debe elegir un número aleatorio $k$ perteneciente a $Z_{p-1}$ y se define la encripción del mensaje $x$ como el par de valores $y_1$ y $y_2$ que se obtienen de la siguiente manera:

$$y_1 \equiv a^k \bmod p$$
$$y_2 \equiv x b^k \bmod p$$

Entonces el valor original se puede obtener mediante el uso de la clave privada $a$ de la siguiente manera:

$$x \equiv y_2 \left(y_1^{\,a}\right)^{-1} \bmod p$$

En el ataque que se describirá a continuación [10.1] se supondrá de nuevo por simplicidad que se logra producir un único error en un bit de la clave privada $a$ durante el cálculo de la desencriptación. Si el bit erróneo es el i-ésimo el mismo se escribirá como $a'_i$ y el valor erróneo de $a$ que se obtiene como resultado de este error se denotará como $a'$.

En este caso el valor que se obtendrá al intentar desencriptar será:

$$T \equiv y_2 \left( {y_1}^{a'} \right)^{-1} \bmod p \equiv x a^{ak} \left( a^{-a'k} \right) \bmod p \equiv x a^{k(a-a')} \bmod p$$

Si definimos entonces:

$$R_i \equiv a^{k2^i} \bmod p \quad \text{para } i = 0, 1, 2, ..., t-1$$

se tiene que:

$$TR_i \equiv x \bmod p, \text{ si } a_i = 0 \text{ (pues en este caso } a - a' = -2^i)$$

$$\frac{T}{R_i} \equiv x \bmod p, \text{ si } a_i = 1 \text{ (pues en este caso } a - a' = 2^i)$$

Por lo tanto el atacante solamente debe computar $TR_i$ y $T/R_i$ para ver si alguno de ellos coincide con $x$ mod $p$ para algún $i = 0, 1, 2, ..., t-1$. Si se encuentra una coincidencia entonces también se descubre un bit de la clave privada $a$.

Para que este ataque tenga éxito es claro que el atacante deberá poder encriptar mensajes arbitrarios, dado que deberá comparar los resultados con el mensaje $x$ a fin de obtener las claves privadas.

Las medidas que se pueden tomar para evitar este tipo de ataque pueden ser las siguientes:

- Al igual que en el caso anterior se puede computar la desencriptación dos veces y chequear que se obtuvieron los mismos valores antes de devolver un resultado.
- Como opción menos costosa se podría hacer que el algoritmo chequeara que cierta relación entre los datos se cumple durante le ejecución del programa con el fin de detectar posibles alteraciones. Este sistema en general no detectará todas las alteraciones por lo que es una alternativa menos segura al hecho de ejecutar la operación dos veces.
- Otra opción podría ser conservar en una variable aparte una copia del valor de la clave privada $a$, de forma de poder comparar el valor que se usó durante los cálculos con el que se tenía almacenado para de esta forma asegurarse de que el mismo era el correcto.

En general para evitar este tipo de ataques por generación de fallas algunas tarjetas implementan protecciones físicas sobre el procesador y la memoria con el fin de aislar a los mismos de radiación electromagnética extraña y cumpliendo como beneficio extra con la función de proteger a los componentes de la tarjeta del polvo u otros elementos físicos dañinos.

# Conclusiones

Se mostró que existen diferentes formas de atacar criptosistemas implementados en el hardware de dispositivos portátiles para obtener la información secreta almacenada en los mismos. Por esto hay que tener en consideración que la seguridad de una implementación de un criptosistema en un dispositivo de este tipo no va a estar dada solamente por el largo de las claves sino que también hay que tener en cuenta muchos otros aspectos que son significativos a la hora de hablar de la seguridad real del sistema.

La importancia de este tipo de ataques ha sido tomada en consideración tanto por la industria como por los ambientes académicos y de hecho ha sido considerada, por ejemplo, por el NIST (*National Institute of Standards and Technology*) el cual realizó testeos de este tipo para seleccionar a los algoritmos candidatos para el AES (*Advanced Encryption Standard*) que suplantó al DES como estándar del gobierno de los Estados Unidos para encriptar información no clasificada [10.5].

Es cierto que para realizar muchos de estos ataques se debe disponer del dispositivo a atacar durante cierta cantidad de tiempo, pero esto no quita que los mismos puedan llegar a ser factibles en muchos casos reales.

