UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

POSGRADO EN CIENCIA E INGENIERIA DE LA COMPUTACION

CURVAS ELIPTICAS SOBRE CAMPOS FINITOS DE CARACTERISTICA DOS Y SUS APLICACIONES EN CRIPTOGRAFIA DE CLAVE PUBLICA

T E S I S

QUE PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS

P R E S E N T A :

ALEJANDRO ANDRADE ALVAREZ

DIRECTOR DE LA TESIS:

DR. VLADISLAV K. KHARTCHENKO.

MÉXICO, D. F., 2003

Agradecimientos.

Este proyecto, pudo llevarse a feliz término, gracias al apoyo e interés de aquellas personas que creyeron en mí. El Dr. Vladislav Kirillovich Khartchenko, amplio mis conocimientos sobre álgebra abstracta: en especial, la teoría de grupos y campos finitos. Siempre tuvo la disponibilidad de orientarme y aconsejarme en las situaciones y momentos que yo requería para está tesis.

Mi esposa también fué apoyo y aliciente para culminar está obra.

A la Universidad Nacional Autónoma de México y a la gente que me apoyo, mi agradecimiento y admiración.

¡Gracias!.

Prólogo.

Resulta interesante notar el desarrollo de la tecnología en tan pocos años. Tecnología nueva que ha enriquecido áreas tan diversas como comunicaciones. electrónica, informática, por citar algunos. Pero el manejo y control de está tecnología implica una gran responsabilidad.

Con la evolución de los medios de comunicación electrónica digital apoyados por computadores, se han superado obstáculos físicos y sociales permitiéndonos tener fuentes de información al alcance y con costos cada vez más económicos.

Aunado a esto, información privilegiada sólo para algunos por diversos motivos. también está riesgosamente al alcance de cualquiera que posea los conocimientos necesarios para obtenerla.

El problema es evidente, cuando la privacidad de nuestra información es invadida y el contenido de está conocido por personas no autorizadas y frecuentemente extrañas para nosotros. Los motivos de está invación son muy variados y muchas veces desconocidos.

Con conceptos como: privacidad e intimidad de la información; surgen temas de discusión e investigación muy profundas y de múltiples enfoques. ¡No es posible resumirlas todas en está tesis!.

El motivo personal que me lleva a redactar esta tesis, es el interés que ha despertado en mí desde hace algún tiempo la criptografía; en especial, la criptografía pública o de clave pública. No es porque guarde grandes secretos que atenten contra la seguridad nacional. Más bien, la guía por comenzar y culminar está investigación, es aquel sentimiento de paranoia que todos. unos más que otros, llevamos dentro. Pensar, que cada uno de nosotros, aún podemos decidir que información compartir, y a quienes autorizar conocer nuestra información es gratificante.

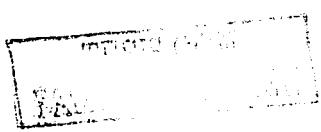
Objetivos.

Por supuesto, existen objetivos más formales que son los que delimítan y dan forma al contenido de está tesis. Estos son:

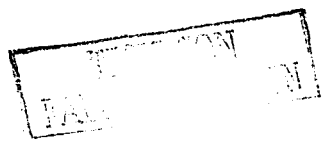
- Planteamiento de los problemas algorítmicos generales para la construcción de criptosistemas de curva elíptica sobre campos finitos de característica dos.
- Describir los algoritmos empleados en cada fase de desarrollo de un criptosistema de curva elíptica sobre un campo finito de característica dos.
- Desarrollar un programa en un lenguaje computacional flexible y eficiente que valide cada algoritmo que interviene en el funcionamiento general de un criptosistema de curva elíptica sobre un campo finito de característica dos.
- Comparar resultados de ejecución del criptosistema con diferentes especificaciones de curvas no-supersingulares sobre campos finitos de característica dos.
- Comparar resultados de ejecución del criptosistema de curva elíptica implementado bajo distintos protocolos criptográficos de clave pública.
- Detallar la relación existente entre la teoría de campos finitos y los algoritmos de un criptosistema de curva elíptica sobre un campo finito de característica dos.

Y por último:

- Resaltar la importancia de buscar nuevos algoritmos que contribuyan a optimizar tiempos de ejecución de los criptosistemas de clave pública, y ofrecer niveles de seguridad cada vez mayores.

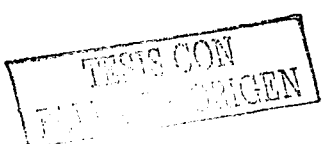


Índice general



Índice de figuras

List of Algorithms



1. Introducción.

Durante los últimos años, el desarrollo del álgebra moderna y en especial el estudio de la teoría de campos finitos, han motivado la búsqueda de aplicaciones prácticas y constructivas que cubran ciertas necesidades tecnológicas actuales. Estas aplicaciones cubren una gran variedad de áreas, pero en algunas de ellas más que en otras, el desarrollo de la teoría de campos finitos es más notable.

Es la criptología, una de las áreas en donde la teoría de campos finitos se aplica eficientemente junto con la teoría de números.

Tuvierón que intervenir diversos factores que llevarón a considerar a la criptografía como el producto de una necesidad tecnológica cada vez más prioritaria. El rápido desarrollo de la tecnología computacional, la competencia de mercado entre fabricantes de está tecnología provocando abaratamiento de equipos de computo, el crecimiento desmesurado de nodos en Internet, el desarrollo de nuevas tecnologías de comunicación, y el intercambio constante de información clasificada de alta seguridad por medios de comunicación públicos, son sólo algúnos de los factores por considerar, sin restar importancia a otros factores de tipo político, militar y social.

La evolución de la criptografía hasta nuestros días, ha permitido extender sus funciones a áreas que hasta hace pocos años la criptografía era limitada o ausente, y sólo era de aprovechamiento exclusivo de militares y diplomáticos. Surgierón también nuevos conceptos que impulsarón esquemas novedosos de criptografía, y sus variantes como la criptografía privada y pública, lograrón incluir dentro de sus funciones una gama de opciones que permiten adaptar un variado criptosistema a las necesidades de un entorno.

1.1. Consideraciones preliminares.

Es cierto, que la tecnología computacional, siempre en desarrollo, le debe en gran medida a las matemáticas su notable evolución; sin menospreciar otras áreas que también intervienen en el proceso. Existe también una relación bilateral entre matemáticas y ciencias computacionales, y como resultado de dicha relación, las matemáticas han obtenido destacado beneficio. Ahora se aprovecha el poder de cálculo que las computadoras proveen para la comprobación de teorías y postulados matemáticos ya existentes. Ese mismo poder de cálculo computacional enriquece aún más las matemáticas al demostrarse nuevas teorías, modeladas mediante algoritmos computacionales; y para estas teorías nuevas, proyectando aplicaciones prácticas en diversas áreas.

Menciono está relación bilateral, porque la criptografía se ha beneficiado de ella.

La evolución de las comunicaciones, ha contribuido a que diversos proyectos sean conocidos en instantes de tiempo cortos, y por una gran cantidad de personas. Ha permitido la colaboración de grupos de trabajo en los mismos proyectos, no importando que los miembros de estos grupos sean de localidades diferentes.

Frecuentemente para evaluar, comprobar y corregir un proyecto, el auxilio de voluntarios entusiastas abate costos y permite mejores resultados. Los parámetros de estas pruebas se enriquecen de la diversidad de circunstancias que proporcionan los voluntarios. Algunos voluntarios inclusive, son capaces de mejorar sustancialmente varios proyectos por sí mismos. El desempeño de pruebas y resultados es aún mejor, cuando el número de voluntarios participantes crece.

Los medios de comunicación masivos como Internet, satisfacen ahora una gama de servicios informáticos, que requieren de normas y estándares de seguridad analizados y probados. Como los cambios en la tecnología computacional y de comunicaciones son muy rápidos, los estándares y normas de seguridad en la información requieren de evaluación constante. Es común que algunos de estos estándares sean modificados, o remplazados por otros, cuando las necesidades de seguridad rebasan las expectativas de estos estándares.

Cuando los avances de la tecnología se diversifica (por el mundo, en un continente, en un país, etc.) y es casi uniforme, es común encontrar proyec-

tos en localidades diferentes; con temáticas, objetivos y elementos similares, aunque no iguales. Tal caso, permite valorar propuestas ya aceptadas con gran objetividad, y permite que la veracidad de pruebas o demostraciones de un proyecto de investigación sean conducidas con mayor exactitud.

El interés por la criptografía, de diversos grupos de investigación, permite cuestionar la efectividad de las técnicas criptográficas existentes, y desarrolla un panorama de colaboración entre estos grupos, para valorar las propuestas que están desarrollándose, y que aún no han sido normalizadas. Al depender la criptografía de la matemática de sus algoritmos, frecuentemente hay un campo rico de propuestas, establecidas o no, por probar y válidar, en proporción a los niveles de seguridad que exige la sociedad en su información y medios de comunicación. ¡Siempre habrá trabajo por hacer!.

1.2. Introducción general.

Las curvas elípticas han sido tema de estudio por matemáticos desde hace más de un siglo. Gracias al estudio continuo de las curvas elípticas, las matemáticas se han enriquecido, y se cuentan ahora de numerosas aplicaciones para las curvas elípticas, resaltando su uso en la factorizacion [L87] y en comprobaciones de primalidad [AM93].

El uso de las curvas elípticas en la criptografía fué propuesta independientemente por Neal Koblitz [K87] y Victor Miller [M85] a mediados de los 80's. Como con todos los criptosistemas, en especial los de clave pública, se consideran años de evaluación pública antes de establecer un nivel aceptable de seguridad y establecer el nuevo criptosistema como un estándar mas.

