Universidad Nacional Autónoma de México

Facultad de Ciencias

Familias de Funciones Hash

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
MATEMÁTICO

PRESENTA:
LEOPOLDO GARCÍA VARGAS

DIRECTOR DE TESIS:
DR. OCTAVIO PÁEZ OSUNA

2013

Índice

Capítulo 1

Introducción.

1.1 Motivación

En los últimos 20 años el mundo en el que vivimos ha cambiado radicalmente, principalmente en las áreas de Ciencia y Tecnología. Estos rápidos avances han hecho que nuestra especie dependa de los canales de transferencia de información casi de una forma vital. La llegada del Internet revolucionó la forma en la que interactuamos y, mas importante aún como intercambiábamos nuestros datos. Estos cambios no sólo trajeron ventajas, también abrieron las posibilidades de distintas formas de ataques electrónicos.

Cada día se llevan a cabo millones de transferencias de datos atravéś del Internet, el cual se ha convertido en un canal tan "eficiente" que básicamente toda compañia o institución tienen su propia página web o al menos ejecuta parte de sus transferencias atravéś de él. Debido a que muchos de estos intercambios de información representan intercambios de dinero en el mundo real, esto ha llevado a muchas empresas e instituciones han decidido invertir millones en métodos para mejorar sus canales de transferencia. La necesidad de transferencias seguras e íntegras de información se ha vuelto un tema de amplia discución he inversión en nuestros días.

1.2 El fin de la Privacidad.

Si bien es increíble hacerse consciente de que 2 millones y medio de habitantes del planeta estamos conectados a través del Internet (Kovacs, 2011). Es más impresionante aún si pensamos que más del 30 por ciento de la población actual tiene el potencial de acceder en cualquier momento a dicha red para aprender, crear y compartir. El tiempo que pasamos conectados a Internet ha incrementado drásticamente en los últimos años, pues, de hecho un estudio reciente (Kaiser Family Foundation, 2010) nos dice que la generación "joven" (niños y adolescentes) pasa en promedio 8 horas diarias conectada a Internet. Estas cifras son alarmantes ya que casi una tercera parte de la población gasta la tercera parte de sus días conectada. Y no obstante, de la misma forma en que el Internet ha abierto el mundo para cada uno de nosotros también nos ha abierto a cada uno al mundo, siendo que más aún; parte del precio que tenemos que pagar por toda esta conectividad es nuestra *privacidad*. Hoy en día nos gustaría mucho pensar que el Internet es un lugar privado y seguro, pero no lo es. Cada día, con cada toque en la pantalla o cada click en el mouse, vamos dejando como Hansel y Grettel "migajas de pan": rastros de nuestra información personal en todos los lugares que visitamos del bosque digital. Estas migajas de pan que vamos dejando a nuestro paso son datos tan importantes como: nuestra fecha de nacimiento, lugares de residencia, intereses y preferencias, relaciones y conexiones personales, historias financieras y más.

Es claro que algunos de estos sitios pueden utilizar nuestra información personal con nuestro consentimiento para propósitos que en última instancia resultan benéficos para el usuario como: dar sugerencias de música y videos con base en nuestros gustos, escritura predictiva, datos GPS más precisos, etc. Es ventajoso que los sitios más visitados entiendan "hábitos" humanos para mejorar la experiencia del usuario, y sin embargo es preocupante que otros sitios o servidores que no tendrían porqué conocer los detalles de nuestra información personal, no sólo la conocen, si no que disponen de ella a su antojo y sin nuestro consentimiento.

Hay un nuevo fenómeno en el Internet llamado "Behavioral Tracking" (Kovacs, 2011), que es una técnica de segmentación basada en el comportamiento o navegación de los usuarios. Consiste en el seguimiento de los hábitos de navegación web del usuario con el fin de proporcionar exacta-

mente la información que quiere ver el usuario. Para ello, utiliza sistemas avanzados que permiten recopilar la actividad o navegación de los usuarios, y crear un perfil a partir de qué contenidos leen, cuánto tiempo se pasan en ellos, con qué frecuencia los consultan, y qué palabras clave buscan, entre otros.

En la actualidad existe una industria millonaria que se dedica a rastrear a la gente a través del bosque digital para crear perfiles, y una vez que estas compañías poseen la información que desean pueden hacer lo que quieran con ella, y utilizarla con propósitos de marketing por ejemplo. Las ganancias de las compañias que llevan a cabo el "behavioral tracking" supera los 39 billones de doláres anuales. Hoy en día ésta es todavía un área muy poco regulada y prácticamente sin vigilancia.

Las imágenes al final del capítulo son fotos de mi pantalla durante una mañana común de trabajo utilizando un Add-on para Mozilla Firefox llamado Collusion, el cual tiene como propósito elaborar una gráfica estilo arbol basada en los sitios que conocen la información del usuario que está navegando. En esta gráfica los nodos con borde azul representan los sitios que el usuario visita y los nodos con borde gris representan los sitios que conocen la información del usuario sin haber sido visitados. Por último las aristas representan conexiones entre sitios y las aristas dirigidas representan intercambios de información entre sitios. En general este servicio de Firefox nos permite saber qué sitios están llevando a cabo un rastreo de comportamiento sobre nosotros.

Es alarmante ver la cantidad de nodos grises que surgen tras unas cuantas horas de utilizar el Internet, debido a que estos nodos representan los sitios que conocen la información del usuario sin haber sido visitados. Cuando visitamos el bosque digital bastan unos clicks en el mouse o unos toques en la pantalla para compartir nuestra información privada con muchos sitios los cuales no son todos bien intencionados. Después de analizar las gráficas nos queda muy claro que el Internet no es un lugar seguro para nuestra información personal, pues más aún, todos los que lo usamos estamos siendo espiados a detalle por empresas multibillonarias, las cuales en su mayoría tienen objetivos mercantiles para nuestra información.
La necesidad de protocolos efectivos para fines de verificación de identidad es inminente, necesitamos saber con quiénes nos conectamos, y estar seguros de que realmente sea la entidad que dice ser. Debemos también acotar la

información personal que poseen los sitios o entidades con las que nos conectamos, al grado de que sólo sepan lo que es verdaderamente necesario saber de los usuarios. La verificación de identidad es uno de los eslabones centrales de los esquemas criptográficos que nos garantizan que las transferencias serán eficientes y seguras. Antes de llevar a cabo cualquier transferencia de datos es necesario verificar que el destinatario (o remitente en su caso) sea realmente quien dice ser. Para llevar a cabo esto hoy en día se usan variantes de DSA usando un previo intercambio de llaves (Diffie-Hellman por ejemplo).
El tema principal de esta tesis es la verificación de identidad, para lo cual construiremos un esquema de firmas digitales que estará basado en "Universal Hashing" (Carter-Wegman, Bibliografía) para calcular las firmas digitales y su verificación. Una vez desarrollado el esquema de Firmas Digitales lo implementaremos en los dispositivos Apple iPhone y iPad con iOS.

Figura 1.1: Día típico de trabajo 9:00 am. Imagen capturada usando FireFox Collusion Add-on

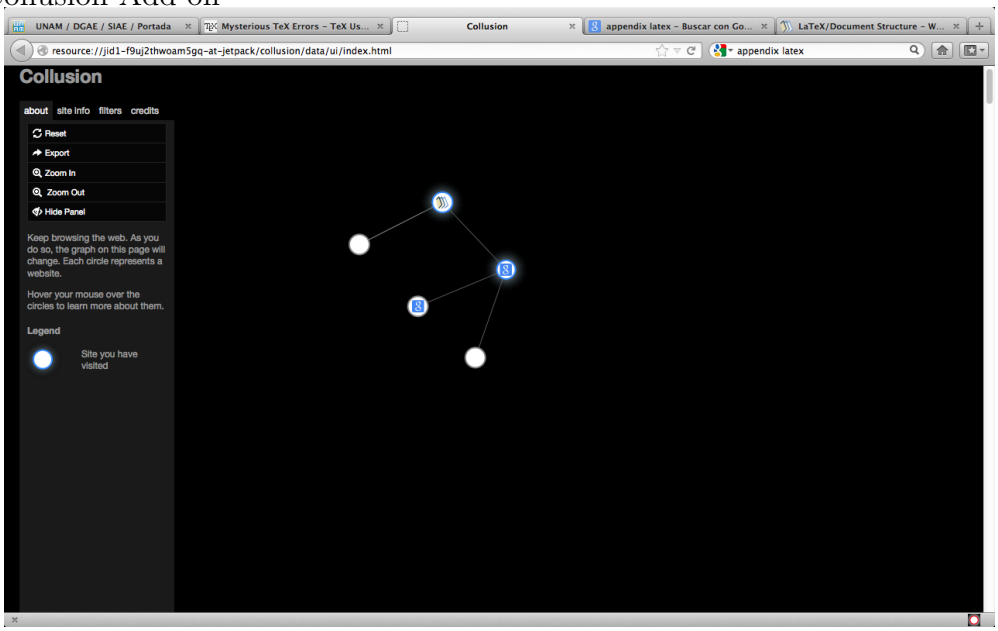


Figura 1.2: Día típico de trabajo 11:30 am. Imagen capturada usando FireFox Collusion Add-on

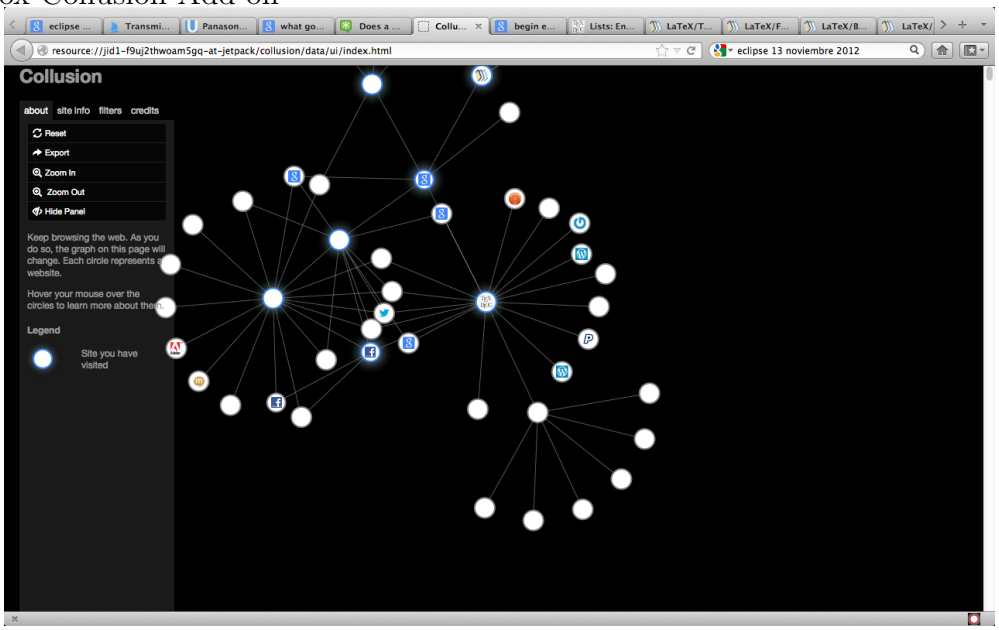
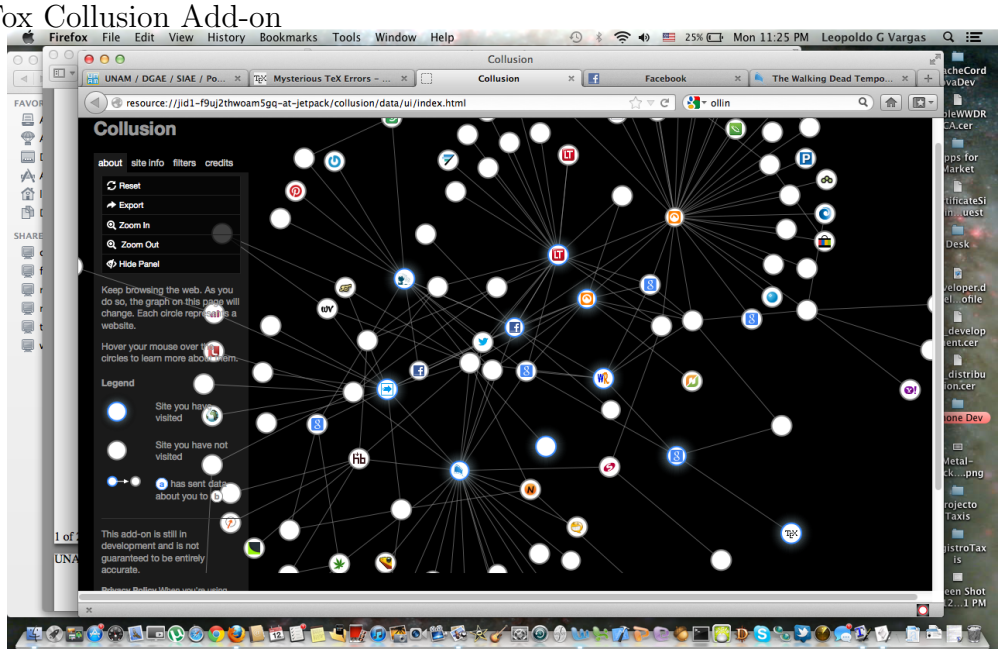


Figura 1.3: Día típico de trabajo 11:30 pm. Imagen capturada usando FireFox Collusion Add-on



Capítulo 2

Bases de Criptografía.

La Criptografiía es la ciencia encargada de diseñar técnicas matemáticas que llamaremos esquemas o dispositivos, capaces de transformar mensajes legibles a mensajes cifrados, e inversamente mensajes cifrados a mensajes legibles, de tal manera que las transformaciones cifrar y descifrar sólo pueden ser factibles con el conocimiento de una o más llaves. Otro de sus objetivos principales es diseñar esquemas que nos permiten intercambiar o generar llaves secretas de una forma segura y efectiva. También nos permite hacer uso de métodos de comprobación de identidad y validación de información respecto a un usuario en específico o respecto a una autoridad central (CA). Estos esquemas son intrínsecamente distintos en términos de estructura y complejidad, la elección de uno o más de ellos para usos prácticos depende de los requerimientos de seguridad y de el contexto bajo el cual se esté implementando el esquema. El punto clave para diseñar sistemas Criptográficos eficientes es hacer que los esquemas de seguridad trabajen en conjunto para crear sistemas confiables de intercambio de información.
El respaldo abstracto de los sistemas criptográficos que usamos hoy en día abarca diversas áreas de las matemáticas. En su clasificación dentro de las ciencias, la criptografía proviene de una rama de las matemáticas, que fue iniciada por el matemático Claude Elwood Shannon en 1948, denominada: Teoría de la Información. Esta rama de las ciencias se divide en: Teoría de Códigos y en Criptología. Y a su vez la Criptología consta de dos áreas: Criptoanálisis y Criptografía.
La palabra Criptografía tiene su origen en el Griego: kryptos que significa oculto, escondido, secreto; y graphos que significa escribir, grabar. Por lo tanto, criptografía significa escritura secreta o escritura oculta.

2.1 Modelo básico de comunicación.

Imaginémonos la siguiente situación; supongamos que dos entidades A y B ajenas y físicamente remotas desean intercambiar información a través de un canal de transferencia no seguro. Con no seguro en este contexto me referiré a un canal de acceso público a través del cual uno o más espías o atacantes están al pendiente de las transferencias que atraviesan el canal de transferencia. Supongamos que todos los intercambios de información están pasando en presencia de una adversario C, cuyo objetivo es vencer y tener acceso a cualquier protocolo de seguridad previamente establecido por las entidades A y B.

Veamos unos cuantos ejemplos:
- A y B podrían ser dos personas que están tratando de comunicarse a través de una red de telefonía celular y, C podría ser una tercera persona externa que esta tratando de escuchar lo que se dice en las llamadas.
- Otro ejemplo (más cercano al contexto de esta tesis) es que A podría ser el explorador de Internet (Safari, Firefox, etc.) de un individuo A* que está tratando de adquirir un producto através de Internet en una tienda electrónica B representado por un sitio Web B*. En este ejemplo el adver-

sario C podría tratar de leer el Tráfico electrónico de A a B con el objetivo de robar la información bancaria de A*, o podría tratar de hacerse pasar por A* o B* a fin de obtener los datos que desea.
- Otro ejemplo es el caso en el que A envía un correo electrónico a B através del Internet. En este caso C podría tratar de leer el contenido de los mensajes, modificar partes de los mismos o hacerse pasar por A o B para mandar sus propios mensajes y así obtener acceso a la información que desea robar.
- Finalmente consideremos el caso en el cual A es una tarjeta de banco inteligente que está en proceso de validar la identidad de su usuario A* respecto al servidor (Autoridad Central) de un banco. En este ejemplo C podría intentar interpretar el tráfico electrónico entre A y B con el objetivo de conocer la información de la cuenta de A*, o podría también hacerse pasar por A* respecto el CA para obtener dinero o crédito de la cuenta de A*.

En los previos ejemplos debe quedar más que claro que A y B no necesariamente son personas; pueden ser: una computadora, una tarjeta de banco inteligente o una entidad abstracta de software (Explorador de Internet o VCA (Virtual Central Authority)) actuando en servicio de una compañía, grupo o persona en específico. El canal de transferencia de información puede ser la radio, el teléfono o el Internet .

2.2 Metas de seguridad para nuestro contexto.

Para implementar un sistema de transferencia de información seguro debemos tomar medidas de seguridad específicas para hacer funcionar el sistema lo más eficientemente posible. Los requerimientos de seguridad de un sistema criptográfico están fuertemente influenciados por el contexto bajo el cual se implementan y también por medidas de seguridad preestablecidas y aprobadas por instituciones oficiales. Dichas medidas de seguridad han ido evolucionando a la par del Internet y los protocolos de transferencia de información han ido cambiando. Hoy en día un sistema criptográfico eficiente tiene que cumplir requerimientos de integridad y de seguridad de información. El cumplir con estos requerimientos al implementar un sistema nos asegura que la información que intercambien nuestros clientes va a mantenerse sec-

reta (seguridad) y va a ser a prueba de modificaciones y robo (integridad).

Si analizamos un poco más a fondo las diferentes situaciones de intercambio de datos a través de un canal público, como en los ejemplos de arriba, podemos darnos cuenta de que las medidas de seguridad que tomamos siempre están relacionadas a ataques imaginarios o a experiencia de ataques previos. En otras palabras; los ataques son el motor de avance de los protocolos de transferencia segura e íntegra de información. Los siguientes cinco objetivos de seguridad deben de ser alcanzados por todo sistema criptográfico que vaya a ser considerado *seguro*:

1) Confidencialidad: Aquí entran los protocolos de cifrado de datos con el objetivo de mantener la información confidencial secreta; i.e. los mensajes intercambiados entre A y B sólo deben ser legibles para ellos mismos. Estos protocolos deben ser tan efectivos que aunque C obtenga la información, no debe de ser capaz de interpretarla.
2) Integridad de la información: Asegurar que los datos que están siendo enviados y recibidos no estén siendo alterados por algún atacante o espía C. En este caso B debería de poder saber cuando C está modificando los datos enviados por A.
3) Verificación del origen de la información: Corroborar el origen de la información recibida. B debe ser capaz de confirmar que los datos supuestamente enviados por A, realmente fueron del todo creados por A.
4) Verificación de identidad: Corroborar la identidad de cualquier entidad con la que se interactúe y se envíe información. B debe de poder estar seguro de la identidad de cualquier otra entidad con la que se comunique.
5) No-repudio de información: Prevenir que alguna entidad desconozca acuerdos y acciones previas.
Recordemos que el contexto de esta tesis es el de Firmas digitales, por lo tanto solamente nos enfocaremos en cumplir en su totalidad los últimos 3 objetivos. Esto quedará más claro en el capítulo tres.

2.3 Ataques y espías.

Con la palabra ataque en este contexto nos referiremos a cualquier acción llevada a cabo por un atacante que compromete la seguridad de la información privada. Dividiremos los ataques en cuatro categorías principales:

1) Interrupción: Este tipo de ataques se centra en la interrupción de la comunicación entre las entidades que intentan establecerla. En este caso el atacante no accede a los datos secretos, pero sí logra interrumpir la comunicación efectiva entre las entidades.
2) Intercepción: El atacante centra sus esfuerzos en interceptar la información que intercambian las entidades. Suponiendo claro que el atacante tiene acceso al canal de información que está siendo utilizado. Tercer ejemplo de la sección 2.1
3) Modificación: Este es un ataque a la integridad de la información. Este tipo de ataques van de la mano con los ataques de intercepción, pero la diferencia es que el atacante trata de modificar la información que intercepta con distintos objetivos. Últimos dos ejemplos sección 2.1
4) Fabricación: Éste es un ataque a la identidad de la entidad que genera la información . El atacante tratará de hacerse pasar por alguna de las entidades para obtener la información que se desea. Últimos dos ejemplos sección 2.1.

Todos los ataques posibles caben en las cuatro categorías anteriores o son una mezcla de más de una de ellas.

Es importante considerar al atacante o espía C lo más capacitado posible, esto con el objetivo de preveer amenazas y ataques realistas sobre los protocolos de seguridad establecidos entre A y B. Más aún daremos por hecho que C tiene acceso directo a la información (puede leer los textos cifrados) y puede alterar los datos intercambiados por el canal de información. Supondremos también que C tiene suficientes recursos computacionales para llevar acabo criptoanálisis y ataques eficientes. Finalmente le daremos a C también pleno conocimiento de los protocolos de comunicación y mecanismos criptográficos que están siendo utilizados , exceptuando evidentemente las llaves secretas utilizadas. Nuestra meta es diseñar mecanismos y protocolos que sean eficientes incluso en la presencia de adversarios tan poderosos como C. También es posible dividir los ataques en Pasivos y Activos. Los ataques pasivos se centran en obtener y analizar la información que está siendo intercambiada. Este tipo de ataques son muy poco evidentes ya que no generan cambios en los canales de información ni modifican la información enviada, y, por lo mismo, son más tardados de llevar acabo por el atacante y son sumamente difíciles de detectar por la entidades que se están comunicando. Por otro

lado, en los ataques activos como su nombre lo indica, involucran algún tipo de modificación en el flujo de datos o la creación de datos o identidades falsas.

Capítulo 3

Criptografía de Llave Pública.

La criptografía de llave pública es el conjunto de algoritmos criptográficos en los cuales las llaves utilizadas como parámetros para ejecutarlos son tanto de dominio público como de dominio privado. Los parámetros públicos y privados están relacionados entre sí de tal forma que las llaves de dominio publico están generadas a partir de las llaves privadas y en la mayoría de los casos de un parámetro público inicial. El respaldo matemático de los Esquemas de Llave Pública toca varias áreas de las matemáticas, principalmente la Teoría de Números y el Algebra Abstracta. No importa cuál sea el acercamiento matemático en cuestión para la implementación del esquema, el objetivo principal es hacer computacionalmente imposible para un adversario determinar los parámetros privados, únicamente conociendo los públicos.

Para establecer un esquema de llave pública las entidades que van a intercambiar información únicamente requieren intercambiar llaves que son totalmente de dominio público. Cada una de las entidades escoge una pareja (e, d) que consiste de una llave pública e y una llave privada d la cual es mantenida en secreto por cada una de las entidades.
Las entidades que desean intercambiar información entonces harán uso de distintas combinaciones de las llaves públicas y privadas junto con los protocolos matemáticos adecuados para alcanzar sus metas de seguridad.
Las aplicaciones de la criptografía de llave pública son muy amplias y van desde generar llaves seguras, cifrado y descifrado de información, firmas digitales y combinaciones de las anteriores. Las dos principales ramas de la criptografía de llave pública son:

-Cifrado de llave pública: un mensaje cifrado con la clave pública de un destinatario no puede ser descifrado por nadie, excepto por un poseedor de la clave privada correspondiente, presumiblemente, éste será el propietario de esa clave y la persona asociada con la clave pública utilizada. Se utiliza para confidencialidad.

-Firmas digitales: un mensaje firmado con la clave privada del remitente puede ser verificado por cualquier persona que tenga acceso a la clave pública, lo que demuestra que el remitente tenía acceso tanto a la clave privada (y por lo tanto, es probable que sea la persona asociada con la clave pública utilizada) y a la parte del mensaje que no se ha manipulado. Se utiliza para Verificación de Identidad.

Más adelante en el capítulo veremos más a detalle estas dos principales áreas y daremos unos ejemplos con algunos algoritmos para dar una idea más cercana a la esencia de estos tipos de esquemas criptográficos. De esta forma los esquemas de llave pública nos proporcionan soluciones muy elegantes para los problemas principales de los esquemas de llave-simétrica (Apéndice B), los cuales son: distribución de llaves, manejo de llaves y servicios de no-repudio de identidad. Es importante destacar que este tipo de esquemas eliminan la necesidad de crear un canal privado para el intercambio de llaves, y aún así implementan una infraestructura de llave publica (PKI) para distribuir y manejar llaves públicas puede ser un reto muy grande al ser llevado a la práctica. Cabe destacar también que los cálculos computacionales en un sistema de llave pública son considerablemente más lentos que la de sus contrapartes (esto debido a que la llave para ejecutar los algoritmos no es la misma). Por este motivo es más seguro y eficiente hacer uso de esquemas híbridos para de esta forma sacarle provecho a las mejores cualidades de cada uno de los enfoques.

3.1 Respaldo Abstracto

En un esquema de criptografía de llave pública las entidades escogen un par de llaves de tal forma que obtener las llaves privadas utilizando únicamente la llave pública es equivalente a resolver un problema computacional que se cree imposible de resolver en tiempo computacionalmente finito. A continuación describiremos de una forma breve los tres problemas matemáticos que

conforman la estructura abstracta de los esquemas de llave pública.