En general se puede decir que tanto el constructor del hardware como el implementador de los algoritmos criptográficos deben colaborar en forma cercana para lograr, en conjunto, un dispositivo que permita al usuario tener la seguridad de que sus claves y sus datos estarán protegidos dentro del mismo, aún en casos extremos de robo o pérdida. Por esta razón los dispositivos portátiles seguros no solamente son mucho más difíciles de construir de lo que puede parecer a primera vista, sino que son también mucho más difíciles de programar.

# Apéndice VII: Mediciones

Para todas las mediciones que se muestran a continuación se asumió una distribución normal de los datos obtenidos y se utilizó la distribución $t$ de Student para calcular los intervalos de confianza dado que no se tomaron la suficiente cantidad de medidas para justificar el uso directo de una distribución normal. En todos los casos los intervalos de confianza están calculados con un grado de certeza del 95%

## NTRU

Corridas con autenticación no válida:

| Corridas | Tiempo (segundos) |
|---|---|
| 1 | 16.884 |
| 2 | 16.954 |
| 3 | 17.165 |
| 4 | 16.814 |
| 5 | 16.754 |
| 6 | 16.955 |
| 7 | 17.014 |
| 8 | 16.965 |
| 9 | 17.194 |
| 10 | 17.135 |

**Tabla 14: Tiempos de NTRU con autenticación no válida**

$\overline{X} = 16.983$

$S_{10} = 0.147$

$a = 0.05 \quad \Rightarrow \quad 1 - a = 95\%$

$t_{a/2}(n-1) = 2.262$

$$I = \left[ 16.983 - 2.262 \ \frac{0.147}{\sqrt{10}} \ , \ 16.983 + 2.262 \ \frac{0.147}{\sqrt{10}} \right]$$

$$I = \left[ 16.983 - 0.105 \ , \ 16.983 + 0.105 \right]$$

$$I = \left[ 16.878 \ , \ 17.088 \right] \quad con \, 95\,\%$$

Corridas con autenticación válida:

| Corridas | Tiempo (segundos) |
|---|---|
| 1 | 16.754 |
| 2 | 16.985 |
| 3 | 17.145 |
| 4 | 17.034 |
| 5 | 17.015 |
| 6 | 17.224 |
| 7 | 16.925 |
| 8 | 16.934 |

| | |
|---|---|
| 9 | 16.915 |
| 10 | 16.934 |

**Tabla 15: Tiempos de NTRU con autenticación válida**

$\overline{X} = 16.987$
$S_{10} = 0.130$

$a = 0.05 \quad \Rightarrow \quad 1 - a = 95\%$
$t_{a/2}(n-1) = 2.262$

$$I = \left[ 16.987 - 2.262 \ \frac{0.130}{\sqrt{10}} , \ 16.987 + 2.262 \ \frac{0.130}{\sqrt{10}} \right]$$

$$I = [16.987 - 0.093 , \ 16.987 + 0.093]$$

$$I = [16.894 , \ 17.080] \quad con\ 95\ \%$$

En total:

$\overline{X} = 16.985$
$S_{20} = 0.135$

$a = 0.05 \quad \Rightarrow \quad 1 - a = 95\%$
$t_{a/2}(n-1) = 2.093$

$$I = \left[ 16.985 - 2.093 \ \frac{0.135}{\sqrt{20}} , \ 16.985 + 2.093 \ \frac{0.135}{\sqrt{20}} \right]$$

$$I = [16.985 - 0.063 , \ 16.985 + 0.063]$$

$$I = [16.831 , \ 17.048] \quad con\ 95\ \%$$

# RSA (generando claves)

Corridas con autenticación no válida:

| Corridas | Tiempo (segundos) |
|---|---|
| 1 | 23.634 |
| 2 | 24.545 |
| 3 | 23.524 |
| 4 | 24.245 |
| 5 | 24.225 |
| 6 | 23.934 |
| 7 | 24.014 |
| 8 | 24.726 |
| 9 | 23.965 |
| 10 | 25.226 |

**Tabla 16: Tiempos de RSA (generando claves) con autenticación no válida**

$\overline{X} = 24.204$
$S_{10} = 0.515$

$a = 0.05 \quad \Rightarrow \quad 1-a = 95\%$
$t_{a/2}(n-1) = 2.262$

$$I = \left[ 24.204 - 2.262 \ \frac{0.515}{\sqrt{10}} , \ 24.204 + 2.262 \ \frac{0.515}{\sqrt{10}} \right]$$