En años recientes, la criptografía basada en curvas elípticas (ECC[1]) ha recibido especial atención y aceptación para propósitos comerciales, justificándose su inclusión como estándar por organizaciones acreditadas que regulan estándares, tal es el caso de ANSI (American National Standards Institute)[A99. A99-2]. IEEE (Institute of Electrical and Electronics Engineers)[IEEE00], ISO (International Standards Organization)[ISO98, ISO99], y NIST(National Institute of Standards and Technology)[NIST00]. En estas organizaciones, aunque ya consideran el ECC y el ECDSA[2], aún se estudia sus regulaciones

[1]Siglas en inglés: "Elliptic Curve Cryptography".
[2]Siglas en inglés de: "Elliptic Curve Digital Signature Algorithm".

como estándar.

La criptografía basada en curvas elípticas es un producto de la criptografía de clave pública. El esquema de Diffie - Hellman, también conocido como criptografía de clave pública, o simplemente criptografía pública, es un concepto introducido por Whitfield Diffie y Martin Hellman en 1976 [SCH96]. Desde entonces, una gran variedad de criptosistemas se han propuesto como estándares, difiriendo entre ellos, en la estructura matemática utilizada en la que basan su seguridad. La criptografía pública es aún joven, y como suele pasar, se descubren nuevas propiedades o algoritmos para romper los criptosistemas en un tiempo menor al que se había calculado inicialmente. A menudo los resultados que se requieren en criptografía son del tipo que cierta propiedad no vale o que ciertos algoritmos eficientes no se puedan desarrollar [Bel00].

Los primeros criptosistemas estandarizados de clave pública son el RSA, utilizado para encriptación y firma digital, el Diffie-Hellman para acuerdos de claves, y el DSA para firmas digitales. Sin embargo, debido a la aparición en los últimos años de métodos que resuelven el problema matemático en que se basan los algoritmos mencionados en un tiempo menor al que se había calculado inicialmente, se evalúan criptosistemas cuyo problema matemático sea realmente muy difícil de resolver; y de resolverse, el tiempo empleado para ello sea muy grande, y los recursos computacionales para hacerlo sean muy costosos.

La seguridad del ECC está basada en la intratabilidad del problema del logaritmo discreto de la curva elíptica. Si este problema fuerá resuelto de forma eficiente, entonces los criptosistemas basados en ECC serían "rotos" fácilmente.

1.3. Trabajo preliminar.

La diferencia que hay entre los criptosistemas actuales de criptografía de clave pública, y el criptosistema de ECC, es que el segundo se basa en el grupo multiplicativo de puntos dentro de un objeto matemático llamado curva elíptica. No usa números enteros como símbolos del alfabeto del mensaje a encriptar (o firmar). La curva elíptica utilizada, y los elementos que la componen, son parte integral del campo finito en donde la curva elíptica se conforma. Esto quiere decir, que la curva elíptica elegida, está sujeta a los limites del campo

finito que la contiene.

Decidimos trabajar con campos finitos de característica dos (también llamados binarios), ya que sus elementos encajan a la perfección en una palabra de datos de longitud m bits. Explicaremos más adelante la representación de los elementos en campos finítos binarios, y en curvas elípticas.

Hay una serie de recomendaciones hechas por las organizaciones estandarizadoras para la elección de un campo finito, por ejemplo NIST recomienda los siguientes campos binarios: $F_{2^{163}}$, $F_{2^{233}}$, $F_{2^{283}}$, $F_{2^{409}}$ y $F_{2^{571}}$ [H00]. Bajo estas recomendaciones, los criptosistemas presentados trabajan con los campos finitos $F_{2^{283}}$ y $F_{2^{409}}$.

Antes de implementar un criptosistema ECC, es necesario considerar algunas notas previas. La selección de los parámetros del dominio de la curva elíptica. Está selección influye sustancialmente en el desempeño del futuro criptosistema, y depende también de algúnas características técnicas de los equipos de computo anfitriones. Para la selección de la curva elíptica (la aplicación de la ecuación), son consideradas las curvas elípticas aleatorias y las curvas de Koblitz [K99]. Para el desempeño del criptosistema ECC, también juega un papel muy importante la complejidad de los algoritmos computacionales para la aritmética del campo y la aritmética de la curva elíptica: la selección de los algoritmos cuya complejidad computacional es menor, puede garantizar que el rendimiento y eficiencia del criptosistema ECC sea óptimo.

El nivel de seguridad que se pretende alcanzar, y las características físicas de los sistemas de comunicación y computo, son, sin duda, consideraciones que también se deberán de tomar en cuenta en el desarrollo del criptosistema.

Es necesario un lenguaje de programación que permita interpretar en la computadora de forma óptima y eficaz los algoritmos que aquí se explican.

La elección del lenguaje de programación apropiado fué determinada por dos factores: primero, la necesidad de que el código resultante de la interpretación de los algoritmos, sea independiente de la arquitectura de las computadoras en donde el ECC se ejecuta; segundo, que el lenguaje sea flexible al permitir la adaptabilidad de los algoritmos y las estructuras matemáticas que la componen de forma casi integral.

Se ha considerado que el lenguaje C cubre con estas características. Su posición como lenguaje de nivel medio, le permite interpretar el código, sin

sacrificar demasiado los recursos de computo. La portabilidad del código en lenguaje C está garantizada al existir numerosos compiladores del lenguaje C para casí cualquier sistema operativo. Es un lenguaje de programación ampliamente conocido y regulado por ANSI; por ello, nuestro código resultante, no sufrirá de grandes modificaciones en bastante tiempo.

El lenguaje C++ es una extensión del lenguaje C que permite la programación orientada a objetos. La mayoría de los compiladores del lenguaje C, permiten la interacción de instrucciones en lenguaje C++; esto permite que el código tenga una combinación de ambos lenguajes, otorgando así, una flexibilidad mayor en la programación de los algoritmos. Cuando ha sido posible, nos hemos apegado al lenguaje C original, para facilitar la legibilidad y depuración del código.

2. Criptología, criptografía y criptoanálisis.

La criptología es la ciencia de la integridad de la información, [Sim92]. Asociamos a la criptología con el diseño y rompimiento de sistemas para la comunicación secreta. Tales sistemas son llamados criptosistemas, sistemas cifradores, ó simplemente cifradores, [LN86]. El diseño, creación y mantenimiento de sistemas cifradores se denomina criptografía; el rompimiento de tales sistemas cifradores o criptosistemas se le llama criptoanálisis.

El esquema fundamental de un proceso criptográfico (cifrado/descifrado) puede resumirse del modo en que se muestra en la figura 2.1 [FGH01].

Tomamos a A y B como el *emisor* y receptor respectivo de un determinado mensaje. A transforma el mensaje original mediante un procedimiento de cifrado controlado por una *clave*, en un mensaje cifrado *(criptograma)* que se envía por canal público. Por su parte, B como *receptor*, con conocimiento de la clave transforma ese criptograma en el texto fuente, recuperando así la información original.

Un *criptoanalista* puede interceptar el criptograma, y desencriptar el mensaje sin conocimiento de la clave, recuperando el mensaje original.

Por lo tanto, un buen sistema criptográfico será, aquel que ofrezca un descifrado sencillo pero un desencriptado imposible, o muy difícil.

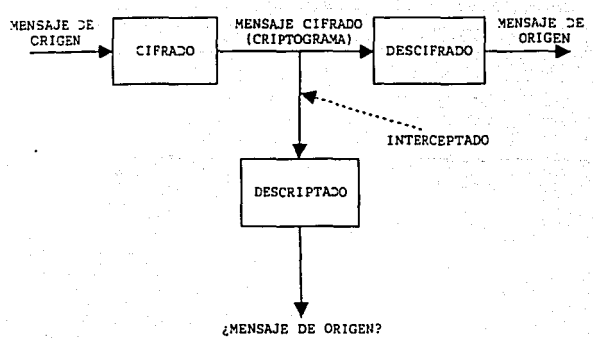


Figura 2.1.: Esquema general de un proceso de encriptación.

2.1. Seguridad de la información y criptografía.

Objetivos de la seguridad de la información. La seguridad de la información, reune una serie de objetivos, en las cuales, algunos de ellos, o la totalidad de ellos, son aplicables de acuerdo a diversas situaciones y circunstancias, [MO97]. Los objetivos que persigue la seguridad de la información son:

- **Privacidad o confidencialidad.** Guardar la información secreta de la vista de todos, pero permitir ver la información a toda persona autorizada para ello.
- **Integridad de los datos.** Asegurar que la información no ha sido alterada por medios no autorizados o desconocidos.
- **Identificación o autenticación de la entidad.** Corroborar la identidad de una entidad quien emite la información (ejemplo: una persona, una terminal de computadora, una tarjeta de crédito, etc.).

- **Autenticación del mensaje.** Corroborar el origen de la información; también conocido como la autenticación del origen.
- **Firma.** Un medio para asociar la información a la entidad que lo emite.
- **Autorización.** Vehículo, para otra entidad, que le permita hacer determinada acción aprobada por la entidad emisora.
- **Validación.** Medios para proveer tiempos de autorización para uso o manipulación de información o recursos.
- **Control de acceso.** Restricción del acceso a recursos sólo a entidades privilegiadas.
- **Certificación.** Garantía de que la información es de una entidad verdadera.
- **Marca de tiempo.** Grabar el tiempo de creación, o de información existente.
- **Testificación.** Verificación de la creación, o de la existencia de información para otra entidad con el creador de la información.
- **Recibo.** Un acuse de recibo de que la información llegó a su destino.
- **Confirmación.** Reconocimiento de que los servicios han sido proveidos.
- **Propiedad.** Un medio para proveer a una entidad de derechos legales para usar o transferir un recurso a otros.
- **Anonimicidad.** Encubrimiento de la identidad de una entidad involucrada en algunos procesos.
- **No-repudio.** Prevenir la denegación de cometidas o acciones previas.
- **Revocación.** Retracción de la certificación o autorización.