3.1.1 El problema de logaritmo discreto PLD.

La seguridad y eficiencia de esquemas de cifrado y generación de firmas ElGamal y sus variantes como Digital Signature Algorithm (DSA) están basados en la dificultad de resolver el Problema de Logaritmo Discreto (PLD). El cual es el análogo a los logaritmos comunes que conocemos en nuestras clases de calculo sólo que en este contexto estaremos trabajando con elementos en un Grupo cíclico finito G de orden n y generador α. Para una aproximación más concreta podemos imaginar que G es el grupo $\mathbb{Z}_p^*$ de orden $p-1$, donde la operación es simplemente la multiplicación modulo p. Veamos un poco más a fondo el contexto abstracto;

Definición 3.1.1. *Sea $(G,*)$ un grupo cíclico finito con n elementos. Sea α un generador de G, y $\beta \in G$. El logaritmo discreto de β a la base α, denotado $log_\alpha \beta$, es el único entero x tal que $\beta = \alpha^x$, con $0 \leq x \leq n-1$.*

Los grupos de mayor interés en la criptografía son el grupo multiplicativo $\mathbb{F}_q^*$ de el campo finito $\mathbb{F}_q$, incluyendo los casos particulares del grupo multiplicativo $\mathbb{Z}_p^*$ de el campo finito de enteros módulo p un primo. Finalmente el grupo multiplicativo $\mathbb{F}_{2^m}^*$ de el campo finito $\mathbb{F}_{2^m}$ de característica dos, los cuales serán de vital importancia en este proyecto ya que tanto el esquema de firmas como el protocolo de llaves que implementaremos están basados en estos campos.

Definición 3.1.2. *El problema de logaritmo discreto (PLD) es el siguiente: dado un primo p, un generador α de $\mathbb{Z}_p^*$ y $\beta \in \mathbb{Z}_p^*$, encontrar un entero x, $0 \leq x \leq p-2$, tal que $\beta \equiv \alpha^x (mod\ p)$.*

Definición 3.1.3. *El problema de logaritmo discreto generalizado (PLDG) es el siguiente: dado un grupo cíclico finito G de orden n, un generador α de G y un elemento $\beta \in G$, encontrar el entero x, $0 \leq x \leq n-1$, tal que $\beta = \alpha^x$.*

¿Es posible determinar la x que fué usada para generar $\beta = \alpha^x \in G$? Es imposible determinarla en un tiempo computacionalmente razonable, mucho menos si la cardinalidad de G es grande. Éste es el Problema de Logaritmo Discreto y ésta es la base abstracta de algunos sistemas de llave pública.
El primer sistema basado en el problema de logaritmo discreto que apareció en la historia de la criptografa fue concebido por Witfeld Diffie y Martin Hellman en 1976. Su objetivo era cubrir las debilidades encontradas en los sistemas de llave simétrica, principalmente el hacer el intercambio de llaves de una forma más segura y eficiente. En 1984 Taher Elgammal aportó un sistema de cifrado de llave pública que lleva su nombre, el cual está basado en el trabajo previo de Diffie y de Hellman.

El problema de Diffie-Hellman

El problema Diffie-Hellman está íntimamente relacionado al problema de logaritmo discreto (PLD). Es importante para la criptografía de llave pública debido a que su aparente insolubilidad constituye la base de seguridad de muchos esquemas criptográficos como en el Cifrado de Llave Pública ELGamal.

Definición 3.1.4. *El problema Diffie-Hellman (PDH) es el siguiente: dado un primo p, un generador α de $\mathbb{Z}_p^*$ y elementos $\alpha^a (mod\ p)$ y $\alpha^b (mod\ p)$; encuentre $\alpha^{ab} (mod\ p)$.*

Definición 3.1.5. *El problema Diffie-Hellman generalizado (PDHG) es el siguiente: dado un grupo cíclico finito G, un generador α de G y elementos α^a, α^b de G. Encontrar α^{ab}.*

Generación de Parametros Públicos y Llaves.

En los sistemas de logaritmo discreto las llaves que son utilizadas están asociadas a los parámetros (p, q, g) los cuales son de dominio público. Los parámetros públicos son escogidos de la siguiente forma; p es un número primo, q es un divisor primo de $p - 1$ y g en el conjunto $\{1, p - 1\}$ tal que g tiene orden q respecto a p, es decir; q es el entero positivo más pequeño

que satisface $g^q = 1(modp)$. Las llaves privadas x son números enteros escogidas aleatoriamente en el conjunto $\{1, q-1\}$. La llave pública para cada llave privada x es de la forma $y = g^x(mod\ p)$. Aquí justamente reside la complejidad abstracta de estos esquemas de llave pública ya que determinar la llave privada x conociendo únicamente los parámetros públicos (p, q, g) y y no es computable en tiempo computacionalmente finito, éste es el Problema de Logaritmo Discreto. La seguridad y eficiencia de estos esquemas está respaldada por la complejidad computacional de determinar x conociendo únicamente los parámetros públicos antes mencionados. Los algoritmos mostrados a continuación resumen la generación de parámetros para Logaritmo Discreto y la generación de las parejas de llaves.

Algorithm 1 Generación de parámetros públicos LD.

Input: Parámetros iniciales de seguridad l, t.
Output: Parámetros de dominio público (p, q, g).

1) Escoger un número primo q de longitud t en bits y un primo p de longitud l en bits tal que q divida a $p-1$.

2) Escoger un elemento g de orden q:
2.1) Escoger h arbitraria en el conjunto $\{1, p-1\}$ y calcular $g = h^{(p-1)/q}(mod\ p)$.
2.2) Si $g = 1(mod\ p)$ regresar al paso 2.1

3) Regresar (p, q, g).

En los algoritmos anteriores mostramos la estructura principal para generar los parámetros de dominio público y las parejas de llaves públicas y privadas, cabe mencionar que esta estructura puede ser modificada dependiendo del contexto en el cual se esté implementando. Para el contexto de esta tesis el cual es Dispositivos Móviles la generación de los parámetros públicos varía un poco como veremos en el capítulo de applicación, en el cual expondré a detalle un App para iOS que he desarrollado como ejemplo.

Algorithm 2 Generación de llave pública y llave privada.

Input: Parámetros de LD de dominio público (p, q, g)
Output: Llave pública y, Llave privada x.

1) Escoger aleatoriamete x en $\{1, q-1\}$.

2) Calcular $y = g^x (mod\ p)$.

3) Regresar la pareja de llaves (y, x).

3.1.2 Factorización de números en $\mathbb{Z}$

RSA fue concebido por Rivest, Shamir y Adleman, fue propuesto en 1977 poco tiempo después del descubrimiento de la Criptografía de Lave Pública.

-Generación de Llaves RSA.
Una pareja de llaves RSA puede ser generada utilizando el siguiente algoritmo;

Algorithm 3 Generación de parejas de llaves RSA.

Input: Parámetro de seguridad l.
Output: Llave pública RSA (n, e) y llave privada d.

1) Escoger aleatoriamente dos números primos p y q que tengan la misma longitud en bits $l/2$.

2) Calcular $n = p * q$,y $\phi = (p-1)(q-1)$.

3) Escoger un entero aleatorio e, tal que; $1 < e < \phi$, y $gcd(e, \phi) = 1$

4) Calcular el entero d que satisface $1 < d < \phi$, y $ed = 1 (mod\ \phi)$

5) Regresar (n, e, d).

La llave pública consta de un par de enteros (n, e) donde el módulo n RSA es el producto de dos números primos aleatoriamente escogidos y secretos que además tienen la misma longitud en bits. El exponente de cifrado e

es un entero que satisface $1 < e < \phi$ y $gcd(e, \phi) = 1$, donde $\phi = (p-1)(q-1)$. La llave privada d también llamada exponente de descifrado es un número que satisface $1 < d < \phi$ y $ed = 1(mod\ \phi)$. Ha sido demostrado que determinar la llave privada d utilizando únicamente la llave pública (n, e) es computacionalmente equivalente a descomponer n en sus factores primos p y q. Este último es conocido como el problema de factorización de enteros (IFP), cuya dificultad constituye la base de seguridad de los esquemas de cifrado y generación de llaves RSA.

El resto del capítulo veremos a mayor profundidad cómo funcionan las variantes principales de los esquemas de llave pública. Es muy importante tener en mente que las versiones de los algoritmos presentados en esta tesis son la versión en papel de la implementaciones en el compilador, ya que antes de llevarlos acabo tendríamos que hacer algunas modificaciones como completar la longitud o peso del texto antes de cifrar. Estas modificaciones se hacen con el objetivo de adaptar el esquema al contexto en el que se está usando a fin de tenerlo listo para cualquier atacante. De cualquier forma le daremos un vistazo que resultará muy ilustrativo a las ideas principales que están detrás de RSA, PLD y a los esquemas de llave pública basados en curvas elípticas.

3.2 Cifrado de Llave Pública.

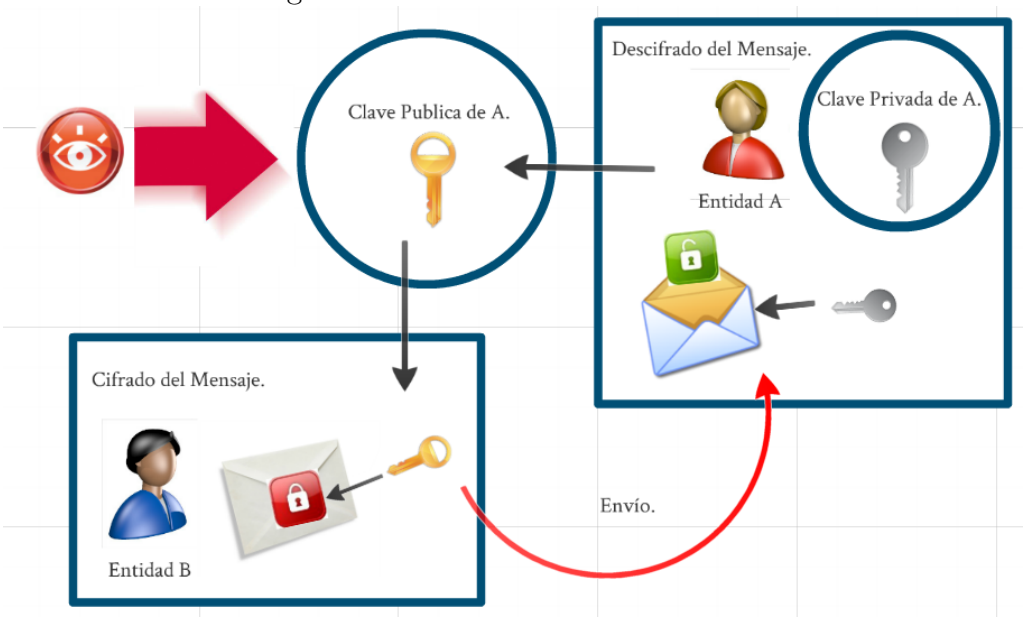
En esta sección consideraremos dos de las más influyentes técnicas para el cifrado de llave pública, también llamado cifrado asimétrico. Como mencionamos previamente en un esquema de llave pública cada entidad A posee una llave pública e y una llave privada d. En un sistema considerado seguro, la tarea de calcular d dada e es computacionalmente imposible. La llave pública define una transformación de cifrado E_e, mientras que la llave privada define la asociada transformación de descifrado D_d. Cualquier entidad B que dese enviar un mensaje m a A tendrá que; obtener una copia de la llave pública de A e, usar la transformación de cifrado asociada para obtener el texto cifrado $c = E_e(m)$, y finalmente transmitir c a A. Para descifrar c, A tendrá que aplicar la transformación de descifrado asociada para obtener el mensaje original $m = D_d(c)$.

La llave pública no tiene que ser secreta, de hecho, puede estar al alcance

de todos los usuarios que deseen verla y usarla. Únicamente requerimos de su autenticidad para garantizar que A es de hecho la única entidad que conoce la correspondiente llave privada, esto debido a que si y sólo si A posee la correspondiente llave privada entonces podrá leer el mensaje. Una ventaja primordial de este tipo de sistemas es el hecho de que generar llaves secretas auténticas es generalmente más fácil que distribuir llaves secretas utilizando un canal seguro, como se es requerido en los sistemas de llave simétrica.

El objetivo principal de el cifrado de llave pública es el de proveer privacidad y confidencialidad. Observemos que ya que la transformación de cifrado de A es de dominio público, el cifrado de llave pública por si mismo no provee de verificación del origen de los datos o integridad de datos. Dichos requerimientos deben ser proporcionados utilizando técnicas adicionales, incluyendo códigos verificación de mensajes y firmas digitales.

Figura 3.1: Cifrado de Llave Pública.



Los esquemas de cifrado de llave pública son sustancialmente más lentos que los de cifrado de llave simétrica (como DES y AES), ya que las llaves utilizadas como parámetros para los algoritmos de cifrado y descifrado son distintas. Debido a este hecho, el cifrado de llave pública es comúnmente

utilizado en la práctica con el objetivo de transportar llaves subsecuentemente utilizadas para el cifrado de bloques de datos mediante algoritmos de llave simétrica y otras aplicaciones incluyendo integridad de datos y verificación. Las técnicas de cifrado de llave pública también nos proveen de garantías de verificación en protocolos de verificación de identidad y de protocolos de establecimiento de llaves verificadas.

3.2.1 Cifrado ElGammal

El esquema de cifrado de llave pública ElGammal puede ser pensado como el Intercambio de Llaves Diffie-Hellman. Su seguridad está basada en el Problema de Logaritmo Discreto (Sección 3.1.1) y en el Problema de Diffie-Hellman(Sección 3.1.1.1). En esta sección presentaré los procedimientos básicos de cifrado y descifrado para el esquema de Llave Pública ElGammal. Veamos cómo funciona: supongamos que y es la llave pública del Receptor, entonces el Emisor selecciona aleatoriamente una k (como vimos en la sección anterior) y hace uso de la llave pública del destinatario para calcular $y^k(mod\ p)$. Para cifrar el mensaje no cifrado m el emisor multiplica m por $y^k(modp)$ (m previamente refinado utilizando una función hash). El Emisor envía el producto calculado $c_2 = m(y)^k(mod\, p)$ y también envía $c_1 = g^k(mod\, p)$ al Receptor quien hace uso de su llave privada x para calcular; $c_1^x = g^k x = y^k(mod\ p)$, después hace uso del resultado para dividir a $c_2 = m(y)^k(mod\ p)$ de esta forma obteniendo m que es el mensaje no cifrado. En este caso si un atacante desea interceptar el mensaje m tendría que calcular $y^k(mod\ p)$ conociendo únicamente los parámetros públicos (p, q, g), y, $c_i = g^k(mod\ p)$ lo cual se cree imposible de calcular en un tiempo computacionalmente finito. En este contexto este problema se llama el Problema de Diffie-Hellman (PDH). Se cree que este problema PDH es tan difícil de resolver como el Problema de Logaritmo Discreto (PLD), lo cual ha sido demostrado ya en varios casos particulares. A continuación presento los algoritmos básicos para este esquema de cifrado.
Es importante resaltar el hecho de que el mensaje cifrado es del doble de la longitud del texto en claro. Esto claramente es una desventaja ya que tanto el tiempo de transmisión como los recursos requeridos para llevarla a cabo se verán duplicados también.

Algorithm 4 Cifrado Básico ElGammal

Input: Parámetros LD de dominio público (p, q, g), llave pública y, texto no cifrado m en $\{0, p-1\}$
Output: Texto Cifrado (c_1, c_2).

1) Escoger k en $\{1, q-1\}$.
2) Calcular $c_1 = g^k (mod\ p)$.
3) Calcular $c_2 = m(y)^k (mod\ p)$.
4) Regresar (c_1, c_2)

Algorithm 5 Descifrado Básico ElGammal

Input: Parámetros LD de dominio público (p, q, g), llave privada x, texto cifrado (c_1, c_2).
Output: Texto No Cifrado m.

1) Calcular $m = c_2 * (c_1)^{(-x)} (mod\ p)$
2) Regresar (m).

3.2.2 Cifrado RSA

El esquema de cifrado RSA fue concebido por R. Rivest, A. Shamir, y L. Adleman y hoy en día es uno de los más utilizados a nivel mundial. Tiene la capacidad de proveer integridad y seguridad de datos, y su respaldo abstracto radica en el problema de factorización de números enteros (Sección 3.1.2). En esta sección describiremos el esquema de cifrado RSA y daremos algoritmos para su implementación.

RSA es un esquema de llave Asimétrica que se basa en el hecho de que $ed = 1 (mod\ \phi)$ lo cual implica que; $m^e d = m (mod\ n)$, para cualquier entero m. Esto se sigue también de que seleccionamos de esta forma a e y d en el paso de generación de llaves. Los siguientes cuadros de texto ilustran de una forma básica los algoritmos de Cifrado y Descifrado RSA respectivamente;

El algoritmo de descifrado funciona ya que $c^d = (m^e)^d = m (mod\ n)$ justo como habíamos visto en el párrafo anterior. El valor e es el inverso multiplicativo de d modulo n, esto nos permite invertir el proceso de cifrado de una forma efectiva y segura. La seguridad de este tipo de esquemas reside en la complejidad de calcular el texto NO cifrado m del texto Cifrado

Algorithm 6 Cifrado RSA Básico.

Input: Llave pública (n, e), texto no cifrado en valor numérico; m en $\{0, n-1\}$.
Output: Texto cifrado c.

1) Calcular $c = m^e (mod\ n)$
2) Regresar (c)

Algorithm 7 Descifrado RSA Básico.

Input: Llave pública (n, e), Llave privada d, texto cifrado c.
Output: Texto Descifrado m.

1) Calcular $m = c^d (mod\ n)$
2) Regresar(m)

$c = m^e (mod\ n)$ y de los parámetros públicos n y e. Aquí encontramos la esencia de la complejidad abstracta de este esquema; encontrar las raíces módulo n para revertir el proceso. Otra forma de revertir el proceso de cifrado RSA es mediante la implementación de un Oráculo que tenga la capacidad de descifrar mensajes RSA; ésta es una solucion alterna a la factorización de $n \in Z$. Dicho de otra forma descifrar RSA no implica factorizar n, y, sin embargo, encontrar la factorización explícita de n implica una ruptura total de RSA.

Capítulo 4

Firmas Digitales.

Los esquemas de firmas digitales nos proveen de métodos y algoritmos que nos permiten asegurarnos de que cada entidad puede imprimir su marca de identidad (firma) en los documentos digitales (mensaje) de una forma análoga a las firmas escritas a mano en papel. En este capítulo estudiaremos (dándole el mayor enfoque matemático posible) los esquemas de firmas digitales y sus propiedades desde el enfoque de la criptografía de llave pública en general.

Un Esquema (mecanismo) de Firma Digital $F(Sig, Ver)$ consiste de un algoritmo de generación de firmas Sig y un algoritmo de verificación de firmas asociado Ver. Sea M un conjunto finito de palabras llamado espacio de mensajes, M_s el espacio de mensajes listos para firmar, K el espacio de todas las llaves y S el espacio de firmas. Un algoritmo de generación de firma digital o algoritmo de generación de firma, es un método para producir una firma digital $Sig(M_s, K) \to S$. Notemos que el algoritmo de firmado no es aplicado directamente a los elementos $m \in M$, si no a los elementos $m_s \in M_s$. Esto se debe a que los mensajes en M tienen todos longitudes diferentes y representaciones, por lo tanto refinamos el espacio de mensajes $M \to M_s$ con el objetivo de dejar todos los mensajes listos para firmar. De aquí es natural que existe una función $R : M \to M_s$ la cual llamamos la función de Redundancia, las propiedades algebráicas y algorítmicas de dicha función dependerán del tipo de esquema y del contexto en el cual se implementen.

Para esta tesis nos interesa fijarnos en el caso en el que la función R es una función hash, si h es la función hash que está siendo usada para refinar los mensajes (firmas con apéndice, sección 4.2.1) que van a ser firmados en-

tonces $h(M) = M_h$ está contenido en M_s. Las funciones hash serán el tema central de el capítulo siguiente y uno de los puntos principales de esta tesis.

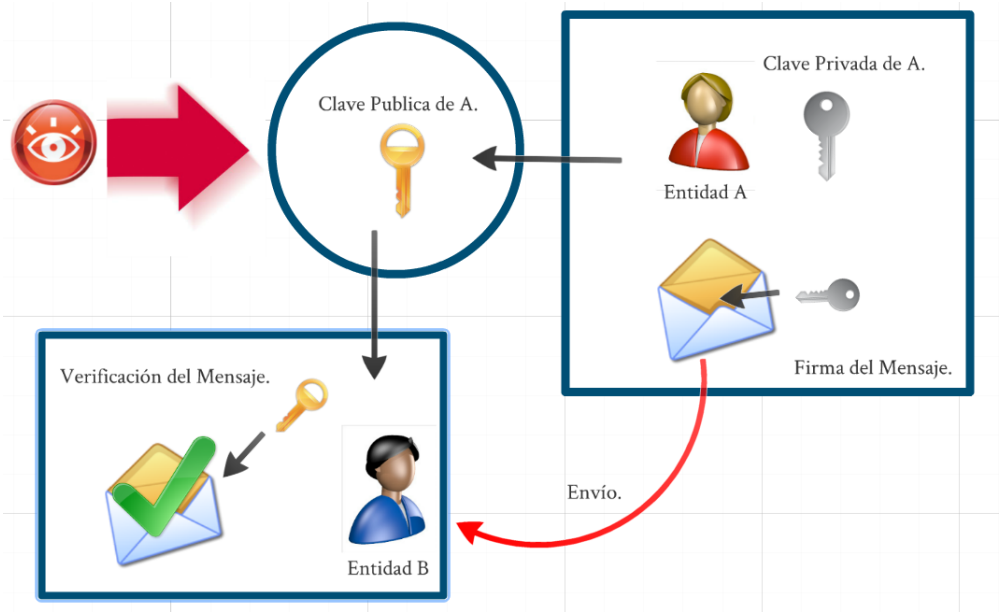
Ahora podemos definir Firma Digital en cuestión del esquema y sus algoritmos Sig y Ver, podemos decir que es el resultado de aplicar el algoritmo Sig a una llave privada k (de ser requerida) y a un mensaje m, es decir; una firma digital es una palabra en algún alfabeto finito que asocia un mensaje (en forma digital) y una llave(privada) a una entidad creadora (firmador).
A su vez un algoritmo de verificación de firma digital (o algoritmo de verificación) es un método para verificar que una firma digital es auténtica, es decir, que realmente fue creada por la entidad especificada. $Ver(S) \to \{0, 1\}$. El codominio de Ver es el conjunto $\{0, 1\}$, será 1 solamente en caso de que la firma sea auténtica, de lo contrario será igual a 0.

Aprovechemos para dar la siguiente definición; Un proceso (o procedimiento) de firmado consta de un algoritmo de generación de firma digital Sig acompañado por un método para formar datos en mensajes que puedan ser firmados. De la misma forma un proceso (o procedimiento) de verificación de firmas digitales consiste de un algoritmo de verificación de firmas digitales acompañado por un método para recuperar datos de un mensaje dado. (A veces se hace poca distinción entre los términos esquema y proceso y son usados sin diferencia.)
La firma digital de un mensaje es una palabra en algún alfabeto finito (generalmente un número entero) el cual, la mayoria de las veces, depende de un secreto conocido únicamente por el firmador y también en los contenidos del mensaje que está siendo firmado. Los contenidos del mensaje deben estar incluidos en la firma digital con el objetivo de hacer posible el calcular qué firma pertenece a qué mensaje. Los contenidos del mensaje son previamente refinados haciendo uso de una función hash criptográfica (SHA-1, MD-5 son usadas hoy en día), esto con el objetivo de hacer computacionalmente imposible obtener el mensaje en claro m de su valor hash $H(m)$ y así asegurarnos de que es imposible obtener los contenidos del mensaje de la firma digital. Una de las propiedades más importantes de las firmas digitales es que son verificables, es decir: si surge una disputa en función del origen o la creación de una firma (un usuario podría negar haber creado y firmado un documento o un tratado con identidad fraudulenta), una tercera entidad debería de poder resolver el conflicto de una forma equitativa sin necesitar accesar las llaves privadas del firmador.

4.1 Seguridad y Aplicaciones

Idealmente un esquema de firma digital debe ser infalsificable bajo el ataque de mensaje escogido. Dicho de una manera informal: un adversario que puede obtener las firmas de la entidad A para cualquier menaje que desee no debe ser capaz de falsificar una firma para algún mensaje dado. Mas tarde en el capítulo veremos en especifico los tipo de ataques a esquemas de firmas digitales.
Las ideas y la utilidad de los esquemas de firmas digitales fueron descubiertas varios años antes de que cualquier aplicación fuera posible. El primer método concebido fue el esquema de llaves RSA, el cual es todavía una de las técnicas más versátiles y prácticas. La investigación en estos esquemas ha dado lugar a en varios nuevos enfoques para este problema. Algunos ofrecen ventajas significativas en funcionalidad e implementación.

Figura 4.1: Firmas Digitales.



Las firmas digitales tienen muchas aplicaciones en la seguridad de la in-

formación, incluyendo; integridad de datos (la seguridad de que los datos no han sido modificados por un atacante), verificación del origen de los datos (la seguridad de que el origen de los datos es confiable), y no repudio (la seguridad de que una entidad no puede negar previos actos y compromisos). Una de las aplicaciones más importantes de las firmas digitales es la certificación de llaves públicas en redes muy grandes. Con certificación nos referimos al hecho de ligar la identidad de un usuario a una llave pública, con el objetivo de que en el futuro otras entidades puedan verificar llaves públicas sin la asistencia de una tercera entidad confiable.

4.2 Clasificación de los esquemas de firmas digitales

Hay distintos y muy diferentes esquemas de firmas digitales, y la esencia de cada uno de ellos depende directamente del problema matemático en el cual esté basado y en el contexto bajo el cual se esté aplicando. La clasificación mas general la podemos dar como sigue:

1) Esquemas de Firmas Digitales con Apéndice: requieren el mensaje original como parámetro de entrada del algoritmo de verificación.

2) Esquemas de Firmas Digitales con recuperación de mensajes: no requieren el mensaje original como parámetro de entrada del algoritmo de verificación. En este caso el mensaje original es recuperado de la firma en sí.