$I = [24.204 - 0.368 , \ 24.204 + 0.368]$

$I = [23.836 , \ 24.572] \quad con\ 95\ \%$

Corridas con autenticación válida:

| Corridas | Tiempo (segundos) |
|---|---|
| 1 | 23.745 |
| 2 | 23.443 |
| 3 | 23.674 |
| 4 | 23.684 |
| 5 | 23.785 |
| 6 | 25.246 |
| 7 | 23.974 |
| 8 | 25.157 |
| 9 | 23.674 |
| 10 | 25.837 |

**Tabla 17: Tiempos de RSA (generando claves) con autenticación válida**

$\overline{X} = 24.222$
$S_{10} = 0.850$

$a = 0.05 \quad \Rightarrow \quad 1-a = 95\%$
$t_{a/2}(n-1) = 2.262$

$$I = \left[ 24.222 - 2.262 \ \frac{0.850}{\sqrt{10}} , \ 24.222 + 2.262 \ \frac{0.850}{\sqrt{10}} \right]$$

$I = [24.222 - 0.608 , \ 24.222 + 0.608]$

$I = [23.614 , \ 24.830] \quad con\ 95\ \%$

En total:

$\overline{X} = 24.213$
$S_{20} = 0.684$

$a = 0.05 \quad \Rightarrow \quad 1-a = 95\%$
$t_{a/2}(n-1) = 2.093$

$$I = \left[ 24.213 - 2.093 \frac{0.684}{\sqrt{20}},\ 24.213 + 2.093 \frac{0.684}{\sqrt{20}} \right]$$

$$I = [24.213 - 0.320,\ 24.213 + 0.320]$$

$$I = [23.893,\ 24.533] \quad con\ 95\%$$

# RSA (sin generar claves)

Corridas con autenticación no válida:

| Corridas | Tiempo (segundos) |
|---|---|
| 1 | 12.148 |
| 2 | 12.247 |
| 3 | 12.218 |
| 4 | 12.227 |
| 5 | 12.258 |
| 6 | 12.238 |
| 7 | 12.317 |
| 8 | 12.258 |
| 9 | 12.228 |
| 10 | 12.237 |

**Tabla 18: Tiempos de RSA (sin generar claves) con autenticación no válida**

$$\overline{X} = 12.238$$
$$S_{10} = 0.042$$

$$a = 0.05 \quad \Rightarrow \quad 1 - a = 95\%$$
$$t_{a/2}(n-1) = 2.262$$

$$I = \left[ 12.238 - 2.262 \frac{0.042}{\sqrt{10}},\ 12.238 + 2.262 \frac{0.042}{\sqrt{10}} \right]$$

$$I = [12.238 - 0.030,\ 12.238 + 0.030]$$

$$I = [12.208,\ 12.268] \quad con\ 95\%$$

Corridas con autenticación válida:

| Corridas | Tiempo (segundos) |
|---|---|
| 1 | 13.159 |
| 2 | 13.229 |
| 3 | 13.249 |
| 4 | 13.149 |
| 5 | 13.249 |
| 6 | 13.229 |
| 7 | 13.239 |
| 8 | 13.229 |
| 9 | 13.159 |

| | |
|---|---|
| 10 | 13.229 |

**Tabla 19: Tiempos de RSA (sin generar claves) con autenticación válida**

$\overline{X} = 13.212$
$S_{10} = 0.040$

$a = 0.05 \quad \Rightarrow \quad 1-a = 95\%$
$t_{a/2}(n-1) = 2.262$

$$I = \left[ 13.212 - 2.262 \ \frac{0.040}{\sqrt{10}} , \ 13.212 + 2.262 \ \frac{0.040}{\sqrt{10}} \right]$$

$$I = \left[ 13.212 - 0.029 , \ 13.212 + 0.029 \right]$$

$$I = \left[ 13.183 , \ 13.241 \right] \quad con \ 95\%$$

En total:

$\overline{X} = 12.725$
$S_{20} = 0.501$

$a = 0.05 \quad \Rightarrow \quad 1-a = 95\%$
$t_{a/2}(n-1) = 2.093$

$$I = \left[ 12.725 - 2.093 \ \frac{0.501}{\sqrt{20}} , \ 12.725 + 2.093 \ \frac{0.501}{\sqrt{20}} \right]$$

$$I = \left[ 12.725 - 0.234 , \ 12.725 + 0.234 \right]$$

$$I = \left[ 12.491 , \ 12.959 \right] \quad con \ 95\%$$

# RPK

Corridas con autenticación no válida:

| Corridas | Tiempo (segundos) |
|---|---|
| 1 | 62.841 |
| 2 | 64.072 |
| 3 | 63.712 |
| 4 | 65.234 |
| 5 | 64.998 |
| 6 | 64.423 |
| 7 | 63.675 |
| 8 | 64.120 |
| 9 | 62.765 |
| 10 | 65.521 |

**Tabla 20: Tiempos de RPK con autenticación no válida**

$\overline{X} = 64.136$
$S_{10} = 0.937$

$a = 0.05 \quad \Rightarrow \quad 1-a = 95\%$
$t_{a/2}(n-1) = 2.262$

$$I = \left[ 64.136 - 2.262 \ \frac{0.937}{\sqrt{10}}, \ 64.136 + 2.262 \ \frac{0.937}{\sqrt{10}} \right]$$

$$I = \left[ 64.136 - 0.670 \ , \ 64.136 + 0.670 \right]$$

$$I = \left[ 63.466 \ , \ 64.806 \right] \quad con \ 95\%$$

Corridas con autenticación válida:

| Corridas | Tiempo (segundos) |
|---|---|
| 1 | 63.275 |
| 2 | 62.857 |
| 3 | 64.607 |
| 4 | 62.765 |
| 5 | 64.467 |
| 6 | 64.651 |
| 7 | 64.718 |
| 8 | 64.947 |
| 9 | 65.001 |
| 10 | 65.749 |

**Tabla 21: Tiempos de RPK con autenticación válida**

$\overline{X} = 64.143$
$S_{10} = 0.908$

$a = 0.05 \quad \Rightarrow \quad 1-a = 95\%$
$t_{a/2}(n-1) = 2.262$

$$I = \left[ 64.143 - 2.262 \ \frac{0.908}{\sqrt{10}}, \ 64.143 + 2.262 \ \frac{0.908}{\sqrt{10}} \right]$$

$$I = \left[ 64.143 - 0.649 \ , \ 64.143 + 0.649 \right]$$

$$I = \left[ 63.494 \ , \ 64.792 \right] \quad con \ 95\%$$

En total:

$\overline{X} = 64.220$
$S_{20} = 0.945$

$a = 0.05 \quad \Rightarrow \quad 1-a = 95\%$
$t_{a/2}(n-1) = 2.093$

$$I = \left[ 64.220 - 2.093 \ \frac{0.945}{\sqrt{20}},\ 64.220 + 2.093 \ \frac{0.945}{\sqrt{20}} \right]$$

$$I = \left[ 64.220 - 0.442 \ ,\ 64.220 + 0.442 \right]$$

$$I = \left[ 63.778 \ ,\ 64.662 \right] \quad con\, 95\,\%$$

# Apéndice VIII: Glosario

## Avanzar

El término avanzar se refiere a hacer que el generador de mezclas (*mixture generator*) corra un ciclo entero, esto significa que cada uno de los componentes del generador corra su propio ciclo y de esta forma obtener una salida.

## Característica

Si *R* es un anillo arbitrario y existe un entero positivo *n* tal que *nr* = 0 para todo $r \in R$, entonces al menor de dichos enteros *n* se le llama la característica de *R*. Si no existe ningún *n* se dice que *R* tiene característica 0.

## Clausura algebraica

El cuerpo $\overline{F}$ es la clausura algebraica de *F* si $\overline{F}$ es algebraico sobre *F* y cada polinomio $f(x) \in F[x]$ puede ser escrito como un producto de factores lineales en $\overline{F}$ [*x*], o sea que deben existir elementos $a_1, a_2, \ldots, a_n \in \overline{F}$ tales que

$$f(x) = a(x - a_1)(x - a_2)\cdots(x - a_n)$$

donde *a* es el primer coeficiente de *f*.
Por ejemplo, el cuerpo de los número complejos es la clausura algebraica del cuerpo de los reales.