Metas de la criptografía. La criptografía puede cubrir algunos de los objetivos de la seguridad de la información, pero no es el único medio para satisfacerlos. La criptografía reune también una serie de metas o fines que satisfacen una parte de los objetivos de la seguridad de la información. Estas son:

1. **Confidencialidad.** Es un servicio usado para guardar el contenido de la información de todos, menos de aquellos que están autorizados para ver está información. El secreto es un sinónimo de confidencialidad, proporcionada por medios físicos o por algoritmos matemáticos para hacer de la información inentendible.

2. **Integridad de los datos.** Mediante ella, es posible detectar la alteración de la información por una entidad no autorizada.

3. **Autentificación.** Identificación de las entidades participantes en una comunicación secreta. Reconocimiento de las entidades autorizadas para ver la información. Incluye también la identificación de entidades y de datos.

4. **No-repudio.** Previene que una entidad deniegue previas acciones o acometidas. Útil para resolver disputas en las cuales una entidad niega ciertas acciones que ya fuerón tomadas, certifica que efectivamente estas acciones fuerón tomadas descubriéndose cierto dólo por parte de la entidad que lo niega.

Una meta fundamental de la criptografía es la de equilibrar estas cuatro áreas en la teoría y práctica. La criptografía trata entonces sobre la prevención y detección del fraude y otras maliciosas actividades con la información.

2.2. Métodos criptográficos.

El tipo de transformación aplicada al texto o las características de las claves utilizadas marcan la diferencia entre los diversos métodos criptográficos. Las características de las claves utilizadas en la codificación y recuperación de la información, clasifican los sistemas criptográficos en dos rubros [FGH01]:

- **Métodos simétricos o de clave secreta:** Son aquellos en los que la clave del cifrado coincide con la de descifrado. Lógicamente, dicha clave tiene que permanecer secreta, lo que presupone que emisor y receptor se han puesto de acuerdo previamente en la determinación de la misma.

- **Métodos asimétricos o de clave pública:** Son aquellos en los que la clave del cifrado es diferente a la del descifrado. En general, la clave del cifrado es conocida por el público, mientras que la de descifrado es conocida únicamente por el usuario.

Los métodos simétricos son propios de la criptografía clásica o criptografía de clave secreta, mientras que los métodos asimétricos corresponden a la criptografía de clave pública, introducida por Diffie y Hellman en 1976 [DH76].

Debido a que está investigación se centra en criptosistemas de clave pública por medio de curvas elípticas, en lo sucesivo sólo mencionaremos lo concerniente a métodos asimétricos y su relación con los criptosistemas de curva elíptica.

2.2.1. Esquema de encriptación de clave pública.

Tenemos a $\{E_e : e \in K\}$, como el conjunto de *transformaciones de encriptación*, y tenemos a $\{D_d : d \in K\}$, como el conjunto de *transformaciones de decriptación*, donde K es nuestro *espacio-clave* [MO97]. Consideramos cualquier par asociado de transformaciones de encriptación y decriptación (E_e, D_d) y supongamos que cada par tiene la propiedad de que conociendo E_e está es computacionalmente intratáble, suponiendo un texto cifrado cualquiera $c \in C$, para encontrar el mensaje $m \in M$ tal que $E_e(m) = c$. Esta propiedad implica que teniendo e está es intratable para determinar la correspondiente llave de desencriptación d. Así, E_e es visto como una *función unidireccional tramposa (trapdoor one-way function)*[1], teniendo a d como el dato *"trampa"* que podría computar la función inversa y lograr la desencriptación del mensaje m.

Para la comunicación entre dos personas, tomamos a Alice y Bob como ejemplo (figura 2.2):

[1] No se ha demostrado aún la existencia de funciones unidireccionales tramposas.

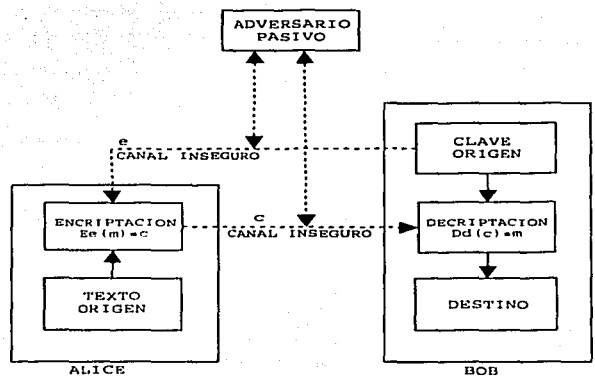


Figura 2.2.: Encriptación usando técnicas de clave pública.

- Bob envía la llave de encriptación e (llamada la clave pública) a Alice sobre un canal inseguro, pero se guarda para si mismo, la clave de desencriptación d (llamada la clave privada) de forma segura y secreta.
- Entonces Alice puede enviar un mensaje m a Bob aplicando la transformación de encriptación determinado por la clave pública de Bob e, de está manera $c = E_e(m)$.
- Bob desencripta el texto cifrado c aplicando la transformación inversa D_d utilizando para ello su clave privada d, para obtener $m = D_d(c)$.

La clave para desencriptar e no necesita ser secreta, puesto que es de conocimiento público. Cualquier persona puede enviar mensajes encriptados a Bob, y sólo el puede desencriptarlos con su clave privada d.

El método de encriptación se dice que es un esquema de encriptación de clave pública si por cada par asociado de encriptación y desencriptación (e, d), una clave e (la *clave pública*) es disponible públicamente, mientras que la clave d (la *clave privada*) es mantenida en secreto. Para que este esquema sea seguro, es necesario que sea computacionalmente *intratable* calcular d de e.

2.3. Problema del logaritmo discreto.

La base del esquema de encriptación con clave pública (de los protocolos criptográficos que se ajustan a este esquema) es la intratabilidad del *problema del logaritmo discreto (DLP[2])*, en la que fundamenta su seguridad. Definimos entonces primero que es un *logaritmo discreto.*

Logaritmo discreto: Sea G un grupo cíclico finito de orden n. Sea α un generador de G, y sea $\beta \in G$. El logaritmo discreto de β a la base α, denotado $\log_\alpha \beta$, es el único entero x, $0 \leq x \leq n-1$, tal que $\beta = \alpha^x$.

Problema del logaritmo discreto generalizado (GDLP[3]). Dados un grupo cíclico finito G de orden n, un generador $\alpha \in G$, y un elemento $\beta \in G$, encontrar el entero x, $0 \leq x \leq n-1$, tal que $\alpha^x = \beta$.

El único requerimiento que se le hace a G para que tenga un uso práctico en criptografía, es que el GDLP se constituya en un problema difícil de resolver en G, mientras que la potenciación sea una operación fácil de ejecutar[Bel00].

El DLP es una *"derivación"* del GDLP, en el cual $G = Z_p^* = \{1, 2, \ldots, p-1\}$, donde p es un número primo y el orden de Z_p^* es $p-1$, de manera que dado un generador $\alpha \in Z_p^*$, y un elemento $\beta \in Z_p^*$, el problema es encontrar un entero x, $0 \leq x \leq p-2$, tal que $\alpha \equiv \beta \pmod p$.

La operación del grupo Z_p^* es la multiplicación modulo p. El GDLP es usado también con grupos de campos finitos, de la forma F_{p^m} con p primo y $m \geq 1$.

El uso del DLP, en un esquema de encriptación de clave pública básico, se trata de la siguiente forma:

- Dos partes, Alice y Bob, quieren compartir un secreto. Alice genera un primo p y un generador $\alpha \in Z_p^*$, $(2 \leq \alpha \leq p-2)$, también genera un número secreto aleatorio x, $1 \leq x \leq p-2$, calcula α^x mód p, y envía a Bob el mensaje $(p, \alpha, \alpha^x \text{ mód } p)$.

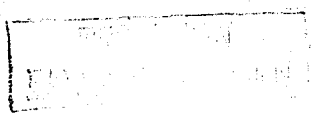
[2]Siglas en ingles: "Discrete Logarithm Problem".
[3]Siglas en ingles de: "General Discrete Logarithm Problem".

- Cuando Bob recibe el mensaje, elije un número aleatorio y, $1 \leq y \leq p-2$, y envía el mensaje (α^y mód p) a Alice.
- Bob calcula el secreto compartido haciendo $K = (\alpha^x)^y$ mód p.
- Alice recibe el mensaje de Bob, y calcula el secreto compartido haciendo $K = (\alpha^y)^x$ mód p.

Un adversario, se encuentra ante el problema de encontrar α^{xy} mód p, dado que conoce el primo p, un generador α de Z_p^* y elementos α^x mód p y α^y mód p, se dice que la solución de este problema, tiene la misma complejidad que la del DLP.

Se puede generalizar el GDLP para uso con cualquier grupo abeliano G y elementos $\alpha, \beta \in G$, donde el problema es encontrar un elemento x tal que $\alpha^x = \beta$, si es que ese elemento existe. En el GDLP no se requiere que G sea cíclico, ni tampoco se requiere α sea un generador de G. Se cree que este problema, es más difícil de resolver que el DLP.