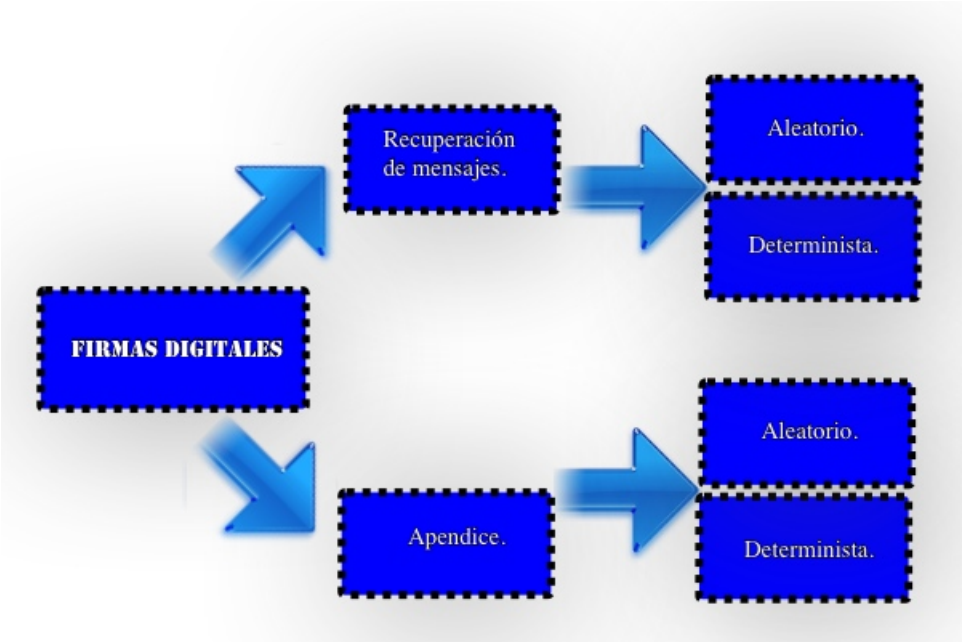
Estas clasificaciones pueden ser divididas más y mas en Esquemas Aleatorios o Deterministas como veremos en la siguiente definición;

Definición 4.2.1. *Un esquema de firmas digitales (ya sea con apéndice o con recuperación de mensajes) es un esquema de firmas digitales aleatorio si* $|R| > 1$*; de otra forma, se dice que el esquema es determinista.*

Donde R es un conjunto utilizado para identificar transformaciones de firmado específicas, lo llamamos *conjunto* de conteo de firmas. Este conjunto puede ser pensado como el espacio de llaves K, en el sentido de que cada vez

que seleccionamos una llave para llevar a cabo un algoritmo de generación de firmas estamos definiendo implícitamente un conjunto de transformaciones $S_k : M_s \to S$,$k \in R$. Esta clasificación evidentemente nos sirve para identificar conjuntos de transformaciones S_k con llaves en el espacio R. En el siguiente diagrama mostramos la clacificación mencionada:

Figura 4.2: Clasificación Firmas Digitales.



4.2.1 Esquemas de Firmas Digitales con Apéndice.

Los esquemas de firmas digitales con apéndice son considerados los mas seguros y estables, y, por ello son los más utilizados la en práctica. Su algoritmo de generación de firma hace uso de funciones criptográficas hash en vez de funciones de redundancia. Esto último los hace menos propensos a ataques de falsificación de identidad.

Definición 4.2.2. *Los esquemas de firmas digitales que requieren el mensaje como un parámetro para el algoritmo de verificación son llamados Esquemas de Firmas Digitales con Apéndice.*

Ejemplos de mecanismos que implementan este tipo de esquemas de firmas son DSA, ElGammal y Schnorr. El esquema de firmas utilizado para la implementación de los algoritmos descritos en esta tesis será un esquema con apéndice ya que el punto central es desarrollar la familia de funciones hash que serán utilizadas en los algoritmos de generación y verificación de firmas.

Algorithm 8 Generación de Firmas para esquemas de Firmas Digitales con Apéndice.

Resumen: Cada entidad selecciona una llave privada para firmar mensajes, a su vez se genera una llave pública para que otras entidades puedan verificar las firmas.

1) La entidad A que va a llevar a cabo el firmado selecciona una llave privada k que define un conjunto $S_A = S_{A,k} : K \in R$ de transformaciones de firmado. Cada $S_{A,k}$ es una función 1-1 de M_h a S y es llamada una transformación de firmado.

2) El conjunto S_A define el conjunto correspondiente de algoritmos de verificación V_A que van de (M_h x S) a $\{0, 1\}$ (falso o verdadero) tal que; $V_A(m, s^*) = 1$, si $S_{A,k}(m) = s^*0$, de otro modo para todos los $\hat{m} \in M_h, s^* \in S$; aquí tomamos $\hat{m} = h(m)$ para $m \in M$ y una función hash h. A V_A le llamamos el conjunto de transformaciones de verificación y está construido de tal modo que sus elementos pueden ser calculados sin el conocimiento de la llave privada del firmador.

3) La llave pública de A es el V_A y su llave privada es S_A.

La figura 4.1 nos provee de una representación esquemática de los algoritmos de generación de firmas y verificación de un esquema de firmas digitales con apéndice.

Las siguientes tres propiedades son requeridas para los algoritmos de generación de firmas y de verificación:

i) Para cada $k \in R$, el cálculo de $S_{A,k}$ debe ser eficiente.
ii) El cálculo de V_A debe ser eficiente
iii) Debe de ser computacionalmente imposible para alguna entidad diferente

Algorithm 9 Generación y Verificación de Firmas (Esquemas de Firmas Digitales con Apéndice)

Resumen: La entidad A produce la firma $s \in S$ para el mensaje $m \in M$ que más tarde puede ser verificada por cualquier entidad B.

1) Generación de Firmas. La entidad A hace lo siguiente:
a) Escoger un elemento $k \in R$
b) Calcular $\hat{m} = h(m)$ y $s^* = S_{A,k}(\hat{m})$.
c) La firma de A para m es s^*. Ambos m y s^* son valores públicos y están al alcance de cualquier entidad que quiera verificar la firma.

2) Verificación de Firmas. La entidad B que desee verificar la firma debe hacer lo siguiente;
a) Obtener la llave pública de A V_A.
b) Calcular $\hat{m} = h(m)$ y $u = V_A(\hat{m}, s^*)$.
c) Aceptar la firma como auténtica si y solamente si $u = 1$ (verdadero).

---

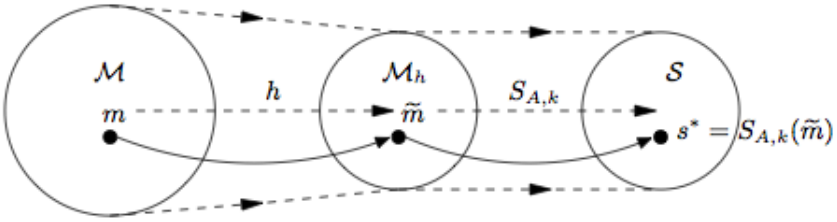
de A encontrar $m \in M$ y alguna $s^* \in S$ tales que $V_A(\dot{m}, s^*) = 1$(verdadero), donde $\hat{m} = h(m)$.
Los algoritmos de generación de firmas de un esquema con apéndice son aplicados a mensajes de una longitud arbitraria, siendo la función hash h la responsable del refinamiento y el acote de longitud requeridos para dejar los mensajes listos para firmar. La función de un solo sentido h del algoritmo de arriba es generalmente escogida como una función hash libre de colisiones. Veremos esto con más detalle en el capítulo siguiente.

4.2.2 Esquemas de Firmas Digitales con recuperación de mensajes

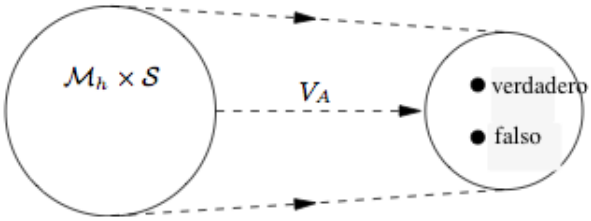
Los esquemas de firmas descritos en esta sección tienen la propiedad de que el mensaje firmado puede ser recuperado de la firma digital en si misma. En práctica son usados para firmar mensajes cortos.

Definición 4.2.3. *Un esquema de Firmas Digitales con Recuperación de Mensajes es un esquema para el cual el conocimiento a priori de los mensajes no es necesario para ejecutar el algoritmo de verificación.*

Figura 4.3: Vistazo global a un Esquema de Firmas Digitales con Apéndice.



(a) El proceso de Firmado.



(b) El proceso de Verificación.

Ejemplos usados hoy en día de mecanismos que implementan estos esquemas son: RSA, Rabin y Nyberg-Rueppel.

El diagrama que presentamos a continuación nos provee de una representación esquemática de los algoritmos de generación de firmas y verificación de un esquema de firmas digitales con recuperación de mensajes.

Las siguientes tres propiedades son requeridas para los algoritmos de generación de firmas y de verificación:
iv) Para cada $k \in R$, el cálculo de $S_{A,k}$ debe ser eficiente.
v) El cálcuo de V_A debe ser eficiente
vi) Debe de ser computacionalmente imposible para alguna entidad diferente de $A4$ encontrar alguna $s^* \in S$ tal que $V_A(s^*) \in M_R$.

Algorithm 10 Creación de llaves para esquemas de firmas digitales con recuperación de mensajes.

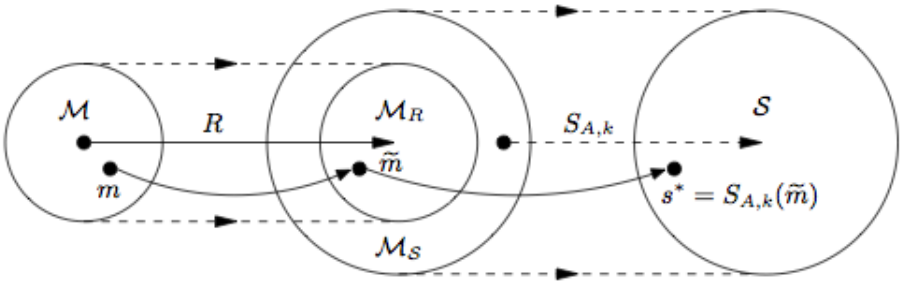
Resumen; Cada entidad selecciona una llave privada para firmar mensajes, también se genera una llave pública a fin de que otras entidades puedan verificar las firmas.

1) La entidad A que va a llevar a cabo el firmado selecciona una llave privada k que define un conjunto $S_A = S_{A,k} : k \in R$ de transformaciones de firmado. Cada $S_{A,k}$ es una función 1-1 de M_h a S y es llamada una transformación de firmado.

2) S_A define una función correspondiente V_A con la propiedad de que V_A seguido de $S_{A,k}$ es la función identidad en M_S para toda $k \in R$. V_A se llama la transformación de verificación y es construida de tal forma que para ejecutarla no es necesario tener conocimiento de la llave privada de la entidad que lleva a cabo el firmado.

3) La llave pública de A es V_A, y su llave privada es el conjunto S_A.

Figura 4.4: Vistazo global a un Esquema de Firmas Digitales con Recuperación de Mensajes.



Función Redundancia.

Aunque la función R es de dominio público y su inverso $R^{(-1)}$ es fácil de calcular, la selección de R es crítica y NO debe ser elegida independientemente de la elección de la transformación de generación en S_A.

Algorithm 11 Generación y Verificaron de Firmas para esquemas con recuperación de mensajes.

Resumen: la entidad A produce una firma $s \in S$ para un mensaje $m \in M$ la cual más tarde puede ser verificada por cualquier entidad que así lo desee. El mensaje m se recupera directamente de s.

1) Generación de Firmas: la entidad A lleva a cabo lo siguiente;
a) Escoger un elemento $k \in R$.
b) Calcular $\hat{m} = R(m)$ y $s^* = S_{A,k}(\hat{m})$. Donde R es la función de redundancia.
c) La firma s^* de A es puesta al alcance de cualquier entidad que desee verificarla o extraer el mensaje de ella.

2) Verificación de Firmas. La entidad que va a verificar B debe hacer lo siguiente;
a) Obtener una copia auténtica de la llave pública V_A de A.
b) Calcular $\hat{m} = V_A(s*)$
c) Verificar que $m \in M_R$. (si no lo está entonces rechazar la firma)
d) Calcular $R^{-1}(\hat{m})$ para recuperar m de $\hat{m}$.

4.3 Tipos de ataques a Esquemas de Firmas Digitales.

El objetivo de los ataques de un adversario será el de falsificar firmas; esto será: producir firmas tales que al ser verificadas sean aceptadas como firmas provenientes de otra entidad ajena al adversario. A continuación enunciamos criterios que nos especifican qué significa romper un esquema de firmas.

1) Ruptura Total: Un adversario que logra calcular la información de la firma del firmador o encuentra un algoritmo eficiente funcionalmente equivalente al algoritmo de generación de firmas con el cual puede falsificar la identidad del firmador.

2) Falsificación selectiva: El adversario atacante es capaz de crear una firma valida para algún tipo, en particular de mensaje o clase de mensaje escogido a *priori*. Crear una firma con este tipo de ataque no necesariamente involucra al firmador legítimo.

3) Falsificación Existencial: Un adversario es capaz de falsificar una firma para al menos un mensaje. El adversario tiene poco o nada de control sobre el mensaje cuya firma es falsificada, el firmador legítimo puede o no estar involucrado en el fraude.

Los puntos anteriores nos describen con precisión las tres metas principales de un adversario para considerar roto un Esquema de Firmas Digitales. Existen dos mecanismos de ataque contra los esquemas de Firmas Digitales de Llave Pública.

1) Ataques de Llave: En este tipo de ataque el adversario conoce únicamente la llave secreta del firmador, y hace uso de ella para implementar un ataque eficiente.

2) Ataques de Mensaje: Aquí un adversario examina las firmas correspondientes a mensajes conocidos o previamente seleccionados. Los ataques de mensaje pueden ser subdivididos en tres clases.

a) Ataque de mensaje conocido: El adversario conoce las firmas para un

conjunto de mensajes que son de su pleno conocimiento pero no seleccionados por él.

b) Ataque de mensaje escogido: El adversario obtiene una lista de firmas que corresponden a un conjunto de mensajes antes de intentar el ataque. En este tipo de ataques los mensajes son seleccionados antes de que cualquier firma sea vista. Los ataques de mensaje escogido en contra de los esquemas de firmas son el análogo de los ataques de texto cifrado escogido contra los esquemas de cifrado de llave pública.

c) Ataque de mensaje escogido adaptable: El adversario tiene la capacidad de usar un conjunto de firmas como un oráculo; el adversario puede solicitar firmas de mensajes que dependan en la llave pública del firmante y puede solicitar firmas de mensajes que dependan de firmas o mensajes previamente obtenidos.

En principio un ataque de mensaje escogido adaptable es el tipo de ataque más difícil de prevenir. Es concebible que dados suficientes mensajes y sus firmas correspondientes un adversario pueda deducir un patrón y falsificar alguna firma de su preferencia. Estos tipos de ataques son difíciles de montar en la práctica, pero incluso así un Esquema de Firmas eficiente debe estar diseñado para prevenirlos.
El nivel de seguridad de un Esquema de firmas digitales puede variar dependiendo del contexto bajo el cual se esté aplicando. Por ejemplo en la situaciones en las que existe la posibilidad de que el adversario monte un ataque de alave, puede que sea suficiente diseñar el esquema para prevenir que el adversario tenga éxito al falsificar una firma. En situaciones en las cuales el adversario tiene la posibilidad de ataques de mensaje, es necesario asegurarse de que el esquema es seguro en caso de falsificaciones existenciales. Cuando una función hash h es usada en un Esquema de Firmas (como común mente se hace en practica), h debe ser una parte fija del proceso de generación de firmas, para que el adversario no tenga la posibilidad de cambiar el valor h en la firma por una función hash mas débil, y entonces monte un ataque de Falsificado Selectivo.
El resto de este capítulo veremos cómo funcionan los Esquemas de firmas Digitales más usados en la actualidad, estudiaremos sus propiedades en general y daremos una versión en papel de los algoritmos que son usados para su implementación. Debido a que el punto central de esta tesis es el diseño

de una familia de funciones hash para la implementación de un Esquema de Firmas, prestaremos atención a lo ya desarrollado para tener éxito en esta tarea.

4.4 DSA y Esquemas relacionados.

En esta sección presento el Algoritmo de Firma Digital (Digital Signature Algorythm (DSA)) y algunos esquemas relacionados con él. La mayoría de éstos tienen sus bases abstractas en los grupos $\mathbb{Z}_p^*$, para algún primo suficientemente grande. En general todos estos mecanismos pueden ser generalizados a algún grupo cíclico finito; como quedará explícitamente ilustrado cuando presentemos el Esquema de Firmas ElGammal.

Todos los Esquemas presentados en esta sección son considerados Esquemas de Firmas Digitales Aleatorios y nos darán como resultado del algoritmo de generación; firmas digitales con apéndice, el cuál puede ser modificado para resultar en firmas con recuperación de mensajes, lo cual resulta ventajoso en la práctica.
Una condición necesaria para la seguridad de este tipo de Esquemas es el hecho de que calcular logaritmos discretos en $\mathbb{Z}_p^*$ es computacionalmente muy costoso y difícil (vid. capítulo anterior), pero el hecho de que se cumpla esta condición no es necesariamente suficiente para considerar seguro el esquema en cuestión. Análogamente para los sistemas RSA; todavía no se ha demostrado que las firmas RSA sean seguras pese a que factorizar $p * q = n$ sea muy difícil.

4.4.1 Algoritmo de Firma Digital (Digital Signature Algorythm; DSA).

En Agosto de 1991 el U.S. National Institute of Standards and Technology (NIST) propuso un algoritmo para generar firmas digitales al cual en ese momento denominaron DSA. El DSA se ha convertido en un Estándar de Procesamiento de Información Federal (U.S. Federal Information Processing Standard (FIPS 186) llamado Estándar de Firma Digital (Digital Signature Standard (DSS),) y es el Esquema de Firmas que es reconocido por cualquier gobierno. El algoritmo de generación de firmas es una variante del esquema ElGammal y es un esquema de firmas digitales con apéndice. El mecanismo

de firmas requiere una función hash definida de la siguiente forma;

$$h : (0,1)^* \to \mathbb{Z}_q,$$

para algún entero q. DSS explícitamente requiere el uso de la función Secure Hash Algorithm (SHA-1).

Algorithm 12 Generación de Llaves para DSA.

Resumen: Cada entidad crea una llave pública y su correspondiente llave privada. Cada entidad A debe hacer lo siguiente:
1) Elegir un número primo q tal que $2^{159} < q < 2^{160}$.
2) Escoger t tal que $0 \leq t \leq 8$ y un número primo p tal que $2^{511+64t}$ <p< $2^{512+64t}$, con la propiedad de que q divida a $(p-1)$.
3) (Escoger un generador α de el único grupo cíclico de orden q en Z_p^*.)
3.1) Escoger un elemento $g \in Z*_p$ y calcular $\alpha = g^{(p-1)/q}(mod\ p)$.
3.2) Si $\alpha = 1$ regresar al paso 3.1
4) Escoger un entero aleatorio a tal que $1 \leq a \leq q-1$
5) Calcular $y = \alpha^a(mod\ p)$.
6) La llave pública de la entidad A es (p, q, α, y); y la llave privada es a.

En el algoritmo 12 la elección de los números primos p y q es de vital importancia para el correcto funcionamiento y más aún para la seguridad del algoritmo de generación de llaves. Debemos seleccionar el primo q primero y después intentar encontrar otro primo p tal que q divida a $p-1$.

Corroboremos ahora que el algoritmo de verificación es válido.

Demostración. Veamos que $v = r$, con v y r como en los algoritmos de arriba. Si (r, s) es una firma legítima de la entidad A respecto al mensaje m, entonces $h(m) = -ar + ks(mod\ q)$ debe cumplirse. Si multiplicamos ambos lados de la congruencia por w y reacomodamos obtenemos $w * h(m) + arw(mod\ q)$. Pero esto simplemente es (sustituyendo $u_1 y u_2$) $u_1 + au_2 = k(mod\ q)$. Elevando α a ambos lados de la ecuación nos queda; $[\alpha^{u_1} y^{u_2}(mod\ p)](mod\ q) = [\alpha^k(mod\ p)](mod\ q)$. Por lo tanto $v = r$ como requeríamos.

La seguridad de DSA radica en dos distintos pero relacionados problemas de logaritmo discreto. Uno es el problema de logaritmo en $\mathbb{Z}_p^*$ donde los

Algorithm 13 Generación y Verificación de Firmas DSA

Resumen. La entidad A firma un mensaje binario m de longitud arbitraria. Cualquier entidad B puede verificar la firma haciendo uso de la llave pública de A.
1) Algoritmo de Generación de Firma. La entidad A lleva a cabo lo siguiente:
a) Escoger un entero aleatorio $k, 0 < k < q$.
b) Calcular $r = \alpha^k (mod\ p)(mod\ q)$
c) Calcular $k^{-1} (mod\ q)$
d) Calcular $s = k^{-1} h(m) + ar (mod\ q)$
e) La firma de A para m es la pareja (r, s).
2) Algoritmo de Verificación. Para verificar la firma (r, s) de A con respecto al mensaje m, B debe hacer lo siguiente:
a) Obtener la auténtica llave pública de A; (p, q, α, y).
b) Verificar que $0 < r < q$ y $0 < s < q$; de lo contrario rechazar la firma.
c) Calcular $w = s^{-1} (mod\ q)$ y $h(m)$.
d) Calcular $u_1 = w * h(m) (mod\ q)$ y $u_2 = rw (mod\ q)$.
e) Calcular $u = (\alpha^{u_1} y^{u_2} mod\ p)(mod\ q)$.
f) Aceptar la firma si y sólo si $v = r$.

poderosos métodos de calculo de índices son aplicables; el otro es el problema de logaritmo en grupos cíclicos de orden q, donde los mejores métodos existentes corren en tiempo raíz cuadrada (vid. capítulo anterior.) .

Como DSA es un caso especial de ElGammal respecto a la ecuación de las firmas, las consideraciones de seguridad para el último son pertinentes para DSA. El tamaño de los parámetros para ejecutar los algoritmos de generación de llaves están especificados en el Federal Information Processing Standards (FIPS 186) con el propósito de asegurar el óptimo desempeño de este tipo de esquemas. El tamaño de q está determinado en 160 bits por el algoritmo de generación de parámentos DSA, mientras que el tamaño de p puede ser un múltiplo de 64 entre 512 y 1024 bits. Un primo de longitud 512 bits provee al esquema de seguridad marginal contra ataques coordinados. Desde 1996 es recomendado usar un módulo de al menos 768 bits de longitud. FIPS 186 no permite usar primos p más grandes que 1024 bits. La elección de paramentos que se ajusten a estas recomendaciones nos garantiza poder crear esquemas seguros, pero debemos recordar que nuestra definición de seguridad gira en torno al poder computacional que tenemos al alcance

hoy en día. Sin duda alguna, estos parámetros van a ir variando conforme nuestro poder computacional aumente. De encontrarse una forma de calcular logaritmos discretos de una forma eficiente en las estructuras sobre las cuales están construidos los protocolos basados en logaritmo discreto (o cualquier otro enfoque matemático), sería necesario someterlos a un proceso de ajuste para seguir siendo considerados eficientes. Estos ajustes se pueden reflejar en el incremento de la longitud de los parámetros o inclusive en una restructuración pertinente de los algoritmos.

DSA es uno de los esquemas de firmas digitales más usados hoy en día, lo que ocurre, sin duda por el desempeño que nos ofrece. Veamos cuáles son las características del desempeño de DSA, para lo cual supondremos que p es un primo de longitud 768 bits. Bajo esta hipótesis la generación de firmas requiere: una exponenciación modular, la cual toma en promedio 240 multiplicaciones modulares (haciendo uso de técnicas comunes de exponenciación), el cálculo de un inverso multiplicativo con un módulo de longitud 160 bits, dos multiplicaciones modulares de 160 bits y una suma. Las operaciones de 160 bits son relativamente fáciles de llevar a cabo comparados con la exponenciación. DSA tiene la enorme ventaja de que los cálculos que requieren exponenciación pueden ser precalculados y no tienen que ser realizados en el momento en que se lleva a cabo el firmado. Esto es una ventaja si se le compara con RSA, en el cual las operaciones de exponención no pueden ser llevadas a cabo a *priori*. La parte más trabajosa del algoritmo de verificación de firmas son dos aplicaciones exponenciales módulo p cada una con exponentes de longitud 160 bits. En promedio cada una de ellas requiere 240 multiplicaciones modulares, y esto considerando que ninguno de los cálculos sea realizado en paralelo.

La probabilidad de que DSA sea desmantelado por un adversario es realmente cercana a cero. La verificación requiere el cálculo de $s^{-1}(mod\ q)$. Si $s = 1$ (neutro del grupo) entonces $s^{-1} = 1$ también, por lo tanto no se podrían llevar a cabo los cálculos para generar la firma. Para evitar esto el firmador debe verificar que s sea distinta de cero; pero s debe de ser un elemento aleatorio en $\mathbb{Z}_q$ entonces la probabilidad de que $s = 1$ es $(1/2)^{160}$. En la práctica es extremadamente difícil que se dé esta situación.

4.5 Esquema de firmas ElGammal.

El esquema de firmas ElGammal es un esquema de firmas aleatorio. Genera firmas digitales con apéndice sobre mensajes binarios de longitud arbitraria, y para generar las firmas requiere el uso de una función hash

$$h : [0,1]^* \to \mathbb{Z}_p,$$

donde p es un numero primo grande. Como dijimos en la sección anterior DSA es una variante de el esquema de firmas ElGammal.

Algorithm 14 Generación de Llaves para el Esquema de Firmas ElGammal.

Resumen: Cada entidad crea una llave pública y su correspondiente llave privada. Cada entidad A debe hacer lo siguiente:

1) Generar un primo aleatorio p y un generador α del grupo multiplicativo $\mathbb{Z}_p^*$.
2) Escoger un entero aleatorio a, $1 \leq a \leq p-2$.
3) Calcular $y = \alpha^a \ (mod\ p)$.
4) La llave pública de A es (p, α, y); la llave privada es a.