## Cuerpo algebraico

Si *K* es un sub-cuerpo de *F*, se dice que *F* es algebraico sobre *K* si todo elemento $q \in F$ satisface una ecuación del tipo $a_n q^n + \cdots + a_1 q + a_0 = 0$ donde los $a_i \in K$ y no son todos 0.

## Discrete Fourier Transform (DFT)

Debido a que las computadoras trabajan solamente con valores discretos, el cálculo de la Transformada de Fourier *f(t)* requiere valores discretos de *f(t)*, los cuales llamaremos $f_k$. Además una computadora solo puede calcular *F(s)* para valores discretos de *s* lo cual provee valores discretos también, $F_r$. La DFT está dada por:

$$F_r = \sum_{k=0}^{N-1} \left( f_k \exp\left( -ir2pN^{-1}k \right) \right)$$

## ElGamal

Es un criptosistema basado en el problema del logaritmo discreto sobre $Z_p$. Consiste tanto en un algoritmo de encripción como en uno de firma digital [8.1].

## Fast Fourier Transform (FFT)

La FFT es un algoritmo de la DFT (*Discrete Fourier Transform*) desarrollada por Tukey y Cooley en 1965 el cual reduce el número de operaciones necesarias para realizar una multiplicación entre enteros grandes de algo en el entorno de $n^2$ a *n.log(n)*.

Dos enteros grandes *X* e *Y*, de tamaño a lo máximo *n*–1 dígitos, pueden ser escritos de la forma *X=P(B)* e *Y=Q(B)* siendo *B* la base (generalmente *B*=10 o una potencia de 10) y *P* y *Q* dos polinomios,

$$P(z)=\sum_{j=0}^{n-1} x_j z^j, \qquad Q(z)=\sum_{j=0}^{n-1} y_j z^j$$

Si denotamos *R(z)* al polinomio resultado de multiplicar *P(z)* con *Q(z)*, obtenemos que *XY=R(B)* y acomodando posteriormente los coeficientes de *R(z)* llegamos al producto *XY*. Es así que ahora estamos tratando con el problema de multiplicar dos polinomios de grado menor que *n*.
Un polinomio de grado menor que *m* queda definido por su evaluación en *m* puntos distintos. Por lo tanto para obtener el producto *R(z)=P(z)Q(z)*, alcanza con computar *R(w_k)* en *2n* puntos distintos $w_k$, esto es, computar *P(w_k)* y *Q(w_k)*.
La idea de la FFT consiste en elegir como valor de $w_k$ a la raíz compleja de

$$w_k = \exp\left(\frac{2ikp}{2n}\right) = v^k, \qquad v = \exp\left(\frac{2ip}{2n}\right)$$

## GCD

Es el máximo común divisor de dos números según sus siglas en inglés (*Greatest Common Divisor*) Esto significa que es una función que devuelve el mayor número entero que divide a los dos parámetros que se le pasan.

## GF

Iniciales de *Galois Field*; el nombre que reciben los cuerpos finitos en general. Dado que se puede probar que existe un único cuerpo finito de cualquier tamaño del tipo $p^n$ (donde *p* es un primo y $n \geq 1$) cada uno de estos cuerpos se nombra como $GF(p^n)$. Cuando $p = 1$ se tiene que $GF(p) = Z_p$.

## Lattice

Sea *M* una matriz de $n \times n$ siendo $b_1 \ldots b_n$ sus filas. El lattice generado por ( $b_1 \ldots b_n$ ) se nota como $L(M)$ y es el conjunto de todas las combinaciones lineales con coeficientes enteros de los vectores $b_i$. Este conjunto de los $b_i$ se dice que es una *base* del lattice. A la cantidad de elementos de una base se le denomina *dimensión* del lattice.
De esta forma, un lattice puede ser definido también como un subgrupo (aditivo) discreto de un espacio vectorial ( $\Re^n$ en los casos estudiados aquí)

## LLL

Este es un algoritmo creado por Lenstra, Lenstra y Lobász que permite encontrar bases reducidas de un lattice. Se trata de un algoritmo polinomial que, para un lattice de dimensión *n* con enteros de tamaño $O(n \log C)$, requiere $O(n^4 \log C)$ operaciones.