El número de puntos sobre una curva elíptica definida sobre un campo finito, representa especial interés para la criptografía. En los capítulos siguientes, trataremos este grupo y su adaptación al *problema del logaritmo discreto generalizado.*



3. Conceptos Básicos.

Orígenes de la Teoría de Campos Finitos. Los orígenes de la teoría de campos finitos se remontan hacia los siglos XVI y XVII [LN94], con matemáticos como Pierre Fermat (1601 - 1665), Leonard Euler (1707 - 1783), Joseph - Louis Lagrange (1736 - 1813), y Adrien-Marie Legendre (1752 - 1833), contribuyendo en especial a la teoría de la estructura de campos finitos especiales, también llamados campos finitos primos. Pero la teoría general de campos finítos puede decirse que comienza con el trabajo de Carl Friedrich Gauss (1777 - 1855) y Evariste Galois (1811 - 1832). Aunque fué en recientes décadas con la emergencia de instituir matemáticas discretas, que la teoría de campos finitos comenzó a ser estudiada de manera más formal.

3.1. Operaciones Binarias.

Una *operación binaria* $\star$ sobre un conjunto S es un de mapeo $S \times S$ dentro de S. Para cada $(a, b) \in S \times S$, nosotros denotamos el elemento $\star(a, b)$ de S como $a \star b$.

Intuitivamente, podríamos nosotros observar una operación $\star$ sobre S como una asignación, para cada par ordenado (a, b) de elementos de S, como un elemento $a \star b$ de S.

Cerradura y Operaciones Inducidas.

Sea $\star$ una operación binaria sobre S y sea H un subconjunto de S. El subconjunto H es ***cerrado bajo*** $\star$ si para todo $a, b \in H$ nosotros también tenemos $a \star b \in H$. En este caso, la operación binaria sobre H proporciona una restricción $\star$ para H, y esta es la ***operación inducida*** de $\star$ sobre H.

3.2. Grupos.

Un grupo es un conjunto G junto con una binaria operación $\star$ sobre G, tal que las siguientes tres propiedades se satisfacen:

1. $\star$ es *asociativo*; esto es, para cualquier $a, b, c \in G : a \star (b \star c) = (a \star b) \star c$
2. Hay un elemento *identidad (ó unidad)* e en G tal que para todo $a \in G$, $a \star e = e \star a = a$
3. Para cada $a \in G$, existe un elemento inverso $a^{-1} \in G$ tal que $a \star a^{-1} = a^{-1} \star a = e$

Se dice que es un *grupo abeliano* (ó conmutativo) si además satisface:

- Para todo $a, b \in G$, $a \star b = b \star a$

Grupos Finitos. Cada grupo tiene al menos un elemento, nombrado como *la identidad.* Un conjunto mínimo que podría tener lo suficiente para que este sea un grupo, es el conjunto de un elemento $\{e\}$. La única posible operación binaria $\star$ sobre $\{e\}$ está definida como $e \star e = e$. También se cuenta con los tres axiomas de un grupo. El elemento identidad es siempre inversa para sí misma en cada grupo. Si G es un grupo finito, entonces el orden $|G|$ de G es el número de elementos en G. En general, para cualquier conjunto finito S, $|S|$ es el número de elementos en S.

Subgrupos.

Si un subconjunto H de un grupo G es cerrado bajo la operación binaria de G, y si H con la operación inducida de G es así mismo un grupo, entonces H es un subgrupo de G. Nosotros tenemos que $H \leq G$ *ó* $G \geq H$ denota que H es un subgrupo de G, y $H < G$ *ó* $G > H$, nos muestra principalmente que $H \leq G$ *pero* $H \neq G$.

Teorema. Un subconjunto H de un grupo G es un subgrupo de G si y sólo si:

1. H es cerrado bajo la operación binaria de G.
2. La identidad e de G está en H.
3. Para todo $a \in G$ existe también $a^{-1} \in H$ también.

Grupo Cíclico.

Un grupo se llama grupo multiplicativo si su operación $\star$ es multiplicación $a \star b = ab$. En este caso, $\underbrace{a \star a \star a \ldots \star a}_{n}$ tiene denotación a^n. Si está operación es adición, $a \star b = a + b$, entonces el grupo se llama grupo aditivo. En este caso $a \star a \star a \ldots \star a$ tiene denotación na.

Un grupo multiplicativo G se dice que es cíclico si hay un elemento $a \in G$ tal que para cualquier $b \in G$ hay algún entero j con $b = a^j$.

Semejante elemento a es llamado ***generador del grupo cíclico***, y nosotros escribimos $G = <a>$.

Subgrupo Cíclico. Si G es un grupo y $a \in G$, entonces
$H = \{a^n \mid n \in Z\}$
es un subgrupo de G. Este grupo es un *subconjunto cíclico* $<a>$*de* G *generado por* a. También tenemos a un grupo G y un elemento de un grupo G.

3.3. Anillos y Campos.

Anillo.

Un anillo $<R, -, \cdot>$es un conjunto R junto con dos operaciones binarias: las cuales nosotros le llamamos adición y multiplicación, definidos sobre R tal que los siguientes axiomas son satisfechos:

- $<R, +>$ es un grupo abeliano.
- La multiplicación es asociativa.

- Posee las leyes distributivas; esto es para todo $a, b, c \in R$ nosotros tenemos $a \cdot (b + c) = a \cdot b + a \cdot c$, $(b + c) \cdot a = b \cdot a + c \cdot a$

Nosotros usamos 0 (llamado el *elemento cero*) para denotar el elemento *identidad* del grupo abeliano R con respecto a la adición, y la inversa aditiva de a es denotada por $-a$; para $a \cdot b$ nosotros usaremos ab. Como consecuencia de la definición de anillo, se obtiene la propiedad de $a0 = 0a = 0$ para todo $a \in R$. Así esto implica que $(-a)b = a(-b) = -ab$ para todo $a, b \in R$.

Como ejemplo natural de un *anillo* tenemos el anillo de los números enteros ordinarios.

Los anillos pueden ser clasificados de acuerdo a las siguientes definiciones:

1. Un anillo es llamado un ***anillo con identidad*** si el anillo tiene una ***identidad multiplicativa***; esto es, si hay un elemento e tal que $ae = ea = a$ para todo $a \in R$.

2. Un anillo es llamado ***conmutativo*** si la multiplicación es *conmutativa*.

3. Un anillo es llamado un ***dominio integral*** si esta es un ***anillo conmutativo*** con identidad $e \neq 0$ en la que $ab = 0$ implicando que $a = 0$ ó $b = 0$.

4. Un anillo es llamado un ***anillo divisional (ó campo oblicuo)***, si los elementos diferentes de cero de R forman un grupo bajo la multiplicación.

5. Un ***anillo divisional conmutativo*** es llamado ***campo.***

Campo.

De acuerdo con la definición anterior, un campo es un conjunto F en la que dos operaciones binarias, llamadas *adición y multiplicación* son definidas y que contienen dos elementos distinguidos como 0 y e con $0 \neq e$. Además, F es un grupo abeliano con respecto a la adición teniendo 0 como el *elemento identidad*, y los elementos de F que son diferentes de 0 forman un *grupo abeliano* con respecto a la multiplicación, teniendo a e como el *elemento identidad*. Las dos operaciones de adición y multiplicación son unidas por las *leyes*

distributivas de izquierda y derecha $a(b + c) = ab + ac$, $(b + c)a = ba + ca$. El elemento 0 es llamado el *elemento cero* y e es llamado el *elemento identidad multiplicativo*. ó simplemente *la identidad* que será usualmente denotada por 1.

Teorema. Cada dominio integral finito es un *Campo*.

Subanillo. Un subconjunto S de un anillo R es llamado un ***subanillo de*** R, si S es cerrada bajo la multiplicación y adición y forma un anillo bajo estas operaciones.

Ideal. Un subconjunto J de un anillo R es llamado ***un ideal, si*** J; es un subanillo de R y para todo $a \in J$ y $r \in R$ nosotros tenemos $ar \in J$ y $ra \in J$.

Ideales principales. Sea R un *anillo conmutativo*. Un ideal J de R se dice *ideal principal* si hay un elemento $a \in R$ tal que $J = aR$. En este caso, J es también llamado el *ideal principal generado por* a.

Anillo de Clase Residuo (Anillo Factor). El *anillo de clase residuo del anillo* R *modulo* del ideal J, es llamado el ***anillo de clases residuo (ó anillo factor) de*** R ***modulo*** J, y es dentado por R/J.

Teorema. El anillo $Z/(p)$ de clases de residuo de los enteros *modulo* del ideal principal generada por un *primo* p es un *campo*.

Característica de R. Si R es un anillo arbitrario y ahí existe un entero positivo mínimo n tal que $nr = 0$ para cada $r \in R$, entonces el entero positivo n es llamado la *característica de* R, y R tiene *característica (positiva)* n. Si tal entero positivo n no existe, entonces R se dice que tiene *característica* 0.

Teorema. Un dominio integral $R \neq \{0\}$ de característica positiva, tiene una *característica prima*.

Corolario. Un Campo Finito tiene una característica prima.

Teorema. Sea R un anillo conmutativo de característica prima p. Entonces

$(a + b)^{p^n} = a^{p^n} + b^{p^n}$ *y* $(a - b)^{p^n} = a^{p^n} - b^{p^n}$
para $a, b \in R$ y $n \in N$.

Teorema. Sea R un *anillo conmutativo con identidad.* Entonces:

1. Un ideal M de R es un *ideal máximo* si y sólo si R/M es un *campo*
2. Un ideal P de R es un *ideal primo* si y sólo si R/P es un dominio integral.
3. Cada *ideal máximo* de R es un *ideal primo.*
4. Si R es un *ideal de dominio principal,* entonces $R/(c)$ es un *campo* si y sólo si c es un elemento primo de R.