Demostremos ahora que el algoritmo de verificación funciona. Si la firma fue generada por A, entonces $s = k^{-1}h(m) - ar(mod\ (p-1))$. Multiplicando ambos lados por k obtenemos; $ks = h(m)ar(mod\ (p-1))$, y reacomodando nos queda; $h(m) = ar + ks(mod\ (p-1))$. Esto implica que $\alpha^{h(m)} = \alpha^{ar+ks} = (\alpha^a)^r r^s (mod\ p)$. Por lo tanto,$v_1 = v_2$ como requeríamos.
Veamos cuáles son las posibilidades de ataque más inminentes para el esquema de firmas ElGammal.

i) Un adversario puede intentar falsificar la firma de A en m escogiendo un entero aleatorio k y calculando $r = \alpha^k (mod\ p)$. El adversario debe determinar $s = k^{-1}h(m) - ar(mod\ (p-1))$. Si el problema de logaritmo discreto es impráctico computacionalmente un adversario no puede hacer más que escoger s aleatoriamente; la probabilidad es solamente $1/p$, última que para una p grande es muy cercana a cero.

ii) Al generar firmas debemos seleccionar una k distinta para cada mensaje m; de otra forma, la llave privada puede ser determinada de la manera siguiente. Supongamos que $s_1 = k^{-1}h(m_1) - ar(mod\ (p-1))$ y $s_2 =$

Algorithm 15 Generación y Verificación de Firmas ElGammal.

Resumen: la entidad A firma un mensaje binario m de longitud arbitraria. Cualquier entidad B puede verificar la firma haciendo uso de la llave pública de A.

1) Generación de Firmas. La entidad A debe hacer lo siguiente:
a) Escoger un entero secreto aleatorio k, $1 \leq k \leq p-2$, tal que $mcm(k, p-1) = 1$ b) Calcular $r = \alpha^k (mod\ p)$.
c) Calcular $k^{-1} (mod\ (p-1))$.
d) Calcular $s = k^{-1}(h(m) - ar)(mod\ (p-1))$.
e) La firma de A para m es la pareja (r, s).

2) Verificación de Firmas. Para verificar la firma (r, s) de A para m, B debe hacer lo siguiente.
a) Obtener la llave pública (p, α, y) auténtica de A.
b) Verificar que $1 \leq r \leq (p-1)$, y si no se cumple rechazar la firma.
c) Calcular $v_1 = y^r r^s (mod\ p)$.
d) Calcular $h(m)$ y $v_2 = \alpha^{h(m)} (mod\ p)$.
e) Aceptar la firma si y solamente si $v_1 = v_2$.

$k^{-1}h(m2) - ar(mod\ (p-1))$, entonces;$(s_1s_2)k = (h(m_1)h(m_2))(mod\ (p-1))$. Si s_1s_2 es distinto de $0(mod\ p-1)$, entonces $k = (s_1s_2)^{-1}(h(m_1)h(m_2))(mod\ (p-1))$. Una vez que conocemos k es muy fácil determinar a.

iii) Si no hacemos uso de una función hash h la ecuación de firmado nos queda: $s = k^{-1}m - ar(mod\ (p-1)$. Se vuelve fácil para un adversario montar un ataque de falsificación existencial. Escoger una pareja de enteros (u, v) tales que $mcd(v, p-1) = 1$. Calcular $r = \alpha^u y^u (mod\ p) = \alpha^{u+av} mod\ p$ y $s = -rv^{-1}(mod\ (p-1))$ La pareja (r, s) es una firma válida para el mensaje $m = su(mod\ (p-1))$, ya que $(\alpha^m \alpha^{-ar})^{s^{-1}} = \alpha^u y^v = r$.

iv) En uno de los pasos del algoritmo de verificación que mostramos arriba le indica al verificador que corrobore $0 < r < p$. Si esta verificación no se hace entonces un adversario puede firmar mensajes de su elección haciendo uso de un mensaje y su firma válida asociada firmado por A.

Supongamos que (r, s) es una firma para el mensaje m producida por A. El adversario escoge un mensaje $\hat{m}$ de su preferencia, calcula $h(\hat{m})$ y $u = h(\hat{m})[h(m)]^{-1}(mop\ (p-1))$. Ahora calcula $\hat{s} = su(mod\ (p-1))$ y $\hat{r}$ tal que $\hat{r} = ru(mod\ p-1)$, esto resulta en $\hat{r} = r(mod\ p)$. Lo último es válido por el teorema chino del residuo. La pareja $(\hat{r}, \hat{s})$ es una firma para el mensaje $\hat{m}$ que sería aceptado por el algoritmo de verificación si no se checa que $0 < r < p$.
La seguridad de este esquema depende también de una manera directa en la selección de los parámetros para generar las llaves. Los mas importantes son;
i) Ataque de Cálculo de Índices. El primo p debe ser suficientemente grande para prevenir el uso de ataques de cálculo de índice.
ii) Ataque de Pohlig-Hellman. El número primo $p-1$ debe ser divisible por un primo q suficientemente grande para prevenir el ataque de Logaritmo Discreto de Pohlig-Hellman.
iii) Generadores Débiles. Supongamos que $p = 1(mod\ 4)$ y el generador α satisface las siguientes condiciones:
a) α divide a $(p-1)$; y
b) Calcular logaritmos en un subgrupo S de orden α de $\mathbb{Z}_p^*$ puede ser eficientemente llevado a cabo.

En casos como éste un adversario gana la capacidad de falsificar firmas sin

conocimiento de la llave privada de A que serán aceptadas por el algoritmo de verificación. Veamos de una forma mas detallada como sería llevada a cabo esta falsificación sobre el mensaje m;

a) Calcular $t = (p-3)/2$ y fijar $r = q$.
b) Determinar z tal que $\alpha^{qz} = y^q (mod\ p)$ donde y es la llave publica de A. Esto es posible ya que α^q yy^q son elementos de S , de hecho α^q es un generador de S.
c) Calcular $s = th(m) - qz (mod\ (p-1))$
d) (r, s) es una firma para m que será aceptada por el algoritmo de verificación descrito arriba.

Este ataque funciona ya que la ecuación de verificación $r^s y^r = \alpha^{h(m)} (mod\ p)$ se satisface con la firma forjada por el adversario. Para ver esto primero observemos que $\alpha q = -1 (mod\ p)$, $\alpha = -q^{-1} (mod\ p)$, y que $q^{(p-1/2)} = 1 (mod\ p)$. De lo anterior deducimos que $q^t = q^{(p-1/2)} q^{-1} = -q^{-1} = \alpha (mod\ p)$. Ahora $r^s y^r = (q^t)^{[h(m)-qz]} y^q = \alpha^{h(m)} \alpha^{-qz} y^q = \alpha^{h(m)} y^{-q} y^q = \alpha^{h(m)} (mod\ p)$. Este ataque puede ser evitado si escogemos α como un generador de un subgrupo de $\mathbb{Z}_p^*$ de orden primo en vez de cómo un generador del grupo completo $\mathbb{Z}_p^*$.

4.6 Esquema de Firmas RSA

En esta sección describiremos el esquema de firmas RSA, considerando que la seguridad de dicho esquema depende directamente de la insolubilidad del problema de fatorización de números enteros (Sección 3.1.2). Tanto el espacio de mensajes como el espacio de mensajes cifrados para el esquema de cifrado RSA (Sección 3.2.2) es $\mathbb{Z}_n = \{0, 1, 2, ., n-1\}$ donde $n = pq$ es el producto de dos números primos escogidos aleatoriamente. La transformación de cifrado es una biyección, entonces podemos crear firmas digitales si revertimos los roles del cifrado y del descifrado. El esquema de firmas RSA es un esquema de firmas digitales determinista que nos provee de recuperación de mensajes (Sección 4.2.2). El espacio listo para firmar M_S y el espacio de firmas S son ambos $\mathbb{Z}_n$. Escogemos una función redundancia que será de dominio público de la forma $R : M \to \mathbb{Z}_n$.

La entidad que va a firmar el mensaje m primero calcula el refinamiento del mensaje $h = H(m)$ haciendo uso de una función hash criptográfica, donde

h sirve como un representante corto de m. El firmador entonces hace uso de su llave privada d para calcular la raíz s de h modulo n: $s = h^d(mod\ n)$. Notemos que $s^e h(mod\ n)$ de la sección anterior. El firmador entonces envía su mensaje m y su firma s a la entidad que los verificará. El receptor calcula nuevamente el refinamiento $h = H(m)$, recupera el refinamiento $\hat{h} = s^e(mod\ n)$ de s, y acepta la validez de la firma para m solamente si $h = \hat{h}$. La seguridad de estas firmas radica en la inhabilidad de un atacante para calcular raíces modulo n sin conocer la llave privada d. Los siguientes cuadros explican los esquemas básicos de generación y verificación de firmas RSA.

Algorithm 16 Generación de Firmas RSA

Input: Llave pública (n, e), llave privada d, mensaje m.
Output: Firma s.

1) Calcular $h = H(m)$, donde H es una función hash.
2) Calcular $s = h^d(mod\ n)$
3) Regresar(s).

Algorithm 17 Verificación de Firmas RSA

Input: Llave pública (n, e), mensaje m, firma s.
Output: Aceptación o rechazo de la firma.

1) Calcular $h = H(m)$, donde H es una función hash.
2) Calcular $\hat{h} = s^e(mod\ n)$
3) Si $h = \hat{h}$ regresar (Firma Aceptada, 1)
De otra forma regresar (Firma NO Aceptada, 0)

Los algoritmos anteriores son fáciles de implementar y hasta cierto punto seguros aunque es computacionalmente costoso cuando llegamos a la parte de la exponenciación modular, es decir: calcular $m^e(mod\ n)$ para cifrar y $c^d(mod\ n)$ para descifrar. Si se desea incrementar la efectividad del cifrado y la verificación de las firmas se sugiere seleccionar un exponente de cifrado pequeño e; en práctica $e = 3$ o $e = 2^{16} + 1$ son comúnmente utilizados.

Capítulo 5

Funciones Hash e Integridad de Datos.

Las funciones hash juegan un papel muy importante en la criptografía moderna, debido a que son el eje del funcionamiento de varios esquemas y mecanismos criptográficos. Encontramos aplicaciones de las mismas en diversas áreas de la teoría de la información, tanto en aplicaciones criptográficas como no criptográficas. En cualquier caso el objetivo de estas funciones es el de comprimir los elementos que reciben como argumento todos a una misma longitud. De esta forma facilitan la aplicación de los protocolos que sean requeridos.
El enfoque particular de esta tesis son las funciones hash criptográficas (de ahora en adelante llamadas solamente funciones hash), en particular nos enfocaremos en su uso para la integridad de datos y la verificación de mensajes. Las funciones hash toman un mensaje como parámetro de entrada y producen un parámetro al cual nos referiremos como código hash, resultado hash, valor hash o simplemente hash. Más a detalle, una función hash h envía bajo su regla de correspondencia, cadenas de ceros y unos de longitud arbitraria y finita m a cadenas de longitud fija, digamos n bits. Sea h una función hash tal que $h : D \to R$ y la cardinalidad de D es mayor que la de R, observemos que por la diferencia de tamaños de los conjuntos R y D la existencia de colisiones (parejas de parámetros de entrada con parámetros de salida idénticos) es inevitable. De hecho si restringimos a h a un dominio de elementos de longitud t bits ($t > n$) y si suponemos que h es aleatoria en el sentido de que existe la misma probabilidad de que h nos regrese cualquier elemento de su rango R para cualquier parámetro de entrada x, entonces alrededor de

2^{t-n} parámetros de entrada irían a dar a cada resultado de $(h, h(x))$, además la probabilidad de que dos parámetros de entrada escogidos al azar vayan a dar al mismo resultado es de 2^{-n} (independiente de t). La idea principal de las funciones hash criptográficas es que el valor hash sirva como una imagen compacta representativa (refinamiento o huella digital) de un mensaje que fue usado como parámetro de entrada. Este valor asignado al mensaje de entrada puede ser utilizado como si fuera la representación única del mensaje en cuestión. Las funciones hash son utilizadas en conjunción con los esquemas de firmas digitales para garantizar la integridad de la información, y cuando esto ocurre, generalmente el mensaje es refinado usando una función hash, para después hacer uso del valor hash como un representante del mensaje a fin de generar la firma. Existe otra clase de funciones hash que denominamos MACs (del inglés: Message Authentification Codes), los cuales nos permiten llevar a cabo la verificación de mensajes usando técnicas simétricas. Podemos pensar a un MAC como una función hash que toma dos parámetros de entrada; un mensaje y una llave secreta, y produce una cadena de ceros y unos de longitud fija (digamos n bits). Estos algoritmos están diseñados con la intención de sea imposible en la práctica producir el mismo valor de salida sin conocimiento de la llave secreta. Los MACs pueden ser usados con el propósito de proveer integridad y verificación del origen de la información así como verificación de identidad en los esquemas de llave simétrica.

5.1 Clasificación General.

Desde el punto de vista más general, las funciones hash pueden ser divididas en dos clases principales: funciones hash sin-llave, cuyas especificaciones nos piden únicamente un parámetro de entrada (un mensaje); y funciones hash con-llave, cuyas especificaciones nos piden dos parámetros de entrada, un mensaje y una llave secreta. A continuación definiremos de una forma informal una función hash.

Definición 5.1.1. *Una función hash tiene como mínimo las siguientes dos propiedades:*
1) Compresión: h manda un parámetro de entrada x con longitud en bits finita variable m, a un refinamiento $h(x)$ de longitud en bits finita fija n, $m > n$.
2) Fácil de calcular: dados h y el parámetro de entrada x, $h(x)$ es fácil de

calcular.

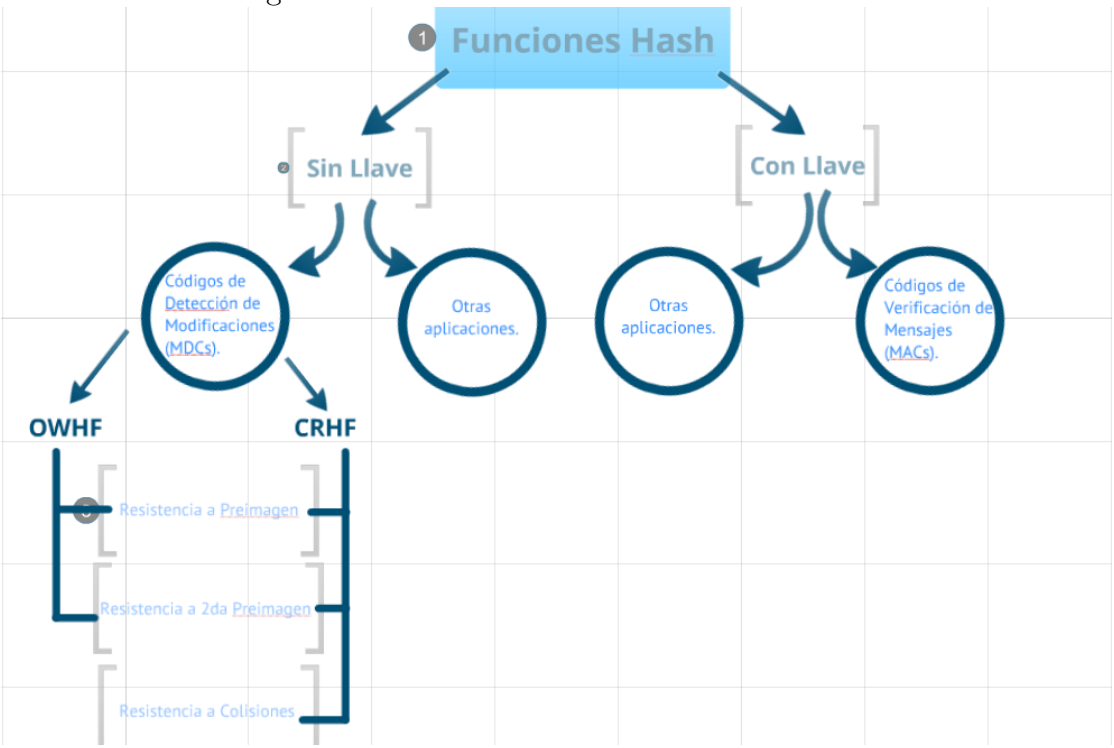
La clasificación anterior es la más evidente que podemos notar, y, no obstante para aproximarnos más a la implementación de dichas funciones es conveniente dar una clasificación mas cercana a los requerimientos de integridad para aplicaciones de esquemas en específico. De todas las clasificaciones del conjunto de funciones hash que podríamos dar, los siguientes dos tipos son consideradas en esta tesis, y el segundo conjunto que definiremos nos será de suma importancia para esta tesis ya que los miembros de la familia de funciones que construiremos pertenecerán a él.

1) Códigos de Detección de Modificaciones. (del inglés: Modification Detection Codes MDCs). También conocidos como Códigos de Detección de Manipulación y menos comúnmente como Códigos de integridad de Mensajes (MICs). El propósito principal de los MDCs es el de proporcionar una imagen hash (refinamiento) de un mensaje con el objetivo de usarlo como parámetro de entrada para los algoritmos de un esquema de integridad de información. El objetivo de este refinamiento es el de definir un conjunto de representantes del conjunto de mensajes y así adaptar el conjunto de mensajes (parámetros de entrada) a los requerimientos (longitud en bits principalmente) de los algoritmos de un esquema de integridad de información. Los MDCs son utilizados en conjunción con otros mecanismos y esquemas para garantizar la integridad de la información, son una subclase de las funciones hash sin llave, y pueden ser subdivididas aun más. Las clases de los MDCs que estudiaremos en esta tesis, enfocándonos principalmente en la segunda, son;
a) Funciones Hash de un solo sentido (OWHFs): para este tipo de funciones, encontrar un parámetro de entrada x tal que $x = h(x)$ con $h(x)$ especificado previamente es muy difícil.
b) Funciones Hash a prueba de colisiones (CRHFs): para este tipo de funciones, encontrar dos paramentos de entrada tales que su imagen bajo la función hash es la misma es muy difícil.

2) Códigos de Verificación de Mensajes (MACs).
El propósito de un MAC es facilitar, sin ayuda de ningún otro mecanismo, la verificación del origen de los mensajes y la integridad de los mismos. Los MACs requieren dos parámetros de entrada; un mensaje y una llave secreta. Son una subclase de las funciones hash con-llave.

La siguiente figura ilustra de una forma más simple la clasificación.

Figura 5.1: Clasificación Funciones Hash.



Generalmente suponemos que las especificaciones algorítmicas de la función hash son de dominio público. Por lo tanto en el caso de los MDCs, dado un mensaje cualquiera puede calcular el valor hash. Mientras que para los MACs, dado un mensaje, cualquiera con conocimiento de la llave privada puede calcular el valor hash para el mensaje.

5.1.1 Definiciones y Propiedades Básicas.

Para facilitar las definiciones subsecuentes daremos tres propiedades potenciales (además de las ya mencionadas en la definición de arriba) para una función hash sin llave h con parámetros de entrada $x, \tilde{x}$ y parámetros de salida $y, \tilde{y}$.

1) Resistencia a Preimagen; Esencialmente para todos los parámetros de salida especificados previamente es computacionalmente imposible encontrar parámetros de entrada que bajo la función hash vayan a dar a los parámetros especificados previamente. Es decir, encontrar una preimagen $\tilde{x}$ tal que $h(\tilde{x}) = y$, donde y es fija y especificada previamente, es computacionalmente imposible.

2) Resistencia a Segunda Preimagen; Es computacionalmente imposible encontrar un segundo parámetro de entrada que tenga el mismo valor bajo la función hash que el parámetro de entrada especificado. Es decir dada x, encontrar una segunda preimagen $\tilde{x}$ distinta de x tal que $h(x) = h(\tilde{x})$ es imposible.

3) Resistencia a Colisiones; Es computacionalmente imposible encontrar cualesquiera dos parámetros de entrada $x, \tilde{x}$ que tengan la misma imagen bajo la función hash, es decir, tales que $h(x) = h(\tilde{x})$ (notemos que en este caso haya libertad de elección para ambos parámetros de entrada $x, \tilde{x}$.).

En este capítulo hemos estado haciendo uso de los términos fácil y computacionalmente imposible (difícil) de una forma recurrente sin especificar a qué se refieren. Hagámoslo ahora; con fácil nos referimos a tiempo y/o espacio polinomial, dicho de una forma más práctica; dentro de un cierto número de operaciones de máquina o unidades de tiempo, posiblemente segundos o milisegundos. Una definición más específica de computacionalmente imposible involucra esfuerzo súper-polinomial, y requeriría un esfuerzo que excede la capacidad de nuestros recursos computacionales y/o matemáticos actuales. Una vez aclarados estos conceptos y considerando las propiedades recientemente definidas podemos dar unas cuantas definiciones con un enfoque más práctico y acercado al contexto que nos interesa.

Con el objetivo de ilustrar y conectar los dos temas principales de este proyecto, analicemos ahora un ejemplo de una de las propiedades arriba definidas dentro del contexto de las firmas digitales. Consideremos un esquema de firmas digitales en donde el algoritmo de generación de firmas es aplicado al valor hash $h(x)$ en vez de al mensaje x. En este caso h debería ser un MDC con resistencia a segunda preimagen, de otra forma un adversario C podra observar la firma de alguna entidad A en $h(x)$, y entonces encontraría $\tilde{x}$ tal que $h(x) = h(\tilde{x})$, falsificando la firma de A para $h(\tilde{x})$. Si le

damos al adversario C la capacidad de elegir el mensaje que A firma entonces sólo necesitaría encontrar la pareja de colisión $(x, \tilde{x})$ en vez de la más difícil tarea de encontrar una segunda preimagen de x; en este caso, la resistencia a colisiones es también necesaria. El requerimiento de resistencia a preimagen es menos evidente para algunos esquemas de firmas digitales; consideremos por ejemplo RSA, donde la entidad A tiene una llave pública (e, n). Un adversario C puede escoger un valor aleatorio y, calcular $z = y^e (mod\ n)$ y (dependiendo del proceso de verificación de firmas RSA que está siendo utilizado) asegurar que y es la firma de A para z. Esta falsificación (existencial) debe ser considerada si C encuentra una preimagen x tal que $h(x) = z$, para la cual x es de uso práctico.

Definición 5.1.2. *Una Función Hash de un Solo Sentido (OWHF, del inglés: One Way Hash Function) es una función hash que cumple las propiedades de la definición 5.1.1 y además cumple las siguientes propiedades: resistencia a preimagen y resistencia a segunda preimagen.*

Definición 5.1.3. *Una Función Hash a Prueba de Colisiones (CRHF, del inglés: Collision Resistant Hash Function) es una función hash que cumple las propiedades de la definición 5.1.1 y además cumple las propiedades: resistencia a segunda preimagen y resistencia a colisiones.*

En la práctica, las CRHF casi siempre tienen la propiedad de resistencia a preimagen, con el objetivo de asegurarnos de que no haya vulnerabilidades en el esquema que se esté implementando. Notemos que las CRHF también son llamadas Funciones Hash Fuertes de un Solo Sentido y las OWHF son llamadas Funciones Hash Débiles de un Solo Sentido en otros textos.

Observemos que para cada elección de función hash h, sin importar cual sea el proceso de selección, existe un conjunto distinto del vacío de llaves cuyas imágenes bajo h son todas iguales. Esto podemos asegurarlo por la paradoja del cumpleaños, que nos garantiza la existencia de colisiones. Al estar garantizada la posibilidad de colisiones para cualquier elección de función hash surge la necesidad de fortalecer el desempeño del mecanismo de verificación de identidad y en particular de la función h. Centraremos nuestra atención en elegir la función h (independiente de las llaves) aleatoriamente

de una familia de funciones hash. Convirtiendo la elección aleatoria de h en el primer punto a vencer para el adversario.

5.2 Universal Hashing

La técnica Universal Hashing (UH) fue introducida por Carter y Wegman en 1979, y se ha convertido en una herramienta esencial en varias áreas de la computación como la generación de números pseudo-aleatorios y la amplificación de la privacidad. Las familias hash universales están fuertemente relacionadas con estructuras combinatorias tales como los arreglos ortogonales y los códigos correctores de errores. En esta tesis construiremos una familia que estará relacionada al código Reed-Solomon. UH es un método para escoger una función hash h de forma aleatoria de una familia de funciones hash H. La familia H estará determinada por un paradigma matemático en específico de tal forma que cada función pueda ser eficientemente evaluada con un respaldo abstracto sólido. Esta elección aleatoria actúa como un amplificador de seguridad, ya que, aunque el adversario tenga acceso a los algoritmos de implementación del esquema, no podrá adivinar colisiones debido a que en primera instancia no será capaz de saber qué función hash h se está utilizando en el momento de la ejecución.
Existen distintos tipos de familias de funciones hash, las cuales varían en método de construcción y respaldo abstracto. En esta tesis nos centraremos en familias construidas sobre campos de orden 2^n, en específico en la evaluación de polinomios con coeficientes en GF(2,n). Antes de pasar a la construcción central daremos unas cuantas definiciones de familias de funciones hash.

5.2.1 Familias de Funciones Hash.

Definición. 5.1. Una Familia Hash-$(N; n, m)$ es un multiconjunto F que consta de N funciones $f : X \to Y$, donde $|X| = n$ y $|Y| = m$, $n > m$.

Definición. 5.2. Una Familia Hash-$(N; n, m)$ es $\epsilon - universal$ para alguna $0 < \epsilon < 1$ si para cualesquiera dos elementos distintos $x_1, x_2 \in X$ el número de funciones $f \in F$ tales que $f(x_1) = f(x_2)$ satisface $v/n \leq \epsilon$.

Esta última definición la podemos interpretar probabilísticamente de la siguiente forma: cuando tomamos dos elementos en el conjunto base $x_1, x_2 \in X$ y una $f \in F$ es dada aleatoriamente de acuerdo a una distribución uniforme, entonces la probabilidad de que los valores coincidan es menor o igual que Epsilon, *ie.* $Pr[f(x_1) = f(x_2)] \leq \epsilon$. Podemos interpretar estas familias de funciones hash como arreglos de la siguiente forma: etiquetamos las columnas con los elementos (en nuestro caso; polinomios en GF(2,n)[X]) en el conjunto base X e interpretamos cada hilera como una función del conjunto X al conjunto de todas las entradas (evaluación de $f \in X$ en cada uno de los elementos del espacio base GL(2,n)).