Dado un lattice $L \in \Re^n$ una base reducida $\{b_1, b_2, \ldots, b_n\}$ para el mismo debe cumplir que:

a. Para cada elemento no nulo $x \in L$, $\|b_1\| \leq 2^{(n-1)/2}\|x\|$
b. Y más generalmente, para cualquier conjunto $\{a_1, a_2, \ldots, a_n\}$ de vectores linealmente independientes en *L* se tiene que:

$$\|b_j\| \leq 2^{(n-1)/2} \max(\|a_1\|, \|a_2\|, \ldots \|a_t\|), \text{ para } 1 \leq j \leq t$$

## Norma $L_1$

La norma 1 de un vector $x_1, x_2, \ldots, x_n$ se define como: $\sum_{r=1}^{n} |x_r|$

## Norma $L_\infty$

La norma infinito de un vector $x_1, x_2, \ldots, x_n$ se define como: $\max_i |x_i|$

## Semilla

Valor inicial que utilizan los generadores de números seudo-aleatorios para generar los datos. Para una semilla inicial dada, un generador siempre producirá la misma secuencia de valores y luego de un cierto tiempo (período del generador) volverá a comenzar otra vez con la secuencia. Por esta razón es esencial modificar frecuentemente la semilla de estos generadores.

## Serialización

Mecanismo integrado en el API de Java que permite transformar un objeto en un stream de bits que puede ser almacenado a un archivo o enviado por una red para después ser reconstruido nuevamente conservando todas sus propiedades originales.

## Teorema Chino del Resto (Chinese Remainder Theorem)

Este teorema, proporciona en realidad un método para resolver ciertos sistemas de congruencias en módulo. Suponiendo que $m_1, \ldots m_r$ son enteros positivos primos entre sí (esto es $\gcd(m_i, m_j) = 1$ si $i \neq j$) y suponiendo que $a_i, \ldots, a_r$ son enteros el teorema indica que el sistema de *r* congruencias $x \equiv a_i \pmod{m_i}$ con $(1 \leq i \leq r)$ tiene una única solución módulo $M = m_1 \times \ldots \times m_r$ que se puede hallar como:

$$x = \sum_{i=1}^{r} a_i M_i y_i \mod M$$

donde $M_i = M / m_i$ y $y_i = M_i^{-1} \bmod m_i$ con $(1 \leq i \leq r)$
Por una demostración del mismo ver [1.4] pag. 120

# Apéndice IX: Índice de tablas y figuras

## Tablas



## Figuras

# Apéndice X: Bibliografía


1. General
   1.1. Priit Karu, Jonne Loikkanen. Practical comparison of fast public-key cryptosystems. Tik-110.501 Seminar on Network Security
   1.2. Phong Q. Nguyen, Jacques Stern. Lattice Reduction in Cryptology: An Update. Corrected version of Algorithmic Number Theory – Proceedings of ANTS – IV, Julio 2000
   1.3. A. Menezes, P. Van Oorschot, S. Vanstone. Handbook of Applied Cryptography, CRC Press, 1996
   1.4. D. Stinson, Cryptography: theory and practice, CRC Press, 1997
   1.5. Rudolf Lidl, Harald Niederreiter, Introduction to finite fields and their applications, Revised Edition, Cambridge University Press, 1997
   1.6. Ralph P. Grimaldi, Matemáticas Discreta y Combinatoria – Una introducción con aplicaciones, Tercera Edición, Addison-Wesley Iberoamericana, 1997
   1.7. Eric Weisstein's World of Mathematics. http://mathworld.wolfram.com
   1.8. Steve Petri, An Introduction to Smart Cards, Octubre 1999
   1.9. http://www.gemplus.com
   1.10. Federal Information Processing Standards Publication 140-2, Security Requirements for Cryptographic Modules, National Institute of Standards and Technology, Mayo 2001
   1.11. Philips Semiconductors, P8WE5032 Secure 8-bit Smart Card Controller, Short Form Specification, Revision 1.0, Julio 2000

2. Autenticación (aplicación de referencia)
   2.1. Marilyn Chun, Authentication Mechanisms – Which Is Best?, SANS Institute, Abril 2001
   2.2. Yves Deswarte, Noreddine Abghour, Vincent Nicomette, David Powell. An Internet Authorization Scheme Using Smart-Card-Based Security Kernels