3.4. Polinomios.

Sea R un anillo cualquiera, un polinomio sobre R es una expresión de la forma

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + \ldots + a_n x^n$$

en donde:

- Los coeficientes a_i, $0 \leq i \leq n$, son elementos de R.
- x es un símbolo no conocido para R, llamado la *indeterminante sobre* R.
- $f(x)$ designa a un polinomio.

Los polinomios $f(x) = \sum_{i=0}^{n} a_i x^i$ y $g(x) = \sum_{i=0}^{n} b_i x^i$ sobre R son considerados iguales si y sólo si $a_i = b_i$ para $0 \leq i \leq n$.

Suma de Polinomios. Definiremos la suma de polinomios como :

$$f(x) + g(x) = \sum_{i=0}^{n} (a_i + b_i)x^i$$

Producto de Polinomios. Para definir un producto de dos polinomios sobre R tenemos que $f(x) = \sum_{i=0}^{n} a_i x^i$ y $g(x) = \sum_{j=0}^{m} b_j x^j$ entonces:

$$f(x)g(x) = \sum_{k=0}^{n+m} C_k x^k$$

donde:

$$C_k = \sum_{i+j=k} a_i b_j$$
$$0 \leq i \leq n$$
$$0 \leq j \leq m$$

Definimos entonces, con las operaciones anteriores que el conjunto de los polinomios sobre R forman un anillo.

Anillo Polinomial.

El anillo formado por los polinomios sobre R con las operaciones de adición y multiplicación, es llamado el *anillo polinomial* sobre R y se denota como $R[x]$.

Polinomio Cero. Un polinomio $f(x)$ es llamado *polinomio cero* y denotado por 0. cuando todos sus coeficientes sean 0.

Polinomio Constante y Polinomio Monic. Sea $f(x) = \sum_{i=0}^{n} a_i x^i$ un polinomio sobre R que no sea un polinomio cero; de está forma, suponemos que $a_n \neq 0$. Entonces a_n es llamado el *coeficiente guía* de $f(x)$ y a_0 el *término constante* mientras que n es llamado el *grado* de $f(x)$. Los polinomios de grado ≤ 0 son llamados *polinomios constantes*. Si R tiene la identidad 1 y si el coeficiente guía de $f(x)$ es 1, entonces $f(x)$ es llamado *polinomio monic*[1].

[1]por el momento, me fue prudente dejar el vocablo "monic" del ingles original sin traducción, por no encontrar la palabra equivalente en español que permitiera contener su significado sin modificación.

Teorema. Sea R un anillo. Entonces:

1. $R[x]$ es *conmutativo* si y sólo si R es *conmutativo.*
2. $R[x]$ es un anillo con *identidad* si y sólo si R tiene una *identidad.*
3. $R[x]$ es un *dominio integral* si y sólo si R es un *dominio integral.*

Algoritmo de la División.

Sea $g \neq 0$ un polinomio en $F[x]$. Entonces para cualquier $f \in F[x]$ existen los polinomios $q, r \in F[x]$ tal que

$$f = qg + r$$

donde

$$grado(r) < grado(g)$$

El hecho de que $F[x]$ permita un algoritmo de división implica por un argumento estándar que cada *ideal* de $F[x]$ es *principal.*

Teorema. $F[x]$ es un dominio de ideales principales. Por consiguiente, por cada $J \neq 0$ de $F[x]$ existe un único *polinomio monic determinado* $g \in F[x]$ con $J = (g) = gF[x]$.

Teorema. Sean $f_1, ..., f_n$ polinomios en $F[x]$, de los cuales no todos son 0. Entonces existe un *polinomio monic único determinado* $d \in F[x]$ con las siguientes propiedades: (i) d divide a cada f_j, $1 \leq j \leq n$; (ii) cualquier polinomio $c \in F[x]$ que divide a cada f_j, $1 \leq j \leq n$, divide a d. Además, d puede ser expresado de la forma

$$d = b_1 f_1 + ... + b_n f_n$$

con

$$b_1, ..., b_n \in F[x]$$

Mayor Común Divisor (g.c.d). El polinomio monic d que se menciona en el teorema anterior es llamado el *mayor común divisor* de $f_1, ..., f_n$ que en simbología se expresa así $d = \gcd(f_1, ..., f_n)$. Si $\gcd(f_1, ..., f_n) = 1$, entonces los polinomios $f_1, ..., f_n$ se dice que son *primos relativos*.

Algoritmo de Euclides.

El mayor común divisor de dos polinomios $f, g \in F[x]$ pueden ser calculados por el *algoritmo de Euclides*. Suponiendo que $g \neq 0$ y que g no divide a f. Entonces usamos repetidamente el algoritmo de la división de la siguiente manera:

$f = q_1 g + r_1,\ 0 \leq grado(r_1) < grado(g)$
$g = q_2 r_1 + r_2,\ 0 \leq grado(r_2) < grado(r_1)$
$r_1 = q_3 r_2 + r_3,\ 0 \leq grado(r_3) < grado(r_2)$
.
.
$r_{s-2} = q_s r_{s-1} + r_s,\ 0 \leq grado(r_2) < grado(r_1)$
$r_{s-1} = q_{s+1} r_s$

Donde $q_1, ..., q_{s+1}$ y $r_1, ..., r_s$ son polinomios de $F[x]$, pero $r_1, r_2, \ldots, r_n$ y el grado (g) es finita. Entonces se puede probar que $\gcd(f, g) = rs$. El procedimiento debe detenerse después de un número finito de pasos.

Polinomio Irreducible. Un polinomio $p \in F[x]$ se dice que es *irreducible sobre F (o irreducible en $F[x]$, o primo en $F[x]$)* si p tiene un grado positivo y $p = bc$ con $b, c \in F[x]$ implicando que b o c es un polinomio constante.

Teorema. *(Factorizacion única en $F[x]$).* Cualquier polinomio $f \in F[x]$ de grado positivo puede ser escrito de la forma $f = ap_1^{e_1}, ..., p_k^{e_k}$, donde $a \in F$. $p_1, ..., p_k$ son distintos polinomios monic irreducibles en $F[x]$, y $e_1, ..., e_k$ son enteros positivos. Además está factorizacion es única aparte del orden en la cual los factores ocurren.

Teorema. Para $f \in F[x]$, el anillo de clase de residuo $F[x]/(f)$ es un *campo* si y sólo si f es irreducible sobre F.

Raíz de un Polinomio. Un elemento $b \in F$ es llamado una *raíz (o un cero) del polinomio* $f \in F[x]$ si $f(b) = 0$.

Teorema. Un elemento $b \in F$ es una *raíz del polinomio* $f \in F[x]$ si y sólo si $x - b$ divide $f(x)$.

Raíz Múltiple, Raíz Simple, Multiplicidad de b. Sea $b \in F$ una raíz del polinomio $f \in F[x]$. Si k es un entero positivo tal que $f(x)$ es divisible por $(x-b)^k$, pero no por $(x-b)^{k+1}$, entonces k es nombrado como la *multiplicidad de* b. Si $k = 1$, entonces b es llamado *raíz simple (o cero simple) de* f, y si $k \geq 2$, entonces b es llamado una *raíz múltiple (o un cero múltiple) de* f.

Teorema. Sea $f \in F[x]$ con grado $f = n \geq 0$. Si $b_1, ..., b_m \in F$ son raíces distintas de f con multiplicidad $k_1, ..., k_m$, respectivamente, entonces $(x-b_1)^{k_1}...(x-b_m)^{k_m}$ divide $f(x)$. Consecuentemente, $k_1 + ... + k_m \leq n$, y f no puede tener más que n-distintas raíces en F.

Definición. Si $f(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n \in F[x]$ entonces la derivada f' de f es definida como $f' = f(x) = a_1 + 2a_2x + ... + na_nx^{n-1} \in F[x]$.

Teorema. El elemento $b \in F$ es una raíz múltiple de $f \in F[x]$ si y sólo si esta es una raíz de ambas f y f'.

Teorema. El polinomio $f \in F[x]$ de grado 2 o 3 es irreducible en $F[x]$ si y sólo si f no tiene raíz en F.

Fórmula de Interpolación de Lagrange.

Para $n \geq 0$, sea $a_0, ..., a_n$ siendo $n+1$ elementos distintos de F, y sea $b_0, ..., b_n$ siendo $n+1$ elementos arbitrarios de F. Entonces ahí existe exactamente un polinomio $f \in F[x]$ de grado$\leq n$ tal que $f(a_i) = b_i$ para $i = 0, ..., n$. Este polinomio es obtenido por

$$f(x) = \sum_{i=0}^{n} b_i \prod_{k=0}^{n} (a_i - a_k)^{-1}(x - a_k)$$

para

$$k \neq i$$

Anillo de Polinomios en número finito de variables.

Sea $x_1, x_2, \ldots, x_n$ son variables distintas, podemos considerar construcciones de anillo polinomial sobre F que es $F[x_1]$. Luego podemos construir el anillo polinomial sobre $F[x_1]$ que es $F[x_1, x_2]$. Luego podemos introducir $F[x_1, x_2, x_3]$ como $F[x_1, x_2][x_3]$, etc.

En esta manera tenemos definición del anillo polinomial $F[x]$ sobre el conjunto $x = \{x_1, x_2, \ldots, x_n\}$ de variables.

Sea $f \in F[x_1, ..., x_n]$ dado por $f(x_1, ..., x_n) = \sum a_{i_1, ..._{i_n}} x_1^{i_1} ... x_n^{i_n}$.