Definición. 5.3. Una Familia de funciones Hash-$(N; n, m)$ es $\epsilon-fuertemente\ universal$ para alguna $0 < \epsilon < 1$ si se satisface lo siguiente:

a) Para cualesquiera $x \in X$ y $y \in Y$ el número v de funciones $f \in F$ tales que $f(x) = y$ es precisamente N/m.

b) Para cualesquiera dos elementos distintos $x_1, x_2 \in X$ y para cualesquiera dos (no necesariamente distintos) elementos $y_1, y_2 \in Y$ el número v de funciones $f \in F$ tales que $f(x_1) = y_1, f(x_2) = y_2$ satisface $v/(N/m) \leq \epsilon$.

La primera condición nos dice que en el arreglo correspondiente a cada entrada ocurre con la misma frecuencia en cada columna. Equivalentemente podríamos decir que el arreglo asociado a la familia es un arreglo ortogonal de fuerza uno. Verlo desde este punto de vista nos facilita la interpretación probabilística de la estructura: sabemos que existen exactamente N/m funciones f que satisfacen $f(x_1) = y_1$. Fijémonos únicamente en este tipo de funciones, escojamos una aleatoriamente. La segunda condición nos dice que la probabilidad de encontrar una función f que satisfaga $f(x_2) = y_2$ es menor o igual que Epsilon. Esto nos dice que Epsilon es una cota superior en una probabilidad condicional: $Pr[f(x_2) = y_2 | f(x_1) = y_1] \leq \epsilon$.

5.2.2 Verificación de identidad.

Sea A una familia de funciones hash $\epsilon-$fuertemente universal. En esta sección explicaré cómo utilizaremos a A como un sistema de verificación de identidad para poder crear nuestro objetivo final: la implemantación de un esquema

de firmas digitales para iOS.
Identifiquemos a A con su arreglo correspondiente, como dijimos en la sección anterior. Supongamos que una entidad origen O (remitente) produce mensajes para ser enviados a través de un canal público a un receptor P remoto. Con el objetivo de poder distinguir entre los mensajes que fueron producidos por O y los mensajes enviados por el canal (que puedan haber sido cifrados o modificados por algún adversario), denotamos a los mensajes producidos por O como "estados origen" y a los ya enviados por el canal de acceso público como mensajes.

Nuestra principal preocupación será la verificación de identidad. Medidas como la integridad (proteger el conocimiento de cuál estado origen está siento enviado) o corrección de errores (el canal puede ser ruidoso) pasarán a segundo plano en el contexto teórico de esta tesis. Notemos aquí que para garantizar el óptimo funcionamiento del esquema de firmas que implementaremos, consideraremos las tres medidas empezando por la verificación. Queremos desarrollar esquemas de verificación de identidad que le permitan al receptor verificar si el mensaje realmente se originó del remitente que dice haberlo enviado o si ha sido alterado en el camino. Aquí es donde las propiedades que definen a las familias hash fuertemente universales (familias $\epsilon-$FU) jugarán un papel muy importante. Utilizaremos los elementos $f \in A$ como llaves, los elementos $y \in Y$ son también llamados verificadores.

Supongamos que $x \in X$ es el estado origen que va a ser transmitido. Receptor y Remitente se pondrán de acuerdo (a través de un canal seguro y privado) en una llave $f \in A$ a ser usada. Entonces el mensaje $x \in X$ (posiblemente cifrado o codificado con propósitos de corrección de errores) contendrá la Etiqueta de Verificación $f(x) = y$. Al recibir el mensaje que decodifica a $\tilde{x} \in X$, con etiqueta de verificación $\tilde{y} \in Y$, el receptor lo acepta como auténtico si y sólo si $f(\tilde{x}) = \tilde{y}$. Observemos que si no hubo problemas tenemos; $\tilde{x} = x$, $\tilde{y} = y = f(x)$, y el mensaje sería aceptado como auténtico.

5.2.3 Ataques a Familias de Funciones Hash $\epsilon - FU$.

Para entender a fondo el porqué utilizamos una familia de funciones hash $\epsilon - FU$, y el significado de ϵ imaginemos un ataque. Imaginemos que un oponente tiene acceso al canal de comunicación. Su objetivo es desequilibrar el sistema, y más precisamente: lograr que mensajes no auténticos

(falsificados) sean aceptados como auténticos. Los resultados de este ataque dependerían claramente en las capacidades de el adversario. A continuación consideraremos dos tipos de ataques.

Ataque de falsificación de identidad.

Daremos por hecho siempre que el adversario conoce el sistema. Su único problema es que no tiene acceso al canal seguro, y, por lo tanto, no tiene ningún conocimiento a *priori* de la llave que está siendo usada. Supongamos que el oponente forma un mensaje y se lo envía al receptor asegurando que el mensaje fue originado por el remitente esperado. El adversario tratará de maximizar las posibilidades de que el mensaje sea aceptado como válido. Esto es un ataque de falsificación de identidad (descrito en el capítulo pasado, "ataques a firmas digitales"). La primera propiedad de la definicin 5.3 nos dice que la probabilidad de que el adversario tenga éxito es de $1/m$, donde $m = |Y|$ es el número de verificadores.

Ataque de substitución.

Supongamos ahora que el adversario tiene la capacidad no sólo de leer los mensajes si no también de alterarlos y de enviar la versión alterada al receptor. El conocimiento del mensaje le da algo de información de la llave usada para crear la etiqueta. Supongamos que el adversario conoce el estado origen $x \in X$ de un mensaje m, entonces la llave secreta es una de las funciones $f \in A$ que satisfacen $f(x) = y$.

Nuestro adversario desea hacer que el receptor acepte como válido un mensaje distinto de m, con estado origen correspondiente $\tilde{x} \in X$ tal que x es distinto de $\tilde{x}$. Tiene que escoger una etiqueta de verificación $\tilde{y}$ tal que sea altamente probable que pase; $f(\tilde{x}) = \tilde{y}$. Sin embargo la segunda propiedad de la Definición 5.3 nos dice que la probabilidad de éxito será siempre menor o igual que Epsilon. Observemos que cuando $\tilde{x} \neq x$ ha sido escogida para formar la etiqueta, entonces la probabilidad de que el adversario tenga éxito puede ser escrita como una probabilidad condicional de la siguiente forma: $Pr[f(\tilde{x}) = y | f(x) = y]$ donde $x, \tilde{x}, y, \tilde{y}$ están dadas, $(x \neq \tilde{x})$ y la probabilidad se refiere a al elección de $f \in A$.

El problema de eficiencia.

En la subsección anterior vimos como una familia de funciones hash $\epsilon - FU$ puede ser interpretada como un sistema de verificación de identidad con la propiedad de que la probabilidad de éxito de un adversario bajo un ataque de substitución esté limitada por ϵ. Debemos observar que para cada transmisión de estados origen una nueva llave tiene que ser generada aleatoriamente y esta información enviada al receptor previamente a través del canal seguro.
Para generar un sistema que pueda ser utilizado en práctica, necesitamos limitar la cantidad de tiempo y espacio requeridos por la verificación. Por ejemplo, si hay $N = 2^{100}$ llaves y $m = 2^{40}$, entonces la etiqueta de verificación debe ser de longitud en bits 40, la información de la llave a ser enviada por el canal seguro será de longitud 100 bits. En práctica no tendría mucho sentido usar dicho sistema cuando el número de mensajes es $n = |X| = 2^{100}$. No podemos arriesgarnos a gastar la mitad del tiempo de trasmisión únicamente en propósitos de verificación, esto nos lleva al problema de eficiencia.
Los parámetros n (número de estados origen) y ϵ (nivel de seguridad) pueden ser pensados como dados. Queremos construir sistemas de verificación de identidad o equivalentemente familias hash $\epsilon - FU$, tales que el número N de llaves y el número m de verificadores sean lo más pequeños posible. La medida más razonable para lograr esto pareciera ser $log_2(N) + log_2(m) = log_2(Nm)$, éste es el parámetro que deseamos minimizar cuando n y ϵ están dados.

5.2.4 Un poco mas de teoría.

En esta sección interpretaremos a las familias hash $\epsilon - FU$ desde el punto de vista de la teoría de códigos y discutiremos sus métodos de construcción. Resultará que las familias hash $\epsilon - U$ son códigos en realidad, más aún los códigos son ingredientes importantes en la construcción de las familias hash $\epsilon - FU$, y estas últimas son generalizaciones de los arreglos ortogonales de fuerza 2.

Consideremos la representación en forma de arreglo de una familia hash $\epsilon - U$ A. Pensemos las columnas como palabras de un código $m - ario$ de longitud N. Sean $\tilde{x}, x$ dos palabras distintas, entonces la definición 5.2 (Fam Hash $\epsilon - U$) puede ser interpretada en términos de la distancia de Hamming

$d(x, \tilde{x})$. En teoría de códigos esto es $1 - \epsilon \leq d(x, \tilde{x})/N$. Si d es la distancia mínima de nuestro código entonces la propiedad que define una familia hash $\epsilon - U$ puede ser equivalentemente expresada como $1 - \epsilon \leq d/N$, donde N es la longitud del código. En teoría de códigos la expresión d/N es conocida como la distancia mínima relativa de un código. El teorema que sigue es consecuencia natural de este razonamiento:

Teorema 5.2.1. *Sea A una Familia Hash-(N; n, m) como en la definición 5.1. Entonces los siguientes puntos son equivalentes:*

a) A es un familia hash $\epsilon - U$.

b) Las columnas de A forman la palabras de un código $m - ario$ *de longitud N con distancia mínima relativa* $1 - \epsilon \leq d/N$.

Debe ahora estar claro que las familias hash $\epsilon - U$ son códigos disfrazados. Cualquier aplicación de una familia hash $\epsilon - U$ es una aplicación de la teora de códigos. Analicemos ahora la estructura de las familias hash $\epsilon - FU$. Como la propiedad que las define es en términos de parejas de columnas del arreglo en cuestión pareciera natural querer compararlos con los arreglos ortogonales de fuerza 2, veamos si es posible.

Lema 5.2.1. *Sea A una familia hash-(N; n, m) que es* $\epsilon - FU$. *Entonces* $1/m \leq \epsilon$. *La igualdad se da si y sólo si A es un arreglo ortogonal de fuerza 2.*

Demostración: Sean x, y y $\tilde{x}$ dados como antes. Mientras $\tilde{y}$ varía sobre el $m - conjunto\ Y$ notamos que la máxima de las propiedades condicionales $Pr[f(\tilde{x}) = \tilde{y} | f(x) = y]$ debe ser mayor o igual que $1/m$. Se sigue que $1/m \leq \epsilon$. En caso de que se dé la igualdad debe ser el caso en el que para todos $x, y, \tilde{x}, \tilde{y}$ como antes $(x, \tilde{x} \in X, x \neq \tilde{x};\ y, \tilde{y} \in Y)$ el número de funciones f que satisfacen $f(x) = y, f(\tilde{x}) = \tilde{y}$ debe ser una constante, digamos λ. Comparando con la definición de arreglo ortogonal (apéndice) vemos que llegamos a lo que queríamos.

Lema 5.2.2. *Sea A una* $familia\ hash - (N; n, m)$ *que es* $\epsilon - FU$. *Entonces A es* $\epsilon - U$.

Dada su simplicidad está de sobra hacer la demostración de este Lema. Se sigue que las familias hash $\epsilon - FU$ satisfacen condiciones más fuertes que las familias $\epsilon - U$. El Lema 5.1 nos muestra que es posible utilizar los Arreglos Ortogonales de fuerza 2 como esquemas de verificación. De cualquier forma nos encontramos con un problema de eficiencia. La cota de Bose-Bush nos dice que el número N de filas (llaves) ciertamente excede el número n de columnas (estados origen). Esto nos muestra que los AO de fuerza dos son extremadamente ineficientes cuando son utilizados como esquemas de verificación. A la brevedad describiremos a detalle la forma más eficiente para construir familias hash $\epsilon - FU$.

Teorema 5.2.2. *(Composición de Stinson) Sea A_1 una familia hash$-(N_1; n, \tilde{n}$ que es $\epsilon_1 - U$ y A_2 una familia hash $- (N_2; \tilde{n}, m)$ que es $\epsilon_2 - U$. La composición $A = A_2 \circ A_1$ está definida como el multiconjunto de composiciones de funciones $f = f_2 \circ f_1$ donde $f_i \in A_i$. Entonces A es una Familia Hash$-(N_1N_2; n, m)$ que es $\epsilon - FU$, donde $\epsilon = \epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2$*

Demostración: Como A_i es un multiconjunto de N_i funciones $(i = 1, 2)$ está claro que el multiconjunto A de todas las composiciones consiste de N_1N_2 funciones. Tenemos que A es una familia hash-$(N_1N_2; n, m)$, quedando la primera condición de la definición 5.3 satisfecha. Para demostrar la condición principal haremos uso del lenguaje de la probabilidad. Sean x_1, x_2 en el conjunto base $(x_1 \neq x_2)$ y z_1, z_2 en el $m - conjunto$. Escribamos $f \in F$ como $f = f_2 \circ f_1$, sea $y_1 = f_1(x_1), y_2 = f_1(x_2)$ y $P = Prob(f(x_1) = z_1 | f(x_2) = z_2)$. Queremos mostrar que $P \leq \epsilon_1 + \epsilon_2 - \epsilon_1\epsilon_2$. Consideremos primero el caso en que $z_1 \neq z_2$. Tenemos que $P \leq Prob(y_1 \neq y_2)\epsilon_2 < \epsilon_2$. El caso $z_1 = z_2$ es un poco más difícil, veamos: De las hipótesis tenemos que $P \leq Prob(y_1 \neq y_2)\epsilon_2 + Prob(y_1 = y_2)$. Sea $\alpha = Prob(y_1 = y_2)$, lo anterior se reduce a $P \leq \alpha + (1 - \alpha)\epsilon_2$, lo cual denotaremos por $h(\alpha)$. Como $\tilde{h}(\alpha) = 0 \leq 1 - \epsilon_2$ y $\alpha \leq \epsilon_1$ por la propiedad que defina a A_1 obtenemos $P \leq h(\epsilon_1)$ que es lo que asegurábamos.

En la sección que sigue mostraremos que esta construcción puede ser utilizada para obtener códigos de verificación hasta cierto punto eficientes. Nos encontraremos con el hecho de que si incrementamos el parámetro de seguridad ϵ solamente un poco sobre su mínimo teórico, obtendremos familias hash $\epsilon - FU$ tales que su número de elementos (llaves) es dramáticamente más

AO para Familia de Funciones Hash					
	$p_1(X)$	$\ldots$	$p_i(X)$	$\ldots$	$p_{\alpha k}(X)$
(u_1, v_1)	$\phi[p_1(u_1)] + v_1$	$\ldots$	$\phi[p_i(u_1)] + v_1$	$\ldots$	$\phi[p_{\alpha k}(u_1)] + v_1$
(u_1, v_2)	$\phi[p_1(u_1)] + v_2$	$\ldots$	$\phi[p_i(u_1)] + v_2$	$\ldots$	$\phi[p_{\alpha k}(u_1)] + v_2$
(u_1, v_3)	$\phi[p_1(u_1)] + v_3$	$\ldots$	$\phi[p_i(u_1)] + v_3$	$\ldots$	$\phi[p_{\alpha k}(u_1)] + v_3$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\iddots$	$\vdots$
(u_i, v_j)	$\phi[p_1(u_i)] + v_j$	$\ldots$	$\phi[p_i(u_i)] + v_j$	$\ldots$	$\phi[p_{\alpha k}(u_i)] + v_j$
$\vdots$	$\vdots$	$\iddots$	$\vdots$	$\ddots$	$\vdots$
$(u_\alpha, v_{\beta-1})$	$\phi[p_1(u_\alpha)] + v_{\beta-1}$	$\ldots$	$\phi[p_i(u_\alpha)] + v_{\beta-1}$	$\ldots$	$\phi[p_{\alpha k}(u_\alpha)] + v_{\beta-1}$
(u_α, v_β)	$\phi[p_1(u_\alpha)] + v_\beta$	$\ldots$	$\phi[p_i(u_\alpha)] + v_\beta$	$\ldots$	$\phi[p_{\alpha k}(u_\alpha)] + v_\beta$

Tabla 5.1: Construcción de Familia de Funciones Hash

chico que los estándares teóricos. Este efecto es denominado Efecto Wegman & Carter.

5.2.5 Construcción de Familias de Funciones Hash Fuertemente Universales.

Teorema 5.2.3. *Sea q una potencia de un número primo, sean k, α, β números naturales tales que $\beta \leq \alpha$. Entonces existe una familia de funciones hash $(q^{\alpha+\beta}; q^{\alpha k}, q^{\beta})$, que es k/q^{β} Fuertemente Universal.*

Demostración: Sea $\phi : F_{q^\alpha} \to F_{q^\beta}$ una función lineal . Construyamos un arreglo A con $q^{\alpha k}$ columnas, ($q^{\alpha+\beta}$ renglones y con entradas en F_{q^β}. Las columnas están etiquetadas por los polinomios $p(X) \in F_{q^\alpha}(X)$ con coeficientes en F_{q^α}, de grado menor o igual que k y con término constante 0. Los renglones están etiquetadas por las parejas (u, v) donde $u \in F_{q^\alpha}, v \in F_{q^\beta}$. La entrada que corresponde a la columna $p(X)$ y al renglón (u, v) es; $\phi(p(u))+v$.

Debe quedar claro que cada columna de A contiene cada elemento de F_{q^β} con frecuencia q^α. Fijemos dos columnas distintas $p_1(X), p_2(X)$ y dos elementos no necesariamente distintos $y_1, y_2 \in F_{q^\beta}$. Contemos los renglones (u, v) que satisfacen lo siguiente;
$\phi(p_1(u)) + v = y_1$ y $\phi(p_2(u)) + v = y_2$

Restando y aplicando la linealidad de ϕ nos queda;
$\phi((p_1 - p_2)(u)) + v = y_1 - y_2$
Cada solución u de la ecuación de arriba determina de forma única un renglón (u, v). Consideremos ahora $M = \phi^{-1}(\alpha_1 - \alpha_2)$. Tenemos que el orden de M es $q^{\alpha-\beta}$. Observemos que $p_1(X) - p_2(X)$ es un polinomio no constante de grado menor o igual que k, de aquí se sigue que el número de soluciones es menor o igual que kq^{n-m}. Justo como queríamos. QED

Consideremos el caso especial $k = 1$ en el teorema 5.3. Esto implicaría que $\epsilon = q^{-\beta} = 1/m$. Como observamos en el lema 5.1 esto caracteriza a los arreglos ortogonales de fuerza 2.

Corolario 5.2.1. *El caso especial $k = 1$ en el teorema 5.3 resulta en Arreglos Ortogonales con parámetros $OA(2, q^\alpha, q^\beta)$.*

Apliquemos la composición de Stinson (teorema 5.2) con los Arreglos Ortogonales del corolario anterior (5.1) en el lugar de A_2. Sabemos de resultados previos en el capítulo que si A_1 es una $familia\ hash\ \epsilon_1 - U$ entonces es equivalente a un código corrector de errores. Utilicemos códigos de Reed-Solomon.

Lema 5.2.3. *El código de Reed-Solomon RS(k, q) como en la definición dada en el apéndice , resulta en una $familia\ hash - (q; q^k, q)$ que es $k - 1/q$ $universal$.*

Demostración; Fijémonos en el teorema 5.1, es claro que los parámetros N, n, m asociados a RS(k,q) satisfacen lo pedido por el teorema. Como el código de Reed-Solomon tiene distancia mínima $d = q - (k - 1)$ se sigue que $\epsilon = 1 - d/q = (k - 1)/q$. QED

La composición de Stinson usando códigos de Reed-Solomon y AO del corolario 5.1 como ingredientes, nos resulta en familias interesantes de funciones hash fuertemente universales.

Teorema 5.2.4. *Sea q una potencia de un número primo, sean k, α, β números naturales tales que $\beta \leq \alpha$. Entonces existe una familia de fun-*

ciones hash $(q^{2\alpha+\beta}; q^{\alpha q^{\alpha-\beta}}, q^{\beta})$*, que es* $2q^{-\beta}FU$*.*

Demostración; $RS(q^{\alpha-\beta}, q^{\alpha})$ nos da una familia hash-$(q^{\alpha}; q^{\alpha q^{\alpha-\beta}}, q^{\alpha})$, la cual es $\epsilon_1 - U$. En este caso $\epsilon_1 = (q^{\alpha-\beta} - 1/q^{\alpha}) < q^{-\beta}$. Un arreglo ortogonal del corolario 5.1 nos da una $familia\ hash - (q^{\alpha+\beta}, q^{\alpha}, q^{\beta})$ la cual es $q^{-\beta} - U$. La composición de ambas familias tiene $q^{2\alpha+\beta}$ elementos. El valor de para la familia de funciones fuertemente universal es $\epsilon \leq \epsilon_1 + q^{-\beta} < 2q^{-\beta}$. QED.

Veamos un ejemplo para ilustrar esta construcción:

Ejemplo. 5.1. Sean $q = 2, \alpha = 32, \beta = 20$. Aplicando el teorema 5.4 obtenemos una familia hash-$(2^{84}; 2^{2^{17}}, 2^{20})$, la cual es $2^{-19} - FU$. En términos de Verificación (Sección 5.2.3) es claro que podemos verificar estados origen de 2^{17} bits con 20 bits de verificacin y 84 bits de llave tal que la probabilidad de ruptura esta limitada a 2^{-19}. Esto ilustra lo dramático que es el efecto Carter-Wegman. Si insistiéramos en una probabilidad de ruptura de 2^{-20}, utilizando también 20 bits de verificación, entonces nos hubiéramos visto forzados a usar un AO de fuerza 2 por el Lema 5.2.1, llevándonos a más de 2^{17} bits de llave debido al efecto de Bose-Bush como mencionamos previamente. En el capítulo siguiente veremos cómo fue construido el esquema de firmas digitales desde la teoría de campos finitos.

Capítulo 6

Implementación del Esquema de Firmas.

En este capítulo veremos a detalle cómo desarrollamos los algoritmos y la implemantación del Esquema de Firmas Digitales.

6.1 Contexto de Programación

Nuestro objetivo es hacer que nuestro esquema de firmas digitales sea ejecutable en un dispositivo móvil Apple iPhone, iPad con sistema operativo iOS. Para llevar a cabo esto utilizaré el lenguaje de programación nativo de los dispositivos Apple: Objective C, y el paradigma de programación orientada a objetos (POO). Los programas y aplicaciones fueron desarrollados utilizando el compilador X-code el cual es nativo del sistema operativo Mac OS X Lion 10.7.4 (11E53), en una MacBook Pro (Processor 2.4 GHz Intel Core 2 Duo y Memoria de 4 GB 1067 MHz DDR3).

Para diseñar y después implementar este esquema hay que darle prioridad al hecho de que para crear las firmas utilizamos Campos finitos de la forma $\mathbb{F}_{2^n}$ y extensiones finitas (en el sentido matemático, ver apéndice) de ellos. Lo cual como matemático me lleva a querer programar los espacios en su totalidad así como los espacios de sus extensiones completos respectivamente. Dada la complejidad de los espacios y de las interacciones de los elementos que en ellos habitan es claro que un paradigma de programación ordinario sería de poca ayuda, por lo tanto, decidí desarrollar este software utilizando

un paradigma de programación orientado a objetos (POO). El cual me permitirá definir clases (en el sentido de programación, con archivo de cabecera, .h y archivo de implementación, .m) cuyos atributos y métodos le darán sus características al espacio que estoy definiendo, creando así una "blue print" de los elementos del espacio. Así depués podremos crear instancias de objetos que nos servirán como representantes de los elementos del espacio que describimos al definir la clase.
Una vez definidas las clases que describen los campos y anillos que necesito, podré emular la interacción de elementos que habitan en las estructuras matemáticas que estamos considerando en mi compilador, por ejemplo: la evaluación de un polinomio en un operador polinomial, en cuyo caso el primero vive en el campo finito $\mathbb{F}_{2^n}$ y el segundo habita el anillo que resulta de considerar el anillo de polinomios de $\mathbb{F}_{2^n}$ en la variable Y. Una vez que el compilador tenga bien definida la representación de las entidades matemátias requeridas podremos obtener datos y resultados prácticos de operaciones aritméticas en los campos finitos $\mathbb{F}_{2^n}$ y sus extensiones. En la sección siguiente veremos a detalle cómo definimos dicha representación.

6.2 Enteros, bits y polinomios

Nuestro objetivo es crear una representación computacional de los campos $\mathbb{F}_{2^n}$ y sus respectivas extensiones utilizando un paradigma de programación orientado a objetos. Los elementos de los ya mencionados campos son polinomios con coeficientes 0 y 1 y grado menor al del polinomio irreducible que usamos para obtener las clases de equivalencia que determinan el campo. Podemos asociar de manera natural cada uno de estos polinomios con un vector que habita un espacio vectorial de dimensión n, proyectando los valores booleanos de los coeficientes de los polinomios ordenadamente sobre las entradas de los vectores de derecha a izquierda, es decir: el valor del monomio de menor grado de nuestro polinomio lo copiaremos en la primer entrada del vector comenzando desde la derecha (Bit-menos importante) y así sucesivamente hasta terminar con todos los monomios. Veamos esto de una forma más precisa y abstracta. Consideremos el morfismo de proyección:

$$P : \mathbb{F}_{2^n} \to \mathbb{Z}_2^n,$$

donde $P(p_0) = v$ tal que; $p_0 \in \mathbb{F}_{2^n}$, $v \in \mathbb{Z}_2^n$.

Para ilustrar esto consideremos $p_0 = a_0 + a_1x^1 + a_2x^2 + a_3x^3 \in \mathbb{F}_2^4$ entonces a $P(p_0)$ es el vector $(a_3, a_2, a_1, a_0) \in Z_2^n$ con $a_i \in \mathbb{Z}_2$. El morfismo de proyección P proyecta los coeficientes de los elementos de $\mathbb{F}_{2^n}$ a las entradas de los elementos de $\mathbb{Z}_2^n$ comenzando de derecha a izquierda con el objetivo de entenderlos como bits computacionales. La dimensión del espacio en donde habitan los vectores está determinada por el grado del polinomio irreducible que determina el campo, en este caso $n = 4$. Dada la simplicidad de la regla de correspondencia de P basta con definir P^{-1} y ver que efectivamente son inversas.