3. GGH
   3.1. Oded Goldreich, Shafi Goldwasser, Shai Halevi. Public-Key Cryptosystems from Lattice Reduction Problems, MIT – Laboratory for Computer Science preprint, Noviembre 1996
   3.2. Phong Nguyen. Cryptanalysis of the Goldreich - Goldwasser - Halevi Cryptosystem from Crypto '97, Advances in Cryptology – Proceedings of CRYPTO '99, Agosto 1999
   3.3. R.J. McEliece, A Public-Key Cryptosystem Based on Algebraic Coding Theory. DSN Progress Report 42-44, Jet Propulsion Laboratory
   3.4. M. Ajtai. Generating hard instances of lattice problems. Electronic Colloquium on Computational Complexity – Report Series 1996
   3.5. Phong Q. Nguyen y Jacques Stern, Lattice Reduction in Cryptology: An Update. École Normale Supérieure, Département d'Informátique
   3.6. C.P. Schnorr, M. Fischlin, H. Koy and A. May, Lattice attacks on GGH Cryptosystems. Rump session of Crypto '97

4. NTRU
   4.1. Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman. NTRU: A Ring-Based Public Key Cryptosystem.
   4.2. Joseph H. Silverman. NTRU Cryptosystems Technical Report.
   Report #010: High-Speed Multiplication of (Truncated) Polynomials. Enero 1999
   4.3. Joseph H. Silverman. NTRU Cryptosystems Technical Report.
   Report #013: Almost Inverses and Fast NTRU Key Creation. Marzo 1999
   4.4. Joseph H. Silverman. NTRU Cryptosystems Technical Report.
   Report #001: Commutative NTRU: Pseudo-code Implementation. Marzo 1997
   4.5. Joseph H. Silverman. NTRU Cryptosystems Technical Report.
   Report #009: Invertibility in Truncated Polynomial Rings. Octubre 1998.
   4.6. Joseph H. Silverman. NTRU Cryptosystems Technical Report.
   Report #004: A Meet-In-The-Middle Attack on an NTRU Private Key. Julio 1997.
   4.7. Joseph H. Silverman. NTRU Cryptosystems Technical Report.
   Report #005: Hard Problems and Backdoors of NTRU and Other PKCS's. Octubre 1997.
   4.8. Joseph H. Silverman. NTRU Cryptosystems Technical Report.
   Report #006: Implementation Notes for NTRU PKC multiple transmissions. Mayo 1998.

4.9. Joseph H. Silverman. NTRU Cryptosystems Technical Report. Report #007: Plain Text Awareness and the NTRU PKCS. Junio 2000.
4.10. Joseph H. Silverman. NTRU Cryptosystems Technical Report. Report #011: Wraps, Gaps and Lattice Constants. Marzo 2001.
4.11. Joseph H. Silverman. NTRU Cryptosystems Technical Report. Report #012: Estimated Breaking Times For NTRU Lattices. Marzo 1999.
4.12. Joseph H. Silverman. NTRU Cryptosystems Technical Report. Report #013: Dimension-Reduced Lattices, Zero-Forced Lattices and the NTRU Public Key Cryptosystem. Marzo 1999.
4.13. Joseph H. Silverman. NTRU Cryptosystems Technical Report. Report #015: Reaction Attacks Against NTRU Public Key Cryptosystem. Junio 2000.
4.14. Jeffrey Hoffstein and Joseph H. Silverman. NTRU Cryptosystems Technical Report. Report #016: Protecting NTRU Against Chosen Ciphertext and Reaction Attacks. Junio 2000.
4.15. http://www.ntru.com
4.16. Don Coppersmith, "Fast Evaluation of Logarithms in Fields of Characteristic Two", IEEE Transactions on Information Theory, Julio 1984

5. RPK
5.1. William M. Raike. The RPK Public-Key Cryptographic System. Technical Summary. http://crypto.swdev.co.nz
5.2. William M. Raike. Detailed Supplemental Technical Description of the RPK Public-Key Cryptographic System. http://crypto.swdev.co.nz
5.3. A. Romeo, G. Romolotti, M. Mattavelli, D. Mlynek. Cryptosystem Architectures for Very High Throughput Multimedia Encryption: The RPK Solution.
5.4. http://www.rpk.com

6. RSA
6.1. Dan boneh, Twenty Years of Attacks on the RSA Cryptosystem.
6.2. R.L. Rivest, A. Shamir y L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Commun. Of the ACM*, 21:120-126, 1978.
6.3. http://www.rsasecurity.com/rsalabs/challenges/factoring/rsa155.html
6.4. M. Wiener, Cryptanalysis of short RSA secret exponents. IEEE Transactions on Information Theory, 36:553-558, 1990
6.5. http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/index.html