Si $a_{i_1 ... i_n} \neq 0$ entonces $a_{i_1 ... i_n} x_1^{i_1} ... x_n^{i_n}$ es llamado un término de f e $i_1 + ... + i_n$ es el grado del término. Para $f \neq 0$ uno define el grado de f, denotado por $grado(f)$, siendo el máximo de los grados de los términos de f. Para $f = 0$ un conjunto $grado(f) = -\infty$. Si $f = 0$ o si todos los términos de f tienen el mismo grado, entonces f es llamado *homogéneo*.

Polinomio Simétrico.

Un polinomio $f \in F[x_1, ..., x_n]$ es llamado *simétrico* si $f(x_{i_1}, ..., x_{i_n}) = f(x_1, ..., x_n)$ para cualquier permutación $i_1, ..., i_n$ de los enteros $1, ..., n$.

Fórmula de Newton.

Sea $\sigma_1, ... \sigma_n$ los elementos de polinomios simétricos en $x_1, ..., x_n$ sobre F, y sea $S_0 = n \in Z$ y $S_k = S_k(x_1, ..., x_n) = x_1^k + ... + x_n^k \in R[x_1, ..., x_n]$ para $k \geq 1$, entonces la fórmula

$$S_k - S_{k-1}\sigma_1 + S_{k-2}\sigma_2 + ... + (-1)^{m-1} S_{k-m+1}\sigma_{m-1} + (-1)^m \frac{m}{n} S_{k-m}\sigma_m = 0$$

teniendo para $k \geq 1$, donde $m = min(k, n)$.

$\sigma_0 = \delta_0 = n$

$\sigma_1 = x_1 + x_2 + x_3 + \ldots + x_n = \delta_1;\ C_n^1 = n$

$\sigma_2 = x_1x_2 + x_1x_3 + \ldots + x_1x_n + x_2x_3 + \ldots + x_{n-1}x_n;\ C_n^2 = \frac{n(n-1)}{2}$

$\sigma_3 = x_1x_2x_3 + x_1x_2x_4 + \ldots + x_{n-2}x_{n-1}x_n;\ C_n^3 = \frac{n(n-1)(n-2)}{2 \cdot 3}$

• • • •

$\sigma_k = x_1x_2 \ldots x_k + \ldots + x_{n-k+1}x_{n-k} \ldots x_n;\ C_n^k = \frac{n!}{k!(n-k)!}$

$\sigma_n = x_1x_2 \cdots x_{n-1}x_n;\ C_n^n = 1$

3.5. Extensiones de Campo.

Sea F un campo. un subconjunto K de F es asimismo un campo bajo las operaciones de F; así, K es llamado el *subcampo de* F. De está forma, F es llamado una *extensión de campo* o simplemente *extensión de* K. Si $K \neq F$, decimos entonces que K es un *subcampo propio de* F.

Denotamos por F_p el campo de clases de residuo de los enteros modulo del ideal principal generadas por un primo p. Este campo tiene p elementos que son $0, 1, 2, \ldots, p-1$. Sea K un subcampo del campo F_p, entonces K tiene al menos los elementos 0 y 1. Por eso todos los elementos $2 = 1+1$, $3 = 1+1+1$, ..., $p-1 = 1+1+\cdots+1$. Entonces el campo F_P no tiene subcampos propios.

Campo Primo. Un campo que tiene subcampos propios es llamado un *campo primo.*

Por las consideraciones anteriores, cualquier campo finito de orden p, $p-$ *primo.* es un *campo primo.*

Extensión Simple.

Sea K un subcampo del campo F y M cualquier subconjunto de F. Entonces el campo $K(M)$ es definida como la intersección de todos los subcampos de F que contengan ambos K y M, y es denominado la extensión de K por medio M. Para el finito $M = \{\theta_1, ..., \theta_n\}$ nosotros escribimos $K(M) = K(\theta_1, ..., \theta_n)$. Si M consiste de un *elemento* $\theta \in F$, entonces $L = K(\theta)$ se dice ser una *extensión simple de* K, *y* θ es denominado como un elemento primitivo definido de L sobre K.

Extensión Algebraica sobre K.

Sea K. un subcampo de F, con $\theta \in F$. Si θ satisface a una *ecuación polinómica no-trivial* con coeficientes en K, esto es, si $a_n\theta^n + ... + a_1\theta + a_0 = 0$ con $a_i \in K$ donde no todos son 0, entonces θ se dice que es *algebraico* sobre K. Una extensión L de K es llamado *algebraico sobre* K *(o una extensión algebraica de* K*)* si cada elemento de L es algebraico sobre K.

Polinomio definido o irreducible.

Si $\theta \in F$ es *algebraico* sobre K, entonces la uniquidad determinada de un *polinomio monic* $g \in K[x]$ que genera el ideal $J = \{f \in K[x] : f(\theta) = 0\}$ de $K[x]$ es llamado el *polinomio definido (o polinomio irreducible, o polinomio*

mínimo) de 0 sobre K. Para el grado de *0* sobre K nosotros utilizamos el grado de g.

espacio vectorial de K.
Si L es un campo de extensión de K, entonces L puede ser visto como un espacio vectorial sobre K.

Grado de L sobre K.
Sea L un campo de extensión de K. Sí L, considerado como un espacio vectorial sobre K, es finito-dimensional, entonces L es llamado una extensión finita de K. La dimensión del espacio vectorial L sobre K es entonces nombrado el grado de L sobre K, en símbolos $[L : K]$.

Teorema. Si L es una extensión finita de K y M, es una extensión finita de L, entonces M es una extensión finita de K con $[M : K] = [M : L][L : K]$.

Teorema. Cada extensión finita de K es algebraica sobre K.

3.6. Campos Finitos.

Consiste en un campo F con un conjunto finito de elementos en él, junto con dos operaciones binarias llamadas adición y multiplicación.
El *orden* de un campo finito es el número de elementos en el campo.

Teorema. (Existencia de un campo finito). Por cada primo p y cada entero positivo n existe un campo finito con p^n elementos.

Del teorema anterior se desprende que existe un campo finito de orden q si y solo si q es una potencia prima. Entonces, si q es una potencia prima, entonces hay esencialmente sólo un campo finito de orden q; que denotamos comúnmente como F_q.
Si $q = p^m$, donde p es un primo, y m es un entero positivo, entonces p es llamado la característica de F_q, y m es llamado el grado de extensión de F_q sobre F_p.

Teorema. Sea F_q el campo finito con $q = p^n$ elementos; entonces cada subcampo de F_q tiene la orden p^m, donde m es un divisor positivo de n. Consecutivamente, si m es un divisor positivo de n, entonces hay exactamente un subcampo de F_q con p^m elementos.

Lema. $(a+b)^p = a^p + b^p$ en cualquier campo de característica p.

Lema. Si F es un campo finito con q elementos, entonces cada $a \in F$ satisface $a^q = a$.

Generadores multiplicativos de campos finitos.

Sabemos que un campo finito está conformado por $q-1$ elementos diferentes de cero, y que estos elementos forman un grupo abeliano bajo la multiplicación. El grupo de los elementos diferentes de cero de F_q es representado por F_q^*.

De está forma, el orden de cualquier elemento de cualquier grupo finito debe dividir el número de elementos de dicho grupo. De manera que, el orden de cualquier $a \in F_q^*$ divide $q-1$.

Definición. Un elemento de un campo finito de orden $q-1$, es llamado generador g si su potencia desarrolla todos los elementos diferentes de cero que conforman dicho campo finito (F_q).

Todos los elementos diferentes de cero de cualquier campo finito forman un grupo cíclico.

Teorema. Cada campo finito tiene un generador, si g es un generador de F_q^*, entonces g^j es también un generador, si y sólo sí $\gcd(j, q-1) = 1$. Así, hay un total de $\varphi(q-1)$ generadores diferentes de F_q^*, donde φ denota la función de Euler.

Corolario. Por cada primo p, existe un entero g, tal que las potencias de g acaba con todas las clases de residuo $no-cero$ mód p.

El campo finito F_p.

Sea p un número primo. El campo finito F_p, llamado un *campo primo*, consiste de el conjunto de enteros $\{0, 1, 2, ..., p-1\}$ con las siguientes operaciones aritmêticas:

- Adición: Si $a, b \in F_p$, entonces $a + b = r$, donde r es el residuo de la división de $a + b$ por p y $0 \leq r \leq p - 1$. Está operación es llamada *adición módulo p.*
- Multiplicación: Si $a, b \in F_p$, entonces $a \cdot b = s$, donde s es el residuo de la división de $a \cdot b$ por p y $0 \leq s \leq p - 1$. Esta operación es llamada *multiplicación módulo p.*
- Inversión: Si a es un elemento diferente de cero en F_p, la inversa de a mód p, denotada como a^{-1}, es el único entero $c \in F_p$ por la cual $a \cdot c = 1$.

El campo finito F_{2^m}.

El campo finito F_{2^m}, llamado campo finito binario, puede ser visto como un espacio vectorial de dimensión m sobre F_2. Esto es, existe un conjunto de m elementos $\{\alpha_0, \alpha_1, ..., \alpha_{m-1}\}$ en F_{2^m} tal que cada $a \in F_{2^m}$ puede ser escrita únicamente en la forma $a = \sum_{i=0}^{m-1} a_i \alpha_i$ donde $a_i \in \{0, 1\}$.

El conjunto $\{\alpha_0, \alpha_1, ..., \alpha_{m-1}\}$ es llamado una base de F_{2^m} sobre F_2. Nosotros podemos entonces representar a como un vector binario $(a_0, a_1, ..., a_{m-1})$. Nosotros ahora introducimos dos de las bases más comunes de F_{2^m} sobre F_2: bases polinomiales y bases normales.