$P^{-1} : \mathbb{Z}_2^n \to \mathbb{F}_{2^n}$ tal que $P^{-1}(v) = p_0$ donde $p_0 \in \mathbb{F}_{2^n}$, $v \in \mathbb{Z}_2^n$

Consideremos; $(a_3, a_2, a_1, a_0) \in \mathbb{Z}_2^n$, entonces si lo mandamos con P^{-1} obtenemos $P^{-1}(v) = a_0 + a_1x^1 + a_2x^2 + a_3x^3$. Este morfismo proyecta las coordenadas del espacio de bits sobre los coeficientes de los elementos del campo finito $\mathbb{F}_{2^n}$. Notemos que $P^{-1}(P(p_0)) = p_0$ por definición, entonces P es un morfismo biyectivo. Observemos que el morfismo de proyección P y su inversa nos proporcionan una forma de movernos del espacio de bits de longitud n ($\mathbb{Z}_2^n$) a el campo finito $\mathbb{F}_{2^n}$, lo que me permitirá llevar y aplicar la teoría de campos finitos y grupos a la arquitectura computacional binaria, dándome así la libertad de jugar con los conceptos y esquemas criptográficos que se me ocurran. Observemos ahora que podemos enumerar los conjuntos de vectores y polinomios de diferentes formas. La enumeración que sigue es la asociada al método para construir el campo finito $\mathbb{F}_{2^n}$. En las tablas 6.1 y 6.2 ilustramos el caso $n = 3$, nos resulta en tres conjuntos de cardinalidad $8 = 2^3$:

La relación que vemos en las tablas 6.1 y 6.2 es la que utiliza la computadora para traducir los bits a números enteros y a polinomios en un campo finito de característica dos. Consideremos nuevamente el morfismo de proyección P y tomemos su inversa P^{-1}, este morfismo nos permitirá pensar los bits como polinomios y por lo tanto aplicar nuestra teoría de campos finitos a los bits computacionales. Ahora operar enteros, polinomio o vectores será exactamente lo mismo, sólo nos falta decir quienes son las operaciones de grupo asociadas al campo. La suma será el operador binario XOR y la multiplicación será una iteración del operador binario AND. En la sección que sigue construiremos la clase que representará el anillo de polínomios con

$\mathbb{Z}$ <> $\mathbb{F}_{2^3}$ <> $\mathbb{Z}_2^3$		
0	0	(0,0,0)
1	1	(0,0,1)
2	x	(0,1,0)
3	x + 1	(0,1,1)
4	x^2	(1,0,0)
5	$x^2 + 1$	(1,0,1)
6	$x^2 + x$	(1,1,0)
7	$x^2 + x + 1$	(1,1,1)

Tabla 6.1: Relación entre $\mathbb{Z}$ <> $\mathbb{F}_{2^3}$ <> $\mathbb{Z}_2^3$

$\mathbb{Z}$ <> $\mathbb{F}_{2^4}$ <> $\mathbb{Z}_2^4$		
0	0	(0,0,0,0)
1	1	(0,0,0,1)
2	x	(0,0,1,0)
3	x + 1	(0,0,1,1)
4	x^2	(0,1,0,0)
5	$x^2 + 1$	(0,1,1,0)
6	$x^2 + x$	(0,1,0,1)
7	$x^2 + x + 1$	(0,1,1,1)
8	x^3	(1,0,0,0)
9	$x^3 + 1$	(1,0,0,1)
10	$x^3 + x$	(1,0,1,0)
11	$x^3 + x + 1$	(1,0,1,1)
12	$x^3 + x^2$	(1,1,0,0)
13	$x^3 + x^2 + 1$	(1,1,0,1)
14	$x^3 + x^2 + x$	(1,1,1,0)
15	$x^3 + x^2 + x + 1$	(1,1,1,1)

Tabla 6.2: Relación entre $\mathbb{Z}$ <> $\mathbb{F}_{2^4}$ <> $\mathbb{Z}_4^4$

coeficientes en $\mathbb{Z}_2$. Los elementos del anillo serán instancias (objetos) de la clase previamente definida.

6.3 La clase del anillo de polinomios $\mathbb{Z}_2[x]$

Primero que nada tenemos que construir la clase base, esto es, la clase de la cual todas las otras clases que creemos heredarán las operaciones polinomiales que vamos a definir. Crearemos la clase PolynomialClass.h/.m la cual constará de atributos y propiedades tales que, al instanciar un objeto de ella, obtendremos la representación abstracta de un polinómio con coeficientes binarios. La elección de $\mathbb{Z}_2$ para ser el campo en el que habitan los coeficientes de nuestros polinomios es clara, ya que los bits computacionales consisten de agrupaciones y permutaciones de los elementos de $\mathbb{Z}_2$, entonces se vuelve natural (como vimos en la sección anterior) relacionar ordenadamente los coeficientes de mis polinomios con las entradas de los bits comenzando por el más significativo.
Esta es la clase más importante, ya que en ella definiremos todas las operaciones básicas que vamos a utilizar en las clases que iremos definiendo más adelante para crear nuestros algoritmos y esquemas. Veamos cómo se construyó el archivo "header" PolynomialClass.h.

Lo primero que hicimos fue acotar la dimensión de mi espacio de bits a 2^{64} dando como máximo 64 entradas, es decir bits de longitud 64. Esto se refleja en el anillo de polinomios ya que estaríamos considerando a lo más polinomios de grado 63 o bien polinomios de grado 64 sin término constante. Si enumeramos estos polinomios como se mostró en la tabla de la sección anterior, estaríamos considerando un conjunto de 2^{64} enteros, cada uno de ellos representa a un y y sólo un polinomio. Justo por esto podemos considerar el tipo de dato polinomio como un "long" (typedef long polinomio).

A continuación definí los atributos de la clase, es decir; las propiedades que quiero que tengan los objetos que voy a instanciar de esta clase. Aquí sólo tengo que pensar como matemático, cada polinomio tiene: una representación entera (long), una representación abstracta (string), un grado (int) y una longitud máxima (previamente definida como typedef).
Para terminar tenemos los métodos de la clase, es decir; las operaciones que son validas en un anillo de polinomios en un sentido algebraico:

```
// PolynomialClass.h
// Created by Leopoldo G Vargas on 12-04-23.
// Copyright (c) 2012 LK Dev. All rights reserved.
+import ¡Foundation/Foundation.h¿
+define NUMERODEBITS 64 (SE DEFINE LA BIT LONGITUD O GRADO MAXIMO DE POL)
typedef long polinomio;(UN ENTERO TIPO LONG PARA CADA POLINOMIO)
@interface PolynomialClass : NSObject[
ATRIBUTOS DE LOS POLINOMIOS long polinomEntero; (CADA POL SERA REPRESENTADO POR UN LONG)
NSString *polyString; (CADA POL TENDRA SU REPRESENTACION ALGEBRAICA)
int degree; (GRADO DE POL)
]
OPERACIONES POLINOMIALES - METODOS -(void) displayPoly;
-(int) setDegree : (polinomio)p;
-(NSString *) generatePoly: (polinomio)p;
-(polinomio) suma: (polinomio)p: (polinomio)q;
-(polinomio) multiplicacion: (polinomio)p: (polinomio)q;
-(polinomio) residuo: (polinomio)p: (polinomio)q;
-(void) evaluar: (polinomio)p: (int)x;
@end
```

[**?**]

-(polinomio) suma: (polinomio)p: (polinomio)q;
polinomio pMASq = p(XOR)q; return (pMASq);
Este método representa nuestra suma polinomial en el anillo haciendo uso del operador binario XOR. Dada la naturaleza de los coeficientes de los polinomios (0,1) es exactamente equivalente hacer la suma de campo en dos polinomios que aplicar el operador XOR a los bits que los representan bajo el morfismo de proyección P (sección anterior).

-(polinomio) multiplicacion: (polinomio)p: (polinomio)q;
int i; int m = [self setDegree:q]; (grado q)
polinomio prod=0L;
for(i=0;i¡=m+1;i++)[
if(q (AND) (1L<<(i))) prod= prod (XOR) (p)<<i;]
return (prod);
Este método de clase representa la multiplicación de campo haciendo uso de un bucle "for" y una compoción de los operadores XOR y AND. Esta operación algorítmica binaria es exactamente equivalente a la operación producto en el anillo que estamos considerando. La demostración de este hecho será através de los resultados prácticos que obtendremos más adelante en el capítulo.

-(polinomio) residuo: (polinomio)p: (polinomio)q;
int p0 = [self setDegree:p]; (GRADO DE P)
int q0 = [self setDegree:q]; (GRADO DE Q)
if(p0¡q0)[[self generatePoly:p]; return (p);]
else [polinomio PentreQ = p(XOR)(q¡¡(p0-q0));
return ([self residuo:PentreQ:q]);]
Como su nombre lo indica este método calcula el residuo de un polinomio p módulo otro polinomio q, hace uso de condicionales y operadores lógicos para llevar a cabo su tarea.

-(void) displayPoly: sirve para mostrar la representación abstracta de un bit-entero- polinomio en la consola del compilador.

-(int) setDegree : (polinomio)p: revisa cuál es el bit prendido más lejano al bit más significativo (busca el bit prendido que este más hacia la izquierda) y usa esta información para regresarnos el grado del polinomio.

-(NSString *) generatePoly: (polinomio)p: crea una representación abstracta polinomial de un bit. Este método recibirá un entero (long) y lo convertirá en un bit para después darle una representación comprensible para un matemático.

También esta el método:
-(void) displayPoly: el cual sirve para extraer la representación polinomial de un bit.

6.4 La clase del campo de Galois $\mathbb{F}_{2^n}$

Ahora vamos a definir la primer sub clase de PolynomialClass.h, con la cual vamos a representar el campo de Galois $\mathbb{F}_{2^n}$. Recordemos que el campo $\mathbb{F}_{2^n}$ resulta de obtener las clases de equivalencia en el anillo $\mathbb{Z}_2[x]$ haciendo uso del ideal K generado por un elemento irreducible $i \in \mathbb{Z}_2[x]$. Por lo tanto los elementos irreducibles jugarán un papel determinante en la construcción de los algoritmos que representan las operaciónes de campo. Veamos cómo definí el archivo de cabecera FiniteFIeldsBase2.h:

Fijémonos en cómo están definidos los atributos de esta clase. Es claro que no describen a un polinomio en específico como la clase anterior. En este caso, una instancia (objeto) de esta clase será un campo finito abstracto, no un elemento en específico. Esta clase fue definida así ya que en algunos casos es necesario considerar el campo completo para llevar a cabo la elección al azar de elementos en él. Las operaciónes de campo serán controladas por los elementos irreducibles que definen al objeto campo que estamos considerando, en específico: los elementos irreducibles de grado n domde n es la misma que el exponente que determina el orden de $\mathbb{F}_{2^n}$ del campo que estamos instanciando en el tiempo de ejecución. Veamos a detalle los atributos de esta clase:

- int base, int exponent: almacenan los enteros que representan el orden del campo tales que $|F_{2^n}| = base^{exponent}$.

- NSArray *irreducPolyArray, *irredPolIntValArray: estos arreglos son lo encargados de guardar las representaciones abstractas y enteras respectivamente, de los polinomios irreducibles del campo que estamos instanciando.

```
// FiniteFIeldsBase2.h
// Created by Leopoldo G Vargas on 12-04-26.
// Copyright (c) 2012 LK Dev. All rights reserved.
+import ¡Foundation/Foundation.h¿
+import "PolynomialClass.h" (Incluimos la clase previa para instanciar polinomios)
@interface FiniteFIeldsBase2 : NSObject[ (Atributos)
int base; (GENERO DEL CAMPO, SIEMPRE SERA 2)
int exponent;(EXP AL CUAL SE ELEVARA LA BASE PARA OBTENER EL ORDEN DEL CAMPO)
NSArray *irreducPolyArray;(CONTIENE LOS POLINOMIOS IRREDUCIBLES DEP DEL ORD)
NSArray *irredPolIntValArray;(CONT EL VALOR ENTERO DE LOS POLS IRRED)
long order;(ORDEN DEL CAMPO QUE ESTAMOS CONSIDERANDO)
NSArray * elements;(ELEMENTOS QUE CONTIENE EL CAMPO) ]
(METODOS)
-(NSArray *) generateField: (int)exp;
-(void) displayField;
-(polinomio)sumaGFbase2:(polinomio)p:(polinomio)q;
-(polinomio)productoGFbase2:(polinomio)p:(polinomio)q;
-(polinomio)elevarGFbase2:(polinomio)p:(int)exp;
-(NSArray *)irreductiblePolynomials: (int)exp;
-(polinomio) compressionEndomorfism:(polinomio)p :(int)newDimension;
@end
```

[**?**]

La elección del polinomio irreducible depende enteramente de la elección del atributo de clase; int exponent; definido arriba.

- long order; order = $base^{exponent}$

-NSArray * elements: sostiene la representación abstracta de los elementos del campo. Con el objetivo de mostrarlos al usuario como mejor convenga.

Los métodos definidos en esta clase se encargan de la interacción de los elementos de la clase PolynomialClass.h, controlados por el elemento irreducible que corresponde a la elección del exponente que determina el orden del campo. Para llamar métodos de la clase declararemos un objeto PolynomialClass *gen = [[PolynomialClass alloc]init]; para a través de él poder accesar los atributos y métodos que deseemos. Veamos cómo funcionan los más importantes:

-(NSArray *) generateField: (int)exp; base =2, exponent = exp, int ord = pow(base, exp), order = ord;
NSMutableArray *tempArray = [[NSMutableArray alloc]init];
for (int i = 0; i¡order; i++) [
PolynomialClass *gen = [[PolynomialClass alloc]init];
NSString *temp = nil;
temp = [gen generatePoly:i];
(tempArray insertObject:temp atIndex:i);
(gen release);]
return (tempArray);
(tempArray release);

Como podemos observar, este método únicamente recibe un *intexp*; con el objetivo de calcular el orden del campo $base^{exp} \rightarrow 2^{exp}$ y de esta forma saber cuántos (orden) enteros tiene que traducir a su representación polinomial haciendo uso del método [gen generatePoly:i] que definimos en la clase pasada.

-(void) displayField: muestra la representación abstracta de los elementos del campo en la consola del compilador.

-(polinomio)sumaGFbase2:(polinomio)p:(polinomio)q;

```
PolynomialClass *sumaPQ = [[PolynomialClass alloc]init];
polinomio pMASq = p(XOR)q;
return (pMASq);
(sumaPQ release);
```

Este método representa la suma de elementos en el campo finito F_{2^n}. Notemos cómo el resultado de aplicar el operador binario (XOR) está siendo "modulado" por un polinomio irreducible cuyo valor está almacenado en el arreglo previamente definido en los atributos de la clase. El método [sumaPQ residuo:pMASq :irreducPoly]; está siendo llamado de la clase anterior para modular los cálculos y asegurarnos de que el resultado de las operaciones caiga en el campo asociado al polinomio irreducible.

```
-(polinomio)productoGFbase2:(polinomio)p:(polinomio)q;
PolynomialClass *prodPQ = [[PolynomialClass alloc]init];
polinomio pPORq = [prodPQ multiplicacion:p :q];
NSString *irredString = [irredPolIntValArray objectAtIndex:0];
polinomio irreducPoly = [irredString intValue];
polinomio pPORqMOD = [prodPQ residuo:pPORq :irreducPoly];
(prodPQ release);
return (pPORqMOD);
```

En este caso la idea es la misma que en el método anterior pero queremos representar el producto de campo. Para llevar esto a cabo llamamos a los métodos [prodPQ multiplicacion:p :q]; y [prodPQ residuo:pPORq :irreducPoly]; que definimos en la clase anterior, para realizar el producto de polinomios como estaba definido previamente, y después calcular el residuo módulo el polinomio irreducible que le corresponde al campo que estamos considerando.

```
-(polinomio)elevarGFbase2:(polinomio)p:(int)exp;
PolynomialClass *prodMod = [[PolynomialClass alloc]init];
if (exp == 0) return (1);
if (exp == 1) return (p);
else [ polinomio p1 = p;
for (int i = 2; i¡exp+1; i++) [
p1 = [prodMod multiplicacion:p :p1]; ]
(prodMod release);
```

return (p1);]

El objetivo de este método es el de elevar un polinomio por numero entero, en otras palabras: multiplicar un polinomio por él mismo tantas veces como un entero. Para llevar esto a cabo haremos uso del método [prodMod multiplicacion:p :p1]; que definimos en la clase anterior y de un bucle for para repetir la multiplicación tantas veces como queramos.

-(NSArray *)irreductiblePolynomials: (int)exp;
PolynomialClass *GLfields = [[PolynomialClass alloc]init];
if (exp == 1) [
irredPolIntValArray = [[NSArray alloc]initWithObjects:@"1", nil];
irreducPolyArray = [[NSArray alloc]initWithObjects:[GLfields generatePoly:1], nil];
return (irredPolIntValArray);]
if (exp == 2) [
irreducPolyArray = [[NSArray alloc]initWithObjects:[GLfields generatePoly:7], nil];
irredPolIntValArray = [[NSArray alloc]initWithObjects:@"7", nil];
return (irredPolIntValArray);]
if (exp == 3) [
irreducPolyArray = [[NSArray alloc]initWithObjects:[GLfields generatePoly:11], [GLfields generatePoly:14], nil];
irredPolIntValArray = [[NSArray alloc]initWithObjects:@"11",@"13", nil];
return (irredPolIntValArray);]

Como podemos observar, este método es muy simple ya que recibe un entero que representa el exponente que determina el orden del campo, y en torno a ese entero decide qué elementos irreducibles le corresponden al campo en el cual queremos llevar acabo las operaciones. Nos interesan ambas representaciones, tanto la entera como la abstracta, por lo tanto este método hace uso de [GLfields generatePoly:1] definido en la clase anterior para traducir el valor entero al valor algebráico y guardar ambos en distintos arreglos. Así los otros métodos y clases pueden hacer uso de los elementos irreducibles directamente sin calcularlos o aproximarlos. Este método hasta ahora soporta un máximo de $int\ exp = 17$, bastará alimentar este método con más datos de elementos irreducibles para poder trabajar en campos de mayor orden.

```
-(polinomio) compressionEndomorfism:(polinomio)p :(int)newDimension;
PolynomialClass *poly = [[PolynomialClass alloc]init];
NSArray *irred = [self irreductiblePolynomials:newDimension];
polinomio newIrred = [[irred objectAtIndex:0]intValue];
polinomio compre = [poly residuo:p :newIrred];
return (compre);
```

Este método será de crucial importancia para esta tesis, su análogo abstracto es el epimorfismo ϕ utilizado en la construcción de familias hash fuertemente universales (Sección 5.2.5). Recibe como parámetros de entrada la representación polinomial e un entero $(polinomio)p$ y otro entero $(int)newDimension$, en donde el primero representa el elemento del campo $\mathbb{F}_{2^n}$ que vamos a enviar con el epimorfismo y el segundo representa la m que determina al campo $\mathbb{F}_{2^m}$, donde $m < n$. Nos sirve para enviar un elemento de un campo en específico a otro elemento en otro campo de orden menor. Para realizar esta tarea satisfactoriamente nuestro método extrae un elemento irreducible del campo que representa el rango del epimorfismo, en este caso determinado por m, y hace uso de ese elemento y de la función residuo definida en la clase PolynomialClass para calcular $(polinomio)pmodirred$ obteniendo así el elemento correspondiente en el nuevo campo de dimensión m.
Juega un papel muy importante a la hora de ejecutar el algoritmo de firmado ya que estará encargado de comprimir el resultado de las evaluaciones en ambos pasos del algoritmo de generación de firmas.

6.5 La clase del anillo de polinomios $\mathbb{F}_{2^n}[Y]$

Ahora queremos utilizar el campo $\mathbb{F}_{2^n}$ como el campo de coeficientes de un nuevo anillo de polinomios $\mathbb{F}_{2^n}[Y]$ de dimensión k. Este nuevo anillo será de la forma: $\mathbb{F}_{2^n}[Y] = [\sum_{i=1}^{k} p_i Y^i | p_i \in \mathbb{F}_{2^n}, i \in (1,k)]$.

Los elementos del anillo $F_{2^n}[Y]$ son polinomios en la variable Y, sin término constante y con coeficientes en el campo finito $\mathbb{F}_{2^n}$. Dada su construcción podemos llamarlos operadores polinomiales, ya que en un breve futuro serán utilizados para evaluar elementos del campo $\mathbb{F}_{2^n}$ y así generar nuestras firmas digitales.
El orden del anillo $\mathbb{F}_{2^n}[Y]$ es $(2^n)^{k-1}$ porque si lo pensamos desde el punto de vista de combinatoria estamos considerando las ordenaciones con repetición

de un alfabeto $\mathbb{F}_{2^n}$ que consta de 2^n elementos en $k-1$ casillas. El orden de este anillo se dispara exponencialmente aun para elecciones "pequeñas" de n y k, y esto sera de vital importancia para la generación de nuestras firmas digitales ya que el algorítmo de firmado dependerá principalmente de la elección al azar de uno de estos operadores polinomiales para firmar el mensaje en cuestión. Entre más grande sea el espacio donde vamos a elegir nuestro operador polinomial nuestro algoritmo de firmado será más efectivo, ya que las probabilidades de encontrar colisiones se verán más y más reducidas, como lo especificamos en el capítulo anterior.

Los objetos que instanciaremos de esta clase serán anillos de la forma $\mathbb{F}_{2^n}[Y]$ en vez de elementos del mismo. Esto con el objetivo de poder contemplar y manipular el espacio completo. Como vimos hace un momento, el orden de estos anillos se dispara exponencialmente para elecciones pequeñas de n y k; por lo tanto instanciaremos parcialmente los espacios y escogeremos elementos específicos de ellos haciendo uso de un protocolo externo de generación de llaves Diffie - Hellman que describiremos más a detalle en la sección siguiente. Veamos cómo definimos la clase PolynomialRingOverGL(2n).h/.m, comencemos con el archivo de cabecera (.h):

Los atributos de esta clase describen las tres principales características del anillo que estamos considerando, es decir: NSArray *ringElements; int dimension, order. El primero es un arreglo que sostiene los valores enteros de los elementos del anillo, para elecciones suficientemente grandes de k y n este arreglo quedará vacío debido al disparado orden del espacio. Después vienen dos enteros que sostienen el valor de k = dimensión y orden = $order$. Estos tres atributos nos son suficientes para describir el anillo $\mathbb{F}_{2^n}[Y]$. Veamos ahora un poco más a detalle la forma en que interactúan estos espacios entre sí y con elementos del campo $\mathbb{F}_{2^n}$. Veamos también como definí los métodos en cuestión.

```
-(NSArray *) generateRing:(int)k;
NSSet *combinations = [[NSSet alloc]initWithObjects:
@"0",@"1",@"2",@"3",@"4",@"5",@"6",@"7", nil];
NSSet *combinationSet = [combinations variationsWithRepetitionsOfSize:k];
NSArray *var = [combinationSet allObjects];
ringElements = var;
dimension = k;
```

```
// PolynomialRingOverGL(2n).h
// Created by Leopoldo G Vargas on 12-05-09. // Copyright (c) 2012 LK Dev. All rights reserved. IMPORTAMOS LAS CLASES DESDE LAS CUALES LLAMAREMOS OPER Y METODOS +import ¡Foundation/Foundation.h¿
+import "FiniteFIeldsBase2.h"
+import "NSSet+Combinatorics.h"
+import "Key-Expanssion.h"
@interface PolynomialRingOverGL2n : NSObject[ ATRIBUTOS DE ANILLO
NSArray *ringElements;
int dimension, order;
]
METODOS
-(NSArray *) generateRing:(int)k;
-(NSString *) displayElment: (int)e;
-(polinomio) evluatePolyinOperator:(polinomio)elem:(long)oper: (int)exp;
-(polinomio) ReedSolomonEval:(polinomio)elem:(long)oper:(int)exp;
@end
```

[?]

order = [var count];
return (var);
Este método se encarga de generar el anillo calculando las ordenaciones con repetición del valor entero de los elementos del campo F_{2^n} en k casillas. Para hacer estos cálculos el método se apoya en la clase NSSet+Combinatorics.h (en cuya explicación no me detendré en esta tesis) la cual es una clase que contiene las principales funciónes combinatorias.

-(NSString *) displayElment: (int)e;
Método simple para mostrarle al usuario los elementos del anillo en la consola del compilador o en alguna otra interfaz de usuario.

6.5.1 Primera parte del esquema de Firmas Digitales.

Este método constituye la primera parte de la firma digital generando el código Reed Solomon $RS(2^n, k)$ Este método recibe tres parámetos distintos; :(polinomio)elem:(long)oper: (int)exp. Los cuales representan: el polinomio que va a ser evaluado, el elemento de $RS(2^n, k)$ en donde se evaluará el polinomio y la dimensión de $RS(2^n, k)$, respectivamente. El último parámetro es de vital importancia, ya que determina cuál será el código Reed=Solomon que utilizaremos.
Este método nos regresa el resultado de evaluar el polinomio obtenido de la primera parte de la firma $p(x) \in F_{2^m}$ en un elemento de $RS(2^m, k)$. Con el objetivo de entender paso a paso cómo diseñe este método, haremos un análisis paso por paso del mismo. La correcta implementación de este método es necesaria para el correcto funcionamiento del esquema y para el éxito de este proyecto.

-(polinomio) ReedSolomonEval:(polinomio)elem:(long)oper:(int)exp;

1. NSMutableArray *monomialArray = [[NSMutableArray alloc]init];
   FiniteFIeldsBase2 *evaluarBase2 = [[FiniteFIeldsBase2 alloc]init];
   PolynomialClass *polyArithmetics = [[PolynomialClass alloc]init];
   KeyExpanssion *keyExpand = [[KeyExpanssion alloc]init];
   NSArray *irredArray= [evaluarBase2 irreductiblePolynomials:exp];
   polinomio irred = [[irredArray objectAtIndex:0] intValue];

Primero que nada nuestro método crea instancias de las clases que utilizará y define dos arreglos: uno mutable y vacío, y el otro poblado con los elementos irreducibles que corresponden al campo que esta determinado por el (int)exponent que fue especificado. Se extrae también un elemento irreducible del arreglo recién creado para modular los cálculos.