7. ECC
7.1. Julio López, Ricardo Dahab, An Overview of Elliptic Curve Cryptography. Institute of Computing, State University of Campinas, Mayo 2000
7.2. Johan Borst, Public Key Cryptosystems using Elliptic Curves. Technische Universiteit Eindhoven, Faculteit Wiskunde en Informatica
7.3. N. Koblitz, Elliptic curve cryptosystems. *Mathematics of computation*, 48, pp.203-209, 1987
7.4. V. Miller, Uses of elliptic curves in cryptography. *Advances in Cryptology: proceedings of Crypto' 85*, LNCS pp. 417-426, New York, Springer Verlag, 1986
7.5. A. Menezes y S. Vanstone, Elliptic Curve Systems and Their Implementation. Journal of Cryptology 6, 1993, pp. 206-224
7.6. Don Johnson, Alfred Menezes. The Elliptic Digital Signature Algorithm (ECDSA). Febrero 2000.
7.7. http://www.secg.org

8. ElGamal
8.1. T. ElGamal, A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms, IEEE Transactions on Information Theory, vol. IT-31, 1985, pp. 469-472

9. JavaCard
9.1. Daniel Perovich, Leonardo Rodríguez, Martín Varela. Programación de JavaCards, Proyecto de taller V (2000)
9.2. Sun Microsystems, Inc. Java Card 2.1.1 Runtime Environment (JCRE) Specification. Revision 1.0, Mayo 2000

9.3. Sun Microsystems, Inc. Java Card 2.1.1 Virtual Machine Specification. Revision 1.0, Mayo 2000
9.4. Sun Microsystems, Inc. Java Card 2.1.1 Application Programming Interface. Revision 1.0, Mayo 2000
9.5. Sun Microsystems, Inc. Java Card 2.1.2 Development Kit User's Guide. Revision 1.0, Abril 2001
9.6. Gemplus. GemXpresso RAD III Getting Started, version 3.1. Mayo 2001
9.7. Gemplus. GemXpresso RAD III User Guide, version 3.1. Mayo 2001
9.8. Gemplus. GemXpresso 211 PK Card, Reference Manual Version 1.0
9.9. Bruce Eckel. Thinking in Java. Prentice Hall, 1998
9.10. Pierre Boury y Nabil Elkadhi. Static Analysis of Java Cryptographic Applets.
9.11. Gemplus. Understanding the Fundamentals of Smart Card Enabled Security for Web and E-Mail & Setting up GemSAFE Applications. Junio 1998.
9.12. http://java.sun.com/products/javacard

10. Ataques
10.1. F. Bao, R.H. Deng, Y. Han, A. Jeng, A.D. Narasimahalu, T. Nagir. Breaking Public Key Cryptosistems on Tamper Resistant Devices in the Presence of Transient Faults. Institute of System Science, National University of Singapore
10.2. Ross Anderson, Markus Kuhn. Low Cost Attacks on Tamper Resistant Devices
10.3. Lauri Karppinen. Attacks related to the smart card used in electronic payment and cash cards. Laboratory of Telecommunications Software and Multimedia, Helsinki University of Technology
10.4. Erwin Hess, Norbert Janssen, Bernd Meyer, Torsten Schültze. Information Leakage Attacks Against Smart Card Implementations of Cryptographic Algorithms and Countermeasures, A Survey
10.5. James Nechvatal, Elaine Barker, Donna Dodson, Morris Dworkin, James Foti, Edward Roback. Status Report on the First Round of the Advanced Encryption Standard. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology
10.6. R. Merckling y A. Anderson, Smart Card Introduction, Marzo 1994
10.7. DG. Abraham, GM. Dolan, GP. Double y JV. Stevens, Transaction Security System de IBM Systems Journal vol. 30 no. 2 (1991), pp. 206-229
10.8. Oliver Kömmerling, Marcus G. Kuhn, Design Principles for Tamper-Resistant Smart cards Processors

11. OCF y VOP
11.1. http://www.opencard.org.
11.2. Open Card Framework, General Information Web Document. Second Edition, Octubre 1998
11.3. http://www.globalplatform.org
11.4. Open Platform, Card Specification, Versión 2.0.1