Bases Polinomiales:

Sea $f(x) = x^m + \sum_{i=0}^{m-1} f_i x^i$ (donde $f_i \in \{0, 1\}$, para $i = 0, 1, ..., m - 1$) sea un polinomio irreducible de grado m sobre F_2. Esto es $f(x)$ no puede ser factorizado como un producto de dos polinomios sobre F_2, cada uno de un grado menor que m. De tales polinomios $f(x)$ define una representación de base polinomial de F_{2^m}. De la siguiente manera: consideraremos el anillo polinomial $F_2[x]$ y su ideal I generada por $f(x)$. Tal que $f(x)$ es un polinomio irreducible de grupo m, entonces el anillo factor $F_2[x]/I$ es un campo de dimensión m sobre F_2. Entonces cada campo es isomórfico a F_{2^m}. Esto significa que para cada polinomio irreducible $f(x)$ de grado m sobre F_2 existe un elemento $\theta \in F_{2^m}$ tal que $f(\theta) = 0$ y en el campo F_{2^m}, está comprendido por todos los polinomios sobre F_2 de grado menor que m:

$$F_{2^m} = \{a_{m-1}\theta^{m-1} + a_{m-2}\theta^{m-2} + ... + a_1\theta + a_0 : a_i \in \{0, 1\}\}$$

El elemento de campo $a_{m-1}\theta^{m-1} + a_{m-2}\theta^{m-2} + \ldots + a_1\theta + a_0$ es usualmente denotado por el segmento de cadena $(a_{m-1}a_{m-2}\ldots a_1a_0)$ de longitud m, así que

$$F_{2^m} = \{(a_{m-1}a_{m-2}\ldots a_1a_0) : a_i \in \{0,1\}\}$$

Así los elementos F_{2^m} pueden ser representados por el conjunto de todas las cadenas binarias de longitud m. El elemento de identidad multiplicativa (1) es representado por el segmento de cadena (00..,01), mientras que el elemento identidad aditiva (0) es representado por el segmento de cadena que comprende a todos los ceros. En esta representación el polinomio $f(x)$ es llamado la reducción polinomial, tal que $\theta^m = -\sum_{i=0}^{m-1} f_i\theta^i$ y está formula define la multiplicación sobre polinomios θ.

Operaciones de campo: Las siguientes operaciones aritméticas son definidas sobre los elementos de F_{2^m} cuando usamos una representación de base polinomial con reducción polinomial $f(x)$:

- Adición: Sí $a = (a_{m-1}a_{m-2}\ldots a_1a_0)$ y $b = (b_{m-1}b_{m-2}\ldots b_1b_0)$ son elementos de F_{2^m}, entonces $a + b = c = (c_{m-1}c_{m-2}\ldots c_1c_0)$, donde $c_i = (a_i + b_i)$ mód 2. Esto es, la adición de campo es desarrollada a pequeña discreción.

- Multiplicación: Sí $a = (a_{m-1}a_{m-2}\ldots a_1a_0)$ y $b = (b_{m-1}b_{m-2}\ldots b_1b_0)$ son elementos de F_{2^m}, entonces $a \cdot b = r = (r_{m-1}r_{m-2}\ldots r_1r_0)$, donde el polinomio $r_{m-1}x^{m-1} + r_{m-2}x^{m-2} + \cdots + r_1x + r_0$ es el residuo cuando el polinomio de grado $2m-2$

$$\left(a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0\right) \cdot$$

$$\left(b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + a_1x + a_0\right)$$

  es dividida por $f(x)$ sobre F_2.

- Inversión: Si a es un elemento no-cero en F_{2^m}, la inversa de a, denotada a^{-1}, es el único elemento $c \in F_{2^m}$ por la cual $a \cdot c = 1$.

Bases Normales:

Una base normal de F_{2^m} sobre F_2 es una base de la forma $\left\{\beta, \beta^2, \beta^{2^2}, \beta^{2^{m-1}}\right\}$, donde $\beta \in F_{2^m}$. Tal base siempre existe. Cualquier elemento $a \in F_{2^m}$ puede ser escrito como $a = \sum_{i=0}^{m-1} a_i \beta^{2^i}$, donde $a_i \in \{0, 1\}$. Las representaciones de base normal tienen la ventaja computacional que equitativamente un elemento puede hacerse muy eficientemente, multiplicando elementos distintos, y por el otro lado, puede ser por lo general engorroso. Por está razón, ANSI[2] X9.62 especifica que bases normales gaussianas sean usadas, por las cuales la multiplicación es simple y más eficiente.

Bases Normales Gaussianas: El tipo de un GNB[3] es un entero positivo que mide la complejidad de la operación multiplicativa con respecto a esta base. Generalmente hablando el tipo más pequeño, es el más eficiente para la multiplicación. Para la obtención de m y T, el campo F_{2^m} pueden tener más de un GNB de tipo T. Así, de está forma se puede hablar del tipo T de GNB del F_{2^m}.

Existencia de Bases Normales Gaussianas: Una base normal gaussiana (GNB) existe siempre que m no sea divisible por 8. Sea m un entero positivo no divisible por 8, y sea T un entero positivo; entonces un tipo T de GNB para F_{2^m} existe si y sólo sí $p = Tm + 1$ es primo y el $\gcd(Tm/K, m) = 1$, donde K es el orden multiplicativo de 2 mód p.

Elementos de campo: Sí $\left\{\beta, \beta^2, \beta^{2^2}, \beta^{2^{m-1}}\right\}$ es una base normal de F_{2^m} sobre F_2, entonces el elemento de campo $a = \sum_{i=0}^{m-1} a_i \beta^{2^i}$ es representado por la cadena binaria $(a_0 a_1 \ldots a_{m-1})$ de longitud m, así que

$$F_{2^m} = \{(a_0 a_1 \ldots a_{m-1}) : a_i \in \{0, 1\}\}$$

El elemento de identidad multiplicativa (1) es por el segmento de cadena de todos los $1's$, mientras que el elemento de identidad aditiva (0) es representado por el segmento de cadena de todos los $0's$.

[2] Abreviatura para "American National Standards Institute" (anteriormente ASA y USAS), una organización que desarrolla y pública normas para la industria.

[3] Siglas del concepto original: "Gaussian Normal Bases".

3. Conceptos Básicos.

Operaciones de campo: Las operaciones aritméticas siguientes son definidas sobre los elementos de F_{2^m} cuando usamos un GNB de tipo T:

- Adición: Sí $a = (a_0 a_1 \dots a_{m-2} a_{m-1})$ y $b = (b_0 b_1 \dots b_{m-2} b_{m-1})$ son elementos de F_{2^m}, entonces $a + b = c = (c_0 c_1 \dots c_{m-2} c_{m-1})$, donde $c_i = (a_i + b_i)$ mód 2.

- Cuadrado: Sea $a = (a_0 a_1 \dots a_{m-2} a_{m-1}) \in F_{2^m}$. Donde el cuadrado es una operación lineal en F_{2^m},

$$a^2 = \left(\sum_{i=0}^{m-1} a_i \beta^{2^i} \right) = \sum_{i=0}^{m-1} a_i \beta^{2^{i+1}} = \sum_{i=0}^{m-1} a_{i-1} \beta^{2^i} = (a_{m-1} a_0 a_1 \dots a_{m-2})$$

con índices reducidos a mód m. Aquí, el cuadrado de un elemento de campo puede ser realizado por una rotación simple de la representación del vector.

- Multiplicación: Sea $p = Tm + 1$ y sea $\mu \in F_p$ un elemento de orden T. Defina la secuencia $F(1), F(2), \cdots, F(p-1)$ por

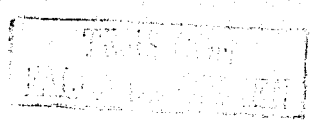
$$F\left(2^i \mu^i \bmod p\right) = j$$

para $0 \leq i \leq m$, $0 \leq j \leq T - 1$. Si $a = (a_0 a_1 \dots a_{m-2} a_{m-1})$ y $b = (b_0 b_1 \dots b_{m-2} b_{m-1})$ son elementos F_{2^m}, entonces $a \cdot b = c = (c_0 c_1 \dots c_{m-2} c_{m-1})$, donde

$$c_j = \begin{cases} \sum_{k=1}^{p-2} a_{F(k+1)+l^b F(p-k)+l} & (si\ T\ es\ par) \\ \sum_{k=1}^{m/2} (a_{k-l-1} b_{m/2+k+l-1} + a_{m/2+k+l-1} b_{k+l-1}) + \sum_{k=1}^{p-2} a_{F(k+1)+l^b F(p-k)+l} & (si\ T\ es\ impar) \end{cases}$$

por cada l, $0 \leq l \leq m-1$, donde los índices son reducidos a mód m.

- Inversión: Si a es un elemento no-cero en F_{2^m}, la inversa de a en F_{2^m}, denotado como a^{-1}, es el elemento único $c \in F_{2^m}$ por cada $a \cdot c = 1$.



4. Estructura de las curvas elípticas.

4.1. Introducción.

Las curvas algebraicas han sido estudiadas extensivamente desde hace 150 años. Sin embargo, el interés por emplear estas estructuras para propósitos criptográficos fue planteada hace poco más de 15 años. El empleo de las curvas algebraicas, en especial las curvas elípticas para su uso en la criptografía moderna, fue propuesto por Koblitz y Miller en forma independiente. Ellos propusieron el uso de un grupo de puntos de una curva elíptica o hiperelíptica definido sobre un campo finito. Estas curvas elípticas o hiperelípticas ofrecen una gran cantidad de grupos abelianos que enriquecen en gran medida estas estructuras algebraicas. Además, ofrecen como cualquier sistema criptográfico de clave pública, la misma seguridad pero empleando claves más pequeñas [Bea96].