2. if (exp ¡17) monomialArray = [keyExpand ExpandKey:oper :6];
   if (exp ¡14) monomialArray = [keyExpand ExpandKey:oper :5];
   if (exp ¡10) monomialArray = [keyExpand ExpandKey:oper :4];
   if (exp ¡7) monomialArray = [keyExpand ExpandKey:oper :3];

   Después se llevan a cabo una serie de condicionales if sobre el $(int)exponent$ para determinar cuál es la mejor forma de segmentar la representación entera del operador polinomial. Para llevar a cabo esta segmentación y expansión este método se vale de la clase KeyExpanssion (Sección 6.6). Al término de este paso la representación entera del operador polinomial se encontrará distribuida y adaptada en un NSMutableArray.

3. for (int i = 0; i¡[monomialArray count]; i++) [
   polinomio coef = [[monomialArray objectAtIndex:i]intValue];
   polinomio coefMOD = [polyArithmetics residuo:coef :irred];
   (monomialArray replaceObjectAtIndex:i withObject:coefMODstring);]

   Ahora hace uso de un bucle for y del elemento irreducible que fue extraído en la primera parte para modular cada entrada del arreglo en donde se encuentra guardada la llave. De esta forma nos aseguramos que cada elemento que guardamos en el arreglo esté efectivamente en el código especificado por la elección de $int(exp)$

4. for (int j = 0; j¡[monomialArray count]; j++) [
   polinomio monomioElev = [evaluarBase2 elevarGFbase2:elem :j+1];
   polinomio coeff = [[monomialArray objectAtIndex:j]intValue];
   polinomio MonomioI = [polyArithmetics multiplicacion:monomioElev
   :coeff];

polinomio MonomioIMod = [polyArithmetics residuo:MonomioI :2053];
NSString *monomial = [NSString stringWithFormat:@"ld", MonomioIMod];
(monomialArray replaceObjectAtIndex:j withObject:monomial);]

En este paso se lleva a cabo la evaluación, el método hará uso de un bucle *for* para evaluar el (polinomio)elem en el operador polinomial refinado por la clase KeyExpanssion. Para esta tarea hace uso de la variación natural de j desde 0 hasta el r, donde r es el número de elementos que tiene el arreglo en el cual está distribuido el operador, para representar la variación del exponente de cada monomio del operador y así poder evaluar *elem* en cada una de las entradas del arreglo con el exponente que le corresponde como si cada una fuera un sumando del operador. Los resultados son guardados en el mismo arreglo.

5. polinomio partial = 0;
for (int k = 0; k¡ [monomialArray count]; k++) [
polinomio monomioMod = [[monomialArray objectAtIndex:k]intValue];
partial = partial(XOR)monomioMod;]
return (partial);

Finalmente el método hace uso de otro *for* para sumar los monomios que resultaron de la evaluación por partes en el paso anterior, obteniendo así nuestra evaluación completa.

6.5.2 Segunda parte del esquema de Firmas Digitales.

El método que describiremos a continuación representa la construcción de la familia de funciones hash fuertemente universales que estudiamos en la Sección 5.2.5. Como vimos en el capítulo anterior, éste es el segundo paso del algoritmo de generación de firmas digitales, ilustrado por la tabla 5.1. Con el objetivo de entender paso a paso cómo diseñamos este método, haremos un análisis paso por paso del mismo. La correcta implementación de este método es necesaria para el correcto funcionamiento del esquema y para el éxito de este proyecto.

Este método es uno de los más importantes en esta tesis, ya que será el encargado de llevar a cabo la evaluación de un polinomio en $\mathbb{F}_{2^n}$ en un operador polinomial de $\mathbb{F}_{2^n}[Y]_1(operadorespolinomialeslinealesdegradouno)$. Esta evaluación sera una representación del patrón de evaluación abstracto que definimos en el capítulo 5. Este método recibe tres parámetos distintos; :(polinomio)elem:(long)oper: (int)exp. Los cuales representan: el polinomio que será evaluado, el operador en el que se evaluará el polinomio, el exponente del $\mathbb{F}_{2^{exp}}$, respectivamente. El último parámetro es de vital importancia, ya que nos dice sobre que campo $\mathbb{F}_{2^n}$ estamos considerando nuestro anillo de operadores polinomiales. Al final este método nos regresará un polinomio en el mismo campo del que partimos que será resultado de la evalución:
EV:$\mathbb{F}_{2^n}[Y] \to \mathbb{F}_{2^n}$, $e \in \mathbb{F}_{2^n}[Y]$ y $x \in \mathbb{F}_{2^n}$; entonces: $e(x) \in \mathbb{F}_{2^n}$.

-(polinomio) evluatePolyinOperator:(polinomio)elem:(long)oper: (int)exp;

1. NSMutableArray *monomialArray = [[NSMutableArray alloc]init];
   FiniteFIeldsBase2 *evaluarBase2 = [[FiniteFIeldsBase2 alloc]init];
   PolynomialClass *polyArithmetics = [[PolynomialClass alloc]init];
   KeyExpanssion *keyExpand = [[KeyExpanssion alloc]init];
   NSArray *irredArray= [evaluarBase2 irreductiblePolynomials:exp];
   polinomio irred = [[irredArray objectAtIndex:0] intValue];

   Primero que nada nuestro método crea instancias de las clases que va a utilizar y define dos arreglos: uno mutable y vacío, y el otro poblado con los elementos irreducibles que corresponden al campo que está determinado por el (int)exponent que fue especificado. Se extrae también un elemento irreducible del arreglo recién creado para modular los cálculos.

2. if ($exp < 17$) monomialArray = [keyExpand ExpandKey:oper :6];
   if ($exp < 14$) monomialArray = [keyExpand ExpandKey:oper :5];
   if ($exp < 10$) monomialArray = [keyExpand ExpandKey:oper :4];
   if ($exp < 7$) monomialArray = [keyExpand ExpandKey:oper :3];

   Después se lleva a cabo una serie de condicionales if sobre el $(int)exponent$ para determinar cuál es la mejor forma de segmentar la representación entera del operador polinomial. Para llevar a cabo esta segmentación y

expansión este método se vale de la clase KeyExpanssion (Sección 6.6). Al término de este paso la representación entera del operador polinomial se encontrará distribuida y adaptada en un NSMutableArray.

3. for (int i = 0;$i <$ [monomialArray count]; i++) [
   polinomio coef = [[monomialArray objectAtIndex:i]intValue];
   polinomio coefMOD = [polyArithmetics residuo:coef :irred];
   NSString *coefMODstring = [NSString stringWithFormat:@"ld", coefMOD];
   (monomialArray replaceObjectAtIndex:i withObject:coefMODstring);]

   Ahora hace uso de un bucle for y del elemento irreducible que fue extraído en la primera parte para modular cada entrada del arreglo en donde se encuentra guardada la llave. De esta forma nos aseguramos de que cada elemento que guardamos en el arreglo esté efectivamente en el campo especificado por la elección de $int(exp)$.

4. for (int j = 0;$j <$[monomialArray count]; j++) [
   polinomio monomioElev = [evaluarBase2 elevarGFbase2:elem :j+1];
   polinomio coeff = [[monomialArray objectAtIndex:j]intValue];
   polinomio MonomioI = [polyArithmetics multiplicacion:monomioElev :coeff];
   polinomio MonomioIMod = [polyArithmetics residuo:MonomioI :irred];
   NSString *monomial = [NSString stringWithFormat:@"ld", MonomioIMod];
   (monomialArray replaceObjectAtIndex:j withObject:monomial);]

   En este paso se lleva a cabo la evaluación, el método hará uso de un bucle for para evaluar el (polinomio)elem en el operador polinomial refinado por la clase KeyExpanssion. Para esta tarea hace uso de la variación natural de j desde 0 hasta r, donde r es el número de elementos que tiene el arreglo en el cual está distribuido el operador, para representar la variación del exponente de cada monomio del operador y así poder evaluar $elem$ en cada una de las entradas del arreglo con el exponente que le corresponde, como si cada una fuera un sumando del operador. Los resultados son guardados en el mismo arreglo.

5. polinomio partial = 0;
   for (int k = 0; $k <$ [monomialArray count]; k++) [
   polinomio monomioMod = [[monomialArray objectAtIndex:k]intValue];
   partial = partial(xor)monomioMod;]
   return (partial);

   Finalmente el método hace uso de otro for para sumar los monomios que resultaron de la evaluación por partes en el paso anterior. Obteniendo así nuestra evaluación completa.

Para concluir de la segunda parte de la generación de la firma debemos asegurarnos de que el resultado de la evaluación sea ahora enviado a un sub campo $\mathbb{F}_{2^m}$, $m < n$ haciendo uso del epimorfismo de compresión que definimos en la clase FiniteFIeldsBase2.h. Así concluimos el algoritmo de firmado.

6.6 Esquema de Firmas basado en Universal Hashing

En esta sección veremos cómo creamos la clase principal de este proyecto en la cual están definidos los algorítmos de Firma y de Verificación de nuestro esquema. Como podremos observar, esta clase únicamente conectará y hará interactuar entre sí a objetos y métodos previamente definidos en otras clases para obtener las Firmas Digitales y su respectiva Verificación. Esta clase refleja el poder de la programación orientada a objetos ya que refleja orden y limpieza, es una clase cuyo funcionamiento depende de varias otras, y esto nos da la posibilidad de modificarla parte por parte (clase por clase) de una forma ordenada y limpia. Veamos ahora cómo está definido el archivo de cabecera para esta clase.

Como podemos observar, esta clase consiste únicamente de dos métodos: SigningAlgorithm y VerificationAlgorithm, los cuales estarán encargados de generar las firmas digitales y de verificar su validez respectivamente. El Método SigningAlgorithm recibe como parámetros de entrada la representación polinomial de un entero $(polinomio)key$ y una cadena de caracteres $(NSString*)message$ los cuales en el sentido matemático son de la forma; $key \in \mathbb{F}_{2^n}$ y $message \in \mathbb{F}_{2^n}[Y]$. Para esto tendremos que asignarle a $message$ dicha representación

```
// FirmasDigitalesUH.h
// Polynomials
// Created by Leopoldo G Vargas on 2012-11-11.

+import < Foundation/Foundation.h >
+import "PolynomialRingOverGL(2^n).h"
+import "MessageToPolynomial.h"
+import "FiniteFIeldsBase2.h"
+import "PolynomialClass.h"

@interface FirmasDigitalesUH : NSObject

-(polinomio)SigningAlgorithm: (polinomio)key: (NSString *)message;
-(BOOL)VerificatonAlgorithm: (polinomio)key: (NSString *)message;

@end
```

[?]

utilizando la clase auxiliar MessageToPolinomial (Seccion final de este capitulo). Una vez establecida la representación el método SigningAlgorithm llevará a cabo la generación de la firma exactamente como fue especificado en la Sección 5.2.5 y en las Secciones previas de este capítulo. Veamos a detalle cómo están constituidos los métodos.

-(polinomio)SigningAlgorithm: (polinomio)key: (NSString *)message[PolynomialRingOverGL2n *polyRing = [[PolynomialRingOverGL2n alloc]init]; FiniteFIeldsBase2 *Orden2n = [[FiniteFIeldsBase2 alloc]init]; PolynomialClass *arithmetics = [[PolynomialClass alloc]init];

polinomio eval = [polyRing ReedSolomonEval:key :message:32];

polinomio secondEval = [polyRing evluatePolyinOperator:eval :message:32];

polinomio compressed = [Orden2n compressionEndomorfism:secondEval :16];

polinomio verif = [arithmetics residuo:key :299];

polinomio final = $compressed^{verif}$;

return (final);
]

Este método hará uso de las clases PolynomialClass, PolynomialRingOverGL2n y FiniteFIeldsBase2 para llevar a cabo los cálculos en los Campos de la forma $\mathbb{F}_{2^n}$ que sean requeridos; y también hará uso de la clase auxiliar Message-ToPolynomial para darle un representación entera a un mensaje o cadena de caracteres a fin de interpretar ese entero como un operador polinomial en $\mathbb{F}_{2^{32}}[Y]$.

Para la primera parte de la firma nuestro método genera el código Reed-Solomon $RS(2^{32}, 1)$ y evalúa la llave $key \in \mathbb{F}_{2^{32}}$ en la representación algebraica de $message \in RS(2^{32}, 1)$ resultando en un nuevo elemento $\tilde{s} \in RS(2^{32}, 1)$.

Para generar la segunda parte de la firma el método evalúa el elemento $\tilde{s} \in \mathbb{F}_{2^{32}}$ obtenido del paso anterior en la representación algebraica de $message \in \mathbb{F}_{2^{32}}[Y]$ para así obtener un elemento $s \in \mathbb{F}_{2^{32}}$. Este elemento s es ahora enviado a un campo de menor dimensión $\mathbb{F}_{2^m}$ donde $m < n$ mediante un epimorfismo, entonces $\phi(s) = \hat{s} \in \mathbb{F}_{2^16}$. Es muy importante notar que la construcción de este algoritmo está enteramente basada en la construcción de familias de funciones hash que expusimos en la Sección 5.2.5, y el proceso de evaluación y compresión está claramente ilustrado en la Tabla 5.1.

-(BOOL)VerificatonAlgorithm: (polinomio)key: (NSString *)messageWsig
El algoritmo de verificación especificado por este método está basado en su enteridad en el algoritmo de generación de firmas. Veamos a qué se debe.

Este método recibe como parámetros de entrada un entero (polinomio)key y una cadena de caracteres (NSString *)messageWsig el cual contiene al mensaje que fue firmado y la firma digital del mismo como un apéndice que le fue concatenado al mensaje. El algoritmo de firmado simplemente extraerá el mensaje en su estado origen y le aplicará el algoritmo de firmado haciendo uno de la llave privada correspondiente. Una vez obtenida la firma, el algoritmo de Verificación compara la firma que fue enviada por el firmador

y la firma que se obtuvo del cálculo propio de la misma.
Si estas dos firmas son iguales el método regresará el valor booleano 1, dando la firma por aceptada: de lo contrario regresará 0, y dará la firma por invalida.

6.7 Clases Auxiliares

En esta sección estudiaremos dos clases que apoyan nuestro esquema de firmas digitales en el sentido de que lo ayudan a encajar en contexto para su óptimo funcionamiento. La clase de generación de llaves Diffie-Hellman nos servirá para generar las llaves (representación polinomial) que serán evaluadas en los operadores que viven en $\mathbb{Z}_{2^n}[Y]$ (Capitulo 5) y la clase de expansión de llaves me servirá para modificar y adaptar la longitud de cadenas de números enteros con el proposito de utilizarlas con eficiencia y presición. Veamos a detalle cómo fueron construidos e implementados los algoritmos que determinan las mencionadas clases.

6.7.1 Expansión de Llaves

En esta sección describiremos la clase auxiliar Key-Expanssion. El objetivo de esta clase es el de incrementar la longitud de una cadena de enteros mediante la evaluación consecutiva en un polinomio en $\mathbb{F}_{2^n}$ (permutaci ón pseudo aleatoria), dicha longitud podrá variar para adaptarse a nuestras necesidades. Veamos cómo está definido el archivo de cabecera Key-Expanssion.h.

Es evidente que esta clase sólo consta de un método, el cual recibe como parámetros de entrada un entero $(long)seed$ y otro entero $(int)length$. En este caso *seed* representa la cadena de números enteros que queremos expandir y length representa la longitud que queremos que tenga cada sección (coeficiente en el sentido polinomial) de nuestra cadena expandida. Este método nos regresa como resultado de sus cálculos y aproximaciones un (NSMutableArray *), esto debido a que en un arreglo podemos segmentar la cadena y expandirla por secciones, obteniendo así una cadena o llave tan larga como deseemos. Veamos ahora cómo funciona este método.

-(NSMutableArray *) ExpandKey:(long)seed: (int)length

NSString *seedST = [NSString stringWithFormat:@"ld", seed];
NSMutableArray *seedArray = [[NSMutableArray alloc]init];

```
// Key-Expanssion.h
// Polynomial
// Created by Leopoldo G Vargas on 12-05-31.

+import < Foundation/Foundation.h >
+import "FiniteFIeldsBase2.h"
+import "PolynomialClass.h"

@interface KeyExpanssion : NSObject

-(NSMutableArray *) ExpandKey:(long)seed: (int)length;

@end
```

[?]

```
FiniteFIeldsBase2 *fieldArithm = [[FiniteFIeldsBase2 alloc]init];
PolynomialClass *poliArithm = [[PolynomialClass alloc]init];
if ([seedST length] AND 1)

    for (int i = 0; i ¡ [seedST length]; i=i+1)
char seed0 = [seedST characterAtIndex:i];
NSString *charSeed = [NSString stringWithFormat:@"c",seed0 ];
[seedArray addObject:charSeed];
else

    for (int i = 0; i ¡ [seedST length]; i=i+2)
char seed0 = [seedST characterAtIndex:i];
char seed1 = [seedST characterAtIndex:i+1];
NSString *charSeed = [NSString stringWithFormat:@"cc",seed0, seed1 ];
[seedArray addObject:charSeed];

    if (length == 2) return (seedArray);
    if (length == 3)[
    for (int j = 0; j¡[seedArray count]; j++)
polinomio var1 = [[seedArray objectAtIndex:j]intValue];
polinomio var3 =[fieldArithm elevarGFbase2:var1 :3];
polinomiovarFinal = var3 + var1 + 1;
```

```
polinomio varFinalMOD = [poliArithm residuo:varFinal :13];
NSString *final = [NSString stringWithFormat:@"ldld", var1, varFinalMOD];

    [seedArray replaceObjectAtIndex:j withObject:final]; ]

    if (length == 4) [

    for (int j = 0; j¡[seedArray count]; j++)
polinomio var1 = [[seedArray objectAtIndex:j]intValue];
polinomio var3 =[fieldArithm elevarGFbase2:var1 :3];
polinomio varFinal = var3 + var1 + 1;
polinomio varFinalMOD = [poliArithm residuo:varFinal :13];
NSString *final = [NSString stringWithFormat:@"ldld", var1, varFinalMOD];

    polinomio var12 = [final intValue];
polinomio var32 =[fieldArithm elevarGFbase2:var12 :3];
polinomio varFinal2 = var32 + var12 + 1;
polinomio varFinalMOD2 = [poliArithm residuo:varFinal2 :13]; NSString *fi-
nal2 = [NSString stringWithFormat:@"@ld", final, varFinalMOD2];
    [seedArray replaceObjectAtIndex:j withObject:final2]; ]
```

Lo primero que hace este método es revisar si la cadena de enteros $(long)seed$ tiene longitud par o impar con el objetivo de escoger el método apropiado de seccionamiento. Si la longitud de *seed* es impar entonces se aplica un bucle for con el objetivo de poblar cada una de las entradas de un arreglo(seedArray) con cada uno de los enteros que constituyen la cadena, empezando por colocar el primer entero que está a la izquierda en la primera entrada del arreglo. Si la longitud de seed es par entonces poblaremos cada entrada del arreglo(seedArray) con una pareja de enteros, comenzando por colocar la primera pareja de enteros que están a la izquierda en la primera entrada del arreglo y así sucesivamente.

Una vez poblado el arreglo nuestro algoritmo considerará el entero (int)length, el cual como dijimos previamente le indica qué tan larga será cada sección de nuestra llave, o, en otras palabras, qué tan larga será cada entrada del arreglo que acabamos de poblar.
Si length = 2 entonces se regresa el mismo arreglo sin modificar.
Si $3 \leq length$ entonces necesitamos incrementar la longitud de cada una de

las entradas de mi arreglo al menos una vez. Para llevar esto a cabo nuestro método hará uso del polinomio $x^3+x+1 \in \mathbb{F}_{2^13}$ para evaluar cada una de las entradas de mi arreglo y después concatenar el resultado de cada evaluación al final de la entrada que fue evaluada para obtener el resultado. Este proceso será repetido cuantas veces sea necesario para satisfacer lo especificado por el parámetro de entrada "length, obteniendo así una llave segmentada en las entradas de un NSMutableArray tan larga como queramos.

6.7.2 Protocolo de generación de llaves Diffie-Hellman (PLD)

En esta clase implementamos un protocolo Diffie-Hellman de generación de llaves basado en el Problema de Logarítmo Discreto sobre el grupo multiplicativo del campo $\mathbb{F}_{2^n}$. Veamos como definimos el archivo de cabecera DiffieHellman-KeyExchange.h:

Como podemos observar en la declaración de la clase, estamos describiendo un sistema completo de generación de llaves en donde el grupo multiplicativo en el cual estamos haciendo las cuentas queda únicamente determinado por el polinomio irreducible que escogemos para modular las multiplicaciones. Es decir, estaremos utilizando el grupo $(\mathbb{F}_{2^n}, *, e)$, el grupo multiplicativo del campo $\mathbb{F}_{2^n}$. A su vez las llaves quedan determinadas por: polinomio baseG y por la libre elección del usuario. Cada objeto que instanciemos de esta clase tendrá la capacidad de proporcionarnos llaves secretas para distintos parámetros libres a la elección del usuario. Veamos a detalle cómo funcionan los métodos que definimos aquí:

```
-(void) generateContext:(int)exp;
FiniteFIeldsBase2 *keyArithmetics = [[FiniteFIeldsBase2 alloc]init];
NSArray *irredArray = [keyArithmetics irreductiblePolynomials:exp];
polinomio irred = [[irredArray objectAtIndex:0] intValue];
irreductible = irred;
int partbase = pow(2, exp);
polinomio base = partbase - exp;
baseG = base;
```

Este método recibe un entero que representa el exponente de $\mathbb{F}_{2^{exp}}$. Este

```
// DiffieHellman-KeyExchange.h
// Created by Leopoldo G Vargas on 12-05-17.
// Copyright (c) 2012 LK Dev. All rights reserved.
IMPORTAMOS LAS CLASES QUE USAREMOS
+import < Foundation/Foundation.h >
+import "FiniteFIeldsBase2.h"
+import "PolynomialClass.h"
ATRIBUTOS @interface DiffieHellmanKeyExchange : NSObject[
polinomio privateKey; VALOR POLINOMIAL PARA LA LLAVE SECRETA
polinomio publicKey; VALOR POLINOMIAL PARA LA LLAVE PUBLICA
polinomio baseG; VALOR POL PARA LA BASE CON LA QUE GENERAREMOS LAS LLAVES
polinomio irreductible; POL IRRED QUE SERA USADO PARA MODULAR LOS CALCULOS
]
-(void) generateContext:(int)exp;
-(polinomio) createPublicKey:(polinomio)secKey;
-(polinomio) createSECRETkey:(polinomio)pubKey:(polinomio)secretKey;
@end
```

[?]

último es utilizado para elegir el polinomio irreducible del arreglo NSArray *irredPolIntValArray definido en la clase; FiniteFIeldsBase2.h. Este polinomio irreducible es utilizado para que los cálculos estén bien definidos en $\mathbb{F}_{2^n}$, determinando así el grupo en donde generaremos las llaves secretas.

```
-(polinomio) createPublicKey:(polinomio)secKey;
FiniteFIeldsBase2 *keyArithmetics = [[FiniteFIeldsBase2 alloc]init];
PolynomialClass *polyKey = [[PolynomialClass alloc]init];
polinomio partKey = [keyArithmetics elevarGFbase2:baseG :secKey];
polinomio pubKey = [polyKey residuo:partKey :irreductible];
publicKey =pubKey;
return (pubKey);
```

El objetivo de este método es crear las llaves públicas serán enviadas por un canal no seguro de transferencia de datos: recibe la representación polinomial de una llave "(polinomio)secKey" y hace uso del protocolo Diffie-Hellman de generación de llaves (Capitulo 3) basado en el PLD.

```
-(polinomio) createSECRETkey:(polinomio)pubKey:(polinomio)secretKey;
FiniteFIeldsBase2 *keyArithmetics = [[FiniteFIeldsBase2 alloc]init];
PolynomialClass *polyKey = [[PolynomialClass alloc]init];
polinomio partKey = [keyArithmetics elevarGFbase2:pubKey:secretKey];
polinomio SECKey = [polyKey residuo:partKey :irreductible];
return (SECKey);
```

Método encargado de calcular las llaves secretas de ambos lados de la transferencia. Recibe la llave pública "(polinomio)pubKey" que fue enviada por el canal, la llave privada "(polinomio)secretKey" que fue usada previamente para generar la llave pública en el método anterior.

Capítulo 7

Conclusión

En este capítulo analizaremos los resultados de la implementación del esquema de firmas que desarrollamos. Para nuestra implementación escogeremos $q = 2$, $\alpha = 32$ y $\beta = 16$ lo cual según el teorema 5.2.4 resulta en una familia de funciones $(2^{80}; 2^{2^{21}}, 2^{16})$, la cual resulta ser $2q^{-16}$ Fuertemente Universal. Notemos que un esquema que funcione con estos parámetros sería ya muy útil en la práctica ya que podemos verificar estados origen de longitud 2^{21} bits con únicamente 16 bits de verificación, y las llaves las escogeremos aleatoriamente de un espacio de llaves de 80 bits. Dicho de una forma más clara: este esquema nos permitirá firmar mensajes (archivos) de longitud $2^{21} = 2097152$ bits es decir $2^{18} = 262144$ bytes con firmas de longitud 2^4 bits, ie; 2 bytes y con probabilidad de ruptura limitada a 2^{-15}. Como dijimos antes, la familia de funciones hash criptográficas asociadas a los parámetros públicos y privados dados, será resultado de componer la familia descrita en el teorema 12.3 con la familia descrita en el corolario 12.1. Cuyos parámetros de seguridad están dados por el teorema 12.4.

Veamos ahora paso a paso el modo en que generamos las firmas digitales. Como dijimos en el capítulo anterior, y de acuerdo a la primera parte de la prueba del teorema 12.4, primero que nada generaremos el código Reed-Solomon con parámetros $RS(q^{\alpha-\beta}, q^{\alpha})$, para nuestro contexto será el $RS(2^{16}, 2^{32})$, es decir; todas las combinaciones del alfabeto de 2^{32}elementos $(|\mathbb{F}_{2^{32}}|)$ en 2^{16} casillas. Ahora evaluaremos el elemento asociado a la llave $k \in \mathbb{F}_{2^{32}}$ en el operador polinomial asociado al mensaje o archivo, $m \in \mathbb{F}_{2^{32}}[Y]$. En el capítulo anterior describimos con mayor detalle la forma en que implementamos la elección de la llave y la asociación del estado origen al operador

polinomial. El resultado de la evaluación $m(k) \in \mathbb{F}_{2^{32}}$ es un elemento del campo base. De esta forma concluimos la primera parte del proceso de asignación de verificador, o, dicho de otra forma, el proceso de firmado digital.

Para continuar con la asignación de verificador consideraremos el campo $\mathbb{F}_{2^{32}}$ y el anillo de polinomios $\mathbb{F}_{2^n}[Y]_1$, los cuales como ya hemos mencionado son un campo finito de característica dos y orden 2^{32}y el anillo de polinomios lineales con coeficientes en $\mathbb{F}_{2^{32}}$ sin término constante. El elemento que resultó de la previa evaluación será ahora evaluado en un elemento en $\mathbb{F}_{2^{32}}[Y]_1$, y este último será escogido mediante una permutación finita (reacomodo y compresión de coeficientes) del representante del archivo o mensaje que escogimos en el paso anterior. Una vez efectuada la evaluación obtendremos nuevamente un elemento de $\mathbb{F}_{2^{32}}$.

Para concluir e incrementar la seguridad de nuestra asignación de verificador tendremos ahora que enviar al elemento que resultó de los dos pasos previos a un espacio de menor cardinalidad que el espacio en el que actualmente habita el resultado. Para hacer esto haremos uso del epimorfismo de compresión ϕ que describimos en el capítulo anterior. El cual para nuestro contexto irá de $\mathbb{F}_{2^{32}}$ a $\mathbb{F}_{2^{16}}$ reduciendo la longitud en bits del elemento que resultó de las dos evaluaciones previas. La función ϕ reduce la longitud de las representaciones binarias de los elementos en $\mathbb{F}_{2^{32}}$ de 32 a 16 bits, esto con el objetivo de acortar todavía más la firma y conceder una mayor eficiencia al esquema. Así estaremos firmando mensajes o archivos de longitud 2^{21} con firmas de únicamente 2^4 bits.

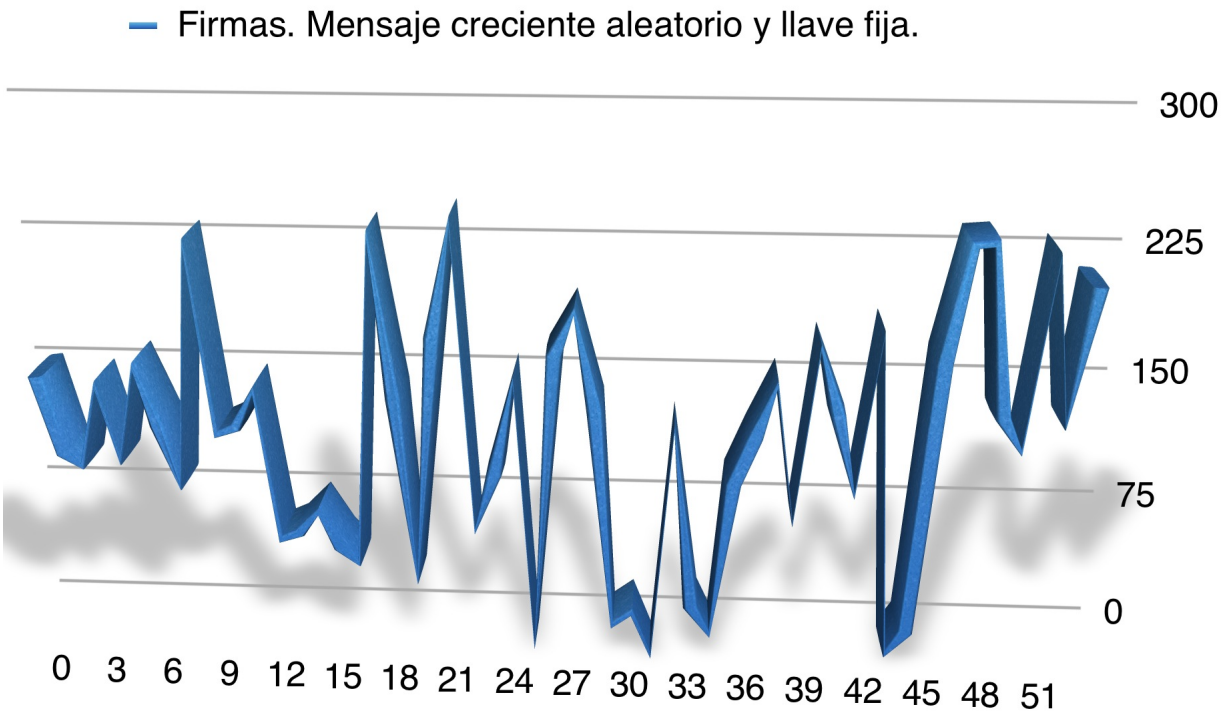
El hecho de que la diferencia de tamaños entre la firma y el archivo a ser firmado es muy grande nos permitirá asignar firmas realmente pequeñas a archivos de a lo más 2097152 bit, es decir, 262144 bytes. Y esto nos lleva a deducir que la firma es 262144/2 o, bien, 131072 veces más corta que el mensaje. El tamaño de la firma es únicamente la primera ventaja ya que la probabilidad de que un atacante tenga éxito falsificando una firma es menor a 2^{-15}, lo cual, para los objetivos iniciales de este proyecto, es altamente efectivo. Esto último, por un lado, es muy conveniente en términos de eficiencia para la implementación y para los requerimientos de seguridad del contexto al cual se está aplicando el esquema; y, por el otro, nos ilustra lo dramático que es el efecto Carter & Wegman.

7.1 Pruebas y trabajo futuro

A continuación presentamos resultados estadísticos de la variación de la firma digital respecto de una cadena de caracteres que tendrá un crecimiento ordenado pero aleatorio, es decir, comenzaremos firmando un estado origen (cadena finita de caracteres) a la cual firmáremos después le concatenáremos un carácter aleatorio, y firmaremos nuevamente. Los resultados serán comparados entre sí para ver la incidencia de colisiones en los resultados de firmado. El objetivo de este análisis será ver tanto qué tan aleatoria resulta ser nuestra implementación del esquema de firmas, como, con qué frecuencia podemos encontrar colisiones. En la gráfica siguiente el eje x esta etiquetado con enteros del 0 al 52, los cuales están asociados a la longitud de la cadena de caracteres que han sido concatenados al estado origen "la tesis esta lista", es decir, el cero está asociado al mensaje inicial, el uno al mensaje "la tesis esta lista$Y$" donde Y será un carácter escogido al azar por la computadora para llevar acabo la prueba y así sucesivamente. El eje y representa los resultados de evaluar una llave fija, en este caso 83912, en un operador polinomial variable determinado por la concatenación aleatoria de un carácter al mensaje, al cual está asociado el operador. Obtenemos así un conjunto de firmas resultante de la evaluación de una llave fija en un conjunto de operadores polinomiales relacionados entre sí. Podemos observar que la gráfica es lo "suficientemente aleatoria", ya que las evaluaciones sucesivas no están relacionadas entre sí, y, de hecho encontrar un patrón para predecir la evaluación que sigue es casi imposible. Para estas primeras 50 evaluaciones el número de colisiones del esquema parece ser nulo. Quedaremos enteramente seguros de esto cuando un adversario intente forjar evaluaciones falsificando firmas, esto lo analizaremos en la tercera prueba de seguridad.

La segunda parte del análisis estadístico consistirá en aplicar un proceso muy similar pero a la llave con la que firmamos el mensaje. De esta manera podremos ver qué tanto afecta la variación de la llave al resultado de aplicar el proceso de firmado a un estado origen fijo.
En la gráfica siguiente podemos observar los resultados de dicho análisis: el eje x representa las llaves que introducíamos al algoritmo de firmado, y el eje y representa los resultados (firmas) de llevar acabo la evaluación de dichas llaves en el operador polinomial en $\mathbb{F}_{2^{16}}[Y]$ que representa un mensaje fijo "la tesis está lista". La gráfica nos permite darnos cuenta de que en las primeras cincuenta evaluaciones de un conjunto de llaves en un operador fijo

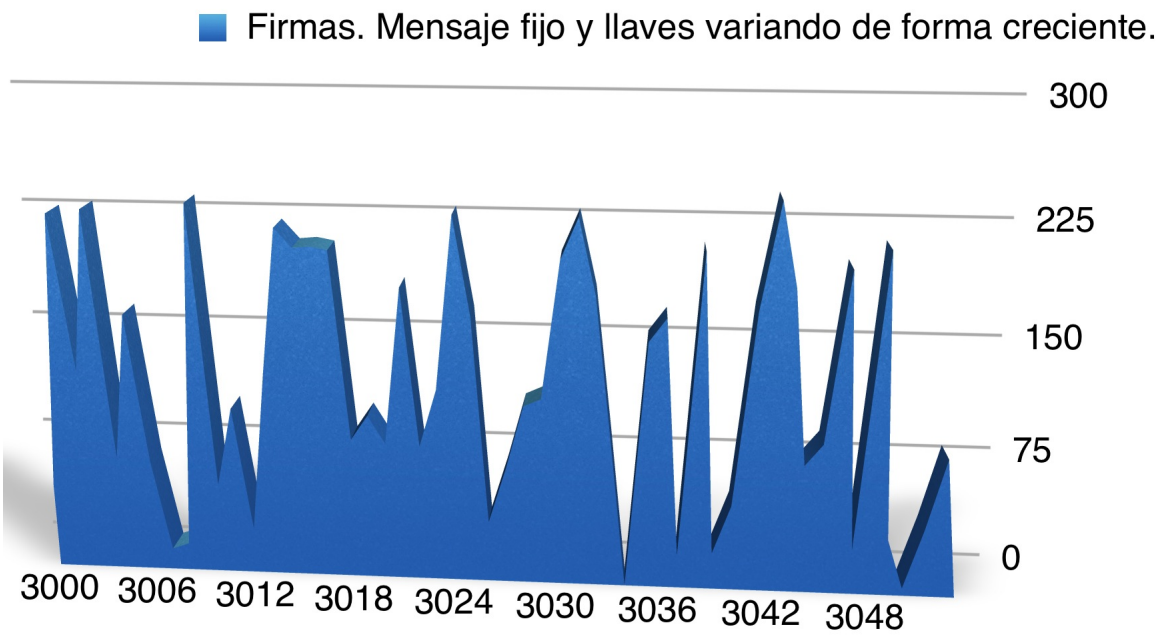
Figura 7.1: Resultados



el número de colisiones es nulo, y, de hecho, en este análisis nuestro esquema de firmas se porta casi como un generador aleatorio de números. Recordemos que el espacio donde habitan las firmas tiene cardinalidad 2^8 mientras que la cardinalidad del espacio donde habitan los mensajes es de 2^{4096}, por la diferencia de tamaños es claro que la existencia de colisiones es inminente, la seguridad de nuestro esquema de firmas digitales con apéndice radicará no en la existencia de colisiones si no en la probabilidad de que un atacante pueda predecir con exactitud si existen colisiones para un mensaje determinado.

La tercera y última prueba de seguridad consistirá en el ataque de un adversario que tratará de llevar nuestro esquema a un estado de ruptura total. Este ataque, como vimos en el capítulo 4, consistirá en un ataque de mensaje adaptable. Es decir, el usuario tendrá pleno acceso a generar tantas firmas como desee para los mensajes que él escoja. Con esta información tratará de forjar una firma que sea aceptada como válida, si el esquema pasa esta

Figura 7.2: Resultados



prueba entonces se le dará al atacante conocimiento de la llave con la que se está generando la firma para darle más herramientas con las que pueda romper el esquema. El atacante será mi asesor de tesis, el Doctor. Octavio Pez Osuna, quien intentará (en un futuro) llevar mi esquema a un estado de ruptura total montando un ataque al mismo instalado en un iPod cuarta generación con sistema operativo iOS 6 y el esquema instalado como un App de prueba.

Nos interesa crear un esquema de firmas con apéndice que sea útil en la práctica, pero para alcanzar esta meta es necesario llevarlo al limite, es decir, debemos montar el ataque más realista posible con el adversario más capacitado posible. Una vez que se hayan pasado las pruebas necesarias para que mi esquema sea considerado seguro, estará al fin listo para ser publicado y salir al mercado de dispositivos móviles como un App para firmar mensajes SMS por lo pronto. La idea final que ya va más allá de esta tesis es hacer que el esquema pueda firmar cualquier tipo de archivo.

Apéndice A

Campos Finitos

Sea p un primo. Entonces $\mathbb{Z}/p\mathbb{Z}$ es un campo finito con p elementos. Lo denotaremos por $\mathbb{F}_p$. Ahora, sea F cualquier campo finito. Denotaremos la suma de n copias de 1 como $n \cdot 1 \in F$. Como F es finito los elementos $n \cdot 1 \in F$ no pueden ser todos distintos. Entonces existe $m < n$ tal que $n \cdot 1 = m \cdot 1$. Se sigue que $(n - m) \cdot 1 = 0$. Denotemos por a al número natural más pequeño tal que $a \cdot 1 = 0$. Al ser F un campo, y en particular un dominio entero, concluimos que $a = p$ debe ser primo. Se sigue por minimalidad que p es el único primo con esta propiedad, y que $n \cdot 1 = 0$ sí y solo sí n es un múltiplo de p. Vemos también que los $i \cdot 1$, $i = 0, 1, \ldots, p - 1$ forman un subcampo de F, isomorfo a $\mathbb{F}_p$. Llamaremos a p la **característica** de F y a $\mathbb{F}_p$, el subcampo generado por 1, su **campo primo**. Entonces todo campo finito F se puede ver como una extensión de su campo primo $\mathbb{F}_p$. Por definición F es un espacio vectorial sobre $\mathbb{F}_p$, por lo que el número de elementos de F es p^n para alguna n (de hecho, $n = [F : \mathbb{F}_p]$).

En el párrafo anterior asumimos que $\mathbb{Z}/p\mathbb{Z}$ es un campo (lo cual no es difícil de ver). Otra manera de demostrar la existencia y unicidad de campos finitos es la siguiente: Sea $q = p^n$, p un primo, n un natural. El campo de descomposición del polinomio $f = x^q - x \in F_p[x]$ es un campo finito con q elementos. La unicidad de F_q se sigue de la unicidad del campo de descomposición de un polinomio. Los elementos de F_q son las raíces (todas distintas) de f. Esto se aclara más adelante.

Para construir directamente campos finitos hacemos uso de polinomios irreducibles: Sea $f(x) \in \mathbb{F}_p[x]$ un polinomio irreducible de grado n. Supongamos que $f(x)$ es mónico, *i.e.* el coeficiente de x^n es 1, de tal forma que $f(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$. Afirmamos que $F = \mathbb{F}_p[x]/(f(x))$, el

anillo cociente de el anillo de polinomios sobre el ideal (maximal) generado por $f(x)$, es un campo con p^n elementos: sea X la imagen de x modulo $(f(x))$, el ideal generado por $f(x)$. Recordemos que $(f(x))$ no es más que el conjunto de polinomios en $\mathbb{F}_p[x]$ que son múltiplos de $f(x)$. Primero que nada observamos que F es un espacio vectorial de dimensión n sobre $\mathbb{F}_p$, de aquí que $|F| = p^n$. Los elementos de F se pueden representar de manera única en la forma $u = \sum_{i=0}^{n-1} c_i X^i$. De hecho, ya que $X^n = -a_{n-1}X^{n-1} - \cdots - a_1 X - a_0$, todo elemento de F tiene esta forma (en pocas palabras F es el conjunto de polinomios en $\mathbb{F}_p[x]$ de grado $< n$). Por otro lado, las X^i, $i = 0, 1, \ldots, n-1$ son linealmente independientes, de lo contrario $f(x)$ dividiría a un polinomio no nulo de grado $< n$, lo cual es imposible. Entonces todo elemento no nulo de F se puede escribir de manera única como polinomio de grado $< n$ con coeficientes en $\mathbb{F}_p$. Supongamos ahora que $h(X), g(X)$ son tales polinomios y que $g(X)h(X) = 0$. Esto querría decir que $f(x)$ divide a $g(x)h(x)$, y al ser $f(x)$ irreducible entonces $f(x)$ divide a $h(x)$ o a $g(x)$ por lo que $h(X) = 0$ o $g(X) = 0$. Esto demuestra que F es un dominio entero (no tiene divisores de cero). Falta demostrar que todo elemento en F tiene un inverso multiplicativo. Sea pues $g(x)$ un polinomio no nulo de grado $< n$. Como $f(x)$ es irreducible, debe ser primo relativo a $g(x)$. Usando el algoritmo de Euclides encontramos polinomios $h(x)$ y $l(x)$ tales que

$$1 = g(x)h(x) + f(x)l(x).$$

Calculando la expresión anterior mod $(f(x))$ obtenemos que $1 = g(X)h(X)$ con lo que habremos encontrado el inverso multiplicativo de $g(X) \in F$.

De ahora en adelante aceptaremos de la teoría de campos el hecho de que una **cerradura algebraica** siempre existe y que está determinada de manera única. Denotaremos por $\mathcal{F}$ a una cerradura algebraica fija de $\mathbb{F}_p$. Esto quiere decir: primero que todo elemento $a \in \mathcal{F}$ es **algebraico** sobre $\mathbb{F}_p$, es decir, que satisface una equación polinomial con coeficientes en $\mathbb{F}_p$; y segundo, que $\mathcal{F}$ es algebraicamente cerrado, es decir, todo polinomio con coeficientes en $\mathcal{F}$ se descompone como producto de factores lineales sobre el mismo campo.

Consideremos de nuevo al polinomio $X^{p^n} - X$. Supongamos la existencia de un campo con p^n elementos. Al ser éste un campo finito, debe ser una extensión algebraica sobre $\mathbb{F}_p$, entonces se puede considerar como un subcampo de $\mathcal{F}$. Como el grupo multiplicativo de este campo tiene $p^n - 1$ elementos, todo elemento no nulo u satisface $u^{p^n-1} = 1$. Por lo que todo elemento de

dicho campo es raíz de dicho polinomio. De esta forma, vemos que un campo con p^n elementos queda determinado de manera única, si es que existe, como cierto subcampo de $\mathcal{F}$. Por otro lado el polinomio $X^{p^n} - X$ es **separable**, es decir sus p^n raices son distintas. Como ya estamos trabajando dentro de un campo, es suficiente demostrar que sumas, productos e inversos multiplicativos de raíces son raíces. Esto es obvio para productos e inversos. Para la suma necesitamos el siguiente lema:

A.1 El automorfismo de Frobenio. Existencia de Campos finitos

Lema A.1.1 (automorfismo de Frobenio)**.** *Sea F un campo de característica p. Entonces el mapeo σ, donde $\sigma(x) = x^p$, es un automorfismo de F en F^p. En el caso que F es un campo finito tenems que $F^p = F$. El campo fijado por σ es $\mathbb{F}_p$.*

Demostración: Sean $x, y \in F$. Tenemos que $\sigma(xy) = x^p y^p = \sigma(x)\sigma(y)$ y $\sigma(x+y) = (x+y)^p$. Usando el teorema del binomio tenemos que

$$(x+y)^p = \sum_{i=0}^{p} \left(\begin{array}{c} p \\ i \end{array} \right) x^i y^{p-i}.$$

Pero $\left(\begin{array}{c} p \\ i \end{array} \right)$ es múltiplo de p si $i \neq 0, p$ por lo que $(x+y)^p = x^p + y^p$ (el sueño de todo estudiante de secundaria). Con esto, $\sigma(x+y) = \sigma(x) + \sigma(y)$. Hemos demostrado que σ abre productos y sumas. ■

Usando el lema A.1.1 podemos demostrar que la suma y el producto de raíces del polinomio $X^{p^n} - X$ son de nuevo raíces. Es decir, demostramos la existencia y unicidad de un campo con p^n elementos.

Resumimos lo anterior en el siguiente

Teorema A.1.1. *Para todo primo p y natural n existe un campo con p^n elementos. Y cada cerradura algebraica $\mathcal{F}$ de $\mathbb{F}_p$ contiene exactamente un subcampo con p^n elementos, consistiendo de las raíces de el polinomio $X^{p^n} - X$. Denotaremos este campo como $\mathbb{F}_{p^n}$*

Apéndice B

Códigos de Reed-Solomon

Sea A un conjunto finito con $v \in \mathbb{N}$ elementos. Un **código** C de **longitud** n sobre el alfabeto A es un subconjunto del producto cartesiano A^n, es decir, $C \subseteq A^n$. A los elementos de C les llamaremos **palabras**.

Dados $a, b \in C$, digamos $a = (a_1, \ldots, a_n)$ y $b = (b_1, \ldots, b_n)$, definimos la **distancia de Hamming** enre las palabras a y b como

$$d(a, b) := |\{i | a_i \neq b_i\}| \tag{B.1}$$

Un parámetro fundamental de un código C es la **distancia mínima** $d := d(C)$ entre sus palabras:

$$d := d(C) := min\{d(a, b) | a, b \in C, a \neq b\}. \tag{B.2}$$

Distinguimos un caso especial: $A = \mathbb{F}_q$ y $C \leq \mathbb{F}_q^n$ es un subespacio vectorial de $\mathbb{F}_q^n$. En este caso decimos que C es un **código lineal**. En este contexto consideramos un parámetro adicional para el código C, su **dimensión** como subespacio vectorial de $\mathbb{F}_q^n$. La denotaremos como $k := dim(C)$.

Si C es un código lineal y $a, b \in C$ entonces $d(a, b) = d(a - b, 0)$. Lo anterior motiva la definición del **peso**, w, de un palabra $a \in C$

$$w(a) := d(a, 0). \tag{B.3}$$

En otras palabras, el peso de la palabra a, de acuerdo a la definición B.1, es el número de sus coordenadas distintas de cero. Por linealidad se sigue que $d = min\{w(a) | a \in C, a \neq 0\}$, es decir la distancia mínima es igual al **peso mínimo**.

Para ahorrar espacio, diremos que C es un $[n, k, d]_q$ código lineal si C tiene longitud n, dimensión k sobre el campo $\mathbb{F}_q$ y distancia mínima d.

Para describir un código lineal $C \leq \mathbb{F}_q^n$ de dimensión k basta con especificar una base. Una **matriz generadora** para el código lineal C es una matriz $k \times n$ cuyos k renglones contienen las coordenadas de los vectores de una base dada. Esta manera de representar a C permite codificar el mensaje $m = (m_1, \ldots, m_k)$ en la palabra

$$m \cdot G.$$

Como C es un subespacio lineal de $\mathbb{F}_q^n$ es conveniente fijarnos el el complemento ortogonal $C^\perp \leq \mathbb{F}_q^n$ de C. Una **matriz de verificación** para el código C es una matriz H cuyos renglones contienen las coordenadas de los vectores de una base para $C^\perp$. Tenemos entonces que

$$c \in C \text{ si y solo si } H \cdot c = 0.$$

De esta manera, al recibir una palabra c, el receptor verifica si hubo errores de transmisión calculando el producto $H \cdot c$. Si el resultado de dicho producto es distinto de cero entonces concluye que hubo errores en la transmisión.

En este curso nos enfocaremos a la construcción de códigos lineales por diversos métodos y seguiremos parte del contenido de los textos [**?**] y [**?**].

B.1 Códigos de Reed-Solomon

La clase de Códigos de Reed-Solomon se considera de gran importancia dentro de la Teoría de Códigos ya que, entre otras propiedades que iremos descubriendo, son miembros de la familia de Códigos de Goppa o Códigos Algebraicos.

Supongamos que $\mathbb{F}_q = \{0, 1, \alpha, \ldots, \alpha^{q-2}\}$. Para $1 \leq k \leq q$ consideramos el espacio vectorial

$$\mathcal{L}_k := \{f \in \mathbb{F}_q[x] | grado(f) < k\} \tag{B.4}$$

y el mapeo $\phi : \mathcal{L}_k \longrightarrow \mathbb{F}_q^q$ dado por

$$\phi(f) := (f(0), f(1), f(\alpha), \ldots, f(\alpha^{q-2})) \in \mathbb{F}_q^q. \tag{B.5}$$

El mapeo lineal B.5 es inyectivo pues un polinomio de grado a lo mas $k-1$ tiene menos de q raíces. Entonces

$$\mathcal{RS}(k,q) := \{(f(0), f(1), f(\alpha), \ldots, f(\alpha^{q-2}))|f \in \mathcal{L}_k\} \tag{B.6}$$

es un Código lineal de longitud q, dimensión k y peso mínimo $q-k+1$.

Bibliografía

[1] Boneh, Dan. *Introduction to Cryptography*. Stanford University, 2012. Coursera. 3 de febrero de 2013.
$< https://www.coursera.org/course/crypto >$;
$< https://www.coursera.org/course/crypto2 >$.

[2] Galaviz Casas José, y Arturo Magidin. *Introducción a la criptología. Vínculos Matemáticos 15.* Ciudad de Mxico: Facultad de Ciencias, UNAM.

[3] Lamport, Leslie. *LaTeX: A Document Preparation System*. Reading, Mass.: Addison-Wesley, 1994.

[4] Menezes, Alfred J., et al. *Hand-Book of Applied Cryptography*. Boca Raton: CRC Press, 1997. Agosto, 1996

[5] Pez Osuna, Octavio. *Notas sobre Teoría de Códigos y Campos de Funciones*. Ciudad de México: Departamento de Matemáticas, Facultad de Ciencias, UNAM, 2012. UNAM. Facultad de Ciencias. 3 de febrero de 2013.
$< http://hp.fciencias.unam.mx// opo/notas/TeoriadeCodigos.pdf >$

[6] Stichtenoth, Henning. *Algebraic Function Fields and Codes*. Berln: Springer, 2009.