Hay otras ventajas importantes por considerar en el subgrupo cíclico de puntos en las curvas elípticas e hiperelípticas sobre el grupo multiplicativo de campos finitos usados en esquemas de Diffie-Hellman y otros algoritmos actuales. Estas podrían ser:

- El problema del logaritmo discreto en el grupo de las curvas elípticas es computacionalmente más complejo que aquellos tomados directamente del logaritmo discreto sobre campos finitos [Kaz92].
- La estructura del grupo de puntos sobre las curvas elípticas que contendrán los parámetros del criptosistema, se escogen con más flexibilidad.

- Al ser las curvas elípticas análogas al grupo multiplicativo de los campos finitos, estas mismas ofrecen más flexibilidad para seleccionarse (escoger la curva elíptica ideal) que en escoger un campo finito [BSS1999].

Las curvas elípticas sobre campos finitos F_{2^m} tienen un interés particular, ya que sus elementos encajan a la perfección en una palabra de datos de longitud m bits. los criptosistemas elípticos implementados sobre estos campos, muestran cierto potencial para proveer pequeños criptosistemas de clave pública a bajo costo.

Curvas Elípticas Generales.

Las curvas elípticas fueron encontradas como resultado del estudio al problema de calcular la longitud del arco de una elipse. Para calcular dicha longitud del arco, uno integra una función que involucra $y = \sqrt{f(x)}$, y la respuesta es obtenida en términos de ciertas funciones sobre la "curva elíptica" $y^2 = f(x)$.

Las curvas elípticas existen sobre cualquier campo (por ejemplo: reales, complejos, finitos, etc.) pero para propósitos criptográficos solamente consideramos las curvas elípticas comprendidas dentro de campos finitos [Bea96].

Asumimos que K es un campo con característica (k). Una curva elíptica sobre K se definirá como el conjunto de soluciones en el espacio proyectivo de una ecuación homogénea de Weierstrass, de la forma:

$$E : y^2z + a_1xyz + a_3yz^2 = x^3 + a_2x^2z + a_4xz^2 + a_6z^3 \quad (1)$$

en donde $a_1, a_2, a_3, a_4, a_6 \in K$.

Existe un punto en E con coordenada Z igual a cero, es decir $(0, 1, 0)$. A este punto le llamamos punto al infinito, El cual es denotado por nosotros como σ.

Por conveniencia, afinaremos la *ecuación general de Weierstrass* transformando las coordenadas del espacio proyectivo $x = x/z$, y $y = y/z$ y junto con el especial punto al infinito (σ), tenemos:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad (2)$$

en donde $a_i \in K$.

Así. $E(K)$ denota el conjunto de puntos $(x, y) \in K^2$ que satisface está ecuación, junto con un *punto al infinito* denotado como σ.

4.2. Supersingularidad y No-singularidad de una Curva Elíptica.

Tomando entonces la misma ecuación general de Weierstrass en su forma afinada, definimos entonces las siguientes constantes para emplearlas en una fórmula posterior [Borst97]:

$$\begin{gathered} b_2 = a_1^2 + 4a_2 \\ b_4 = a_1 a_3 + 2a_4 \\ b_6 = a_3^2 + 4a_6 \\ b_8 = a_1^2 a_6 + 4a_2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 - a_4^2 \\ c_4 = b_2^2 - 24b_4 \\ c_6 = -b_2^3 + 36 b_2 b_4 - 216 b_6 \end{gathered}$$

El discriminante de la curva es definida como:

$$\Delta = -b_2^2 b_8 - 8b_4^3 - 27b_6^2 + 9b_2 b_4 b_6$$

Una curva es entonces *no-singular* si y sólo sí $\Delta \neq 0$.
Cuando $\Delta \neq 0$, el j-invariante de la curva es definido como:

$$j(E) = \frac{c_4^3}{\Delta}$$

Cuando $j(E) = 0$, la curva E se dice que es *supersingular*.

4.3. Leyes de Adición.

Para un mayor entendimiento de como los puntos de una curva forman un grupo abeliano; primero consideraremos las reglas aditivas de estas curvas elípticas sobre el campo de los reales, y después mostraremos las modificaciones y comparaciones con las curvas elípticas aplicadas a campos finitos de característica dos [K99].

Tenemos una curva elíptica E con la *ecuación general de Weierstrass* (2) sobre el campo de los números reales; y tenemos a P y Q como dos puntos sobre E. Definimos entonces el negativo P y la suma $P + Q$ de acuerdo a las siguientes reglas:

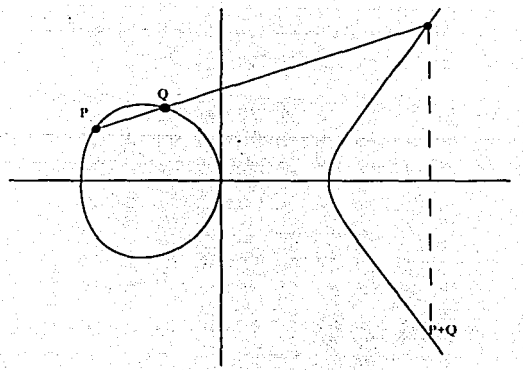


Figura 4.1.: Suma de los puntos P y Q en una curva elíptica.

1. Si P es el punto al infinito σ, entonces nosotros definimos a $-P$ como σ. Para cualquier punto Q nosotros definimos $\sigma + Q$ como Q; es decir, σ sirve como la identidad aditiva (el elemento cero) de el grupo de puntos.

2. El negativo $-P$ es el punto con las mismas coordenadas en $X's$ como P, pero con la coordenada en las $Y's$ negativa; esto es, $-(x, y) = (x, -y)$. Si $Q = -P$, entonces nosotros decimos que $P + Q$ es el punto al infinito σ.

3. Si P y Q tienen diferentes coordenadas en el eje de las $X's$, entonces mostraremos que la linea $l = \overline{PQ}$ intersecta a la curva exactamente uno o más puntos R (al menos que l sea tangente a la curva en P, en tal caso nosotros hacemos $R = P$, ó $R = Q$). Entonces definimos $P+Q = -R$, es decir, la imagen espejo (con respecto al eje de las $X's$) del tercer punto de intersección (fig 4.1).

4. Si $P = (x_1, y_1) \neq 0$, entonces obtendremos: Para curvas supersingulares $-P = (x_1, -y_1)$; para curvas no-supersingulares $-P = (x_1, x_1 + y_1)$.

5. La última posibilidad es que $P = Q$. Entonces tenemos a l como una

línea tangente a la curva en P, sea R el otro y único punto de intersección de l con la curva, y definimos $2P = -R$. (R es tomado como P si la línea tangente tiene una *doble tangencia en* P, en otras palabras, si P es un punto de inflexión).

4.4. Curvas Elípticas sobre el campo F_{2^m}.

Recordemos que para propósitos criptográficos trataremos sólo las curvas elípticas sobre campos de característica dos.

Para cualquier campo finito de característica dos existen 2^{2k} curvas elípticas diferentes que pueden ser usadas para criptosistemas.

Llamamos a una curva elíptica sobre el campo F_{2^m} "supersingular" cuando $j(E) = 0$ con la siguiente ecuación:

$$y^2 + a_3 y = x^3 + a_4 x + a_6 \qquad Supersingular \quad (3)$$

Por el contrario, cuando $j(E) \neq 0$ llamamos a una curva elíptica sobre el campo F_{2^m} "no-supersingular" con la siguiente ecuación:

$$y^2 + xy = x^3 + a_2 x^2 + a_6 \qquad No - supersingular \quad (4)$$

4.4.1. Leyes de Adición.

Definiremos la adición para una curva E sobre un campo finito F_{2^m}, considerando que $P = (x_1, y_1) \in E$; entonces $-P = (x_1, -y_1)$. $P + \sigma = \sigma + P = P$ para todo $P \in E$; $Q = (x_2, y_2) \in E$, y $Q \neq -P$, entonces $P + Q = (x_3, y_3)$, donde:

1. El caso *supersingular* (3):

$$x_3 = \left(\frac{y_1+y_2}{x_1+x_2}\right)^2 + x_1 + x_2, \quad y_3 = \left(\frac{y_1+y_2}{x_1+x_2}\right)(x_1 + x_3) + y_1 + a_3,$$

cuando adicionamos puntos distintos ($P \neq Q$); y

$$x_3 = \frac{x_1^4 + a_4^2}{a_3^2}, \qquad y_3 = \frac{x_1^2 + a_4}{a_3}(x_1 + x_3) + y_1 + a_3$$

cuando doblamos un punto ($P = Q$).

2. El caso *no-supersingular* (4):

$$x_3 = \left(\frac{y_1+y_2}{x_1+x_2}\right)^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a_2, \quad y_3 = \left(\frac{y_1+y_2}{x_1+x_2}\right)(x_1 + x_3) + x_3 + y_1,$$

cuando adicionamos puntos distintos ($P \neq Q$); y

$$x_3 = x_1^2 + \frac{a_6}{x_1^2}, \qquad y_3 = x_1^2 + \left(x_1 + \frac{y_1}{x_1}\right)x_3 + x_3,$$

cuando doblamos un punto ($P = Q$).

Teorema. Para una curva elíptica E sobre el campo F_{2^m} con $j(E) \neq 0$, que tiene los parámetros a_2 y $a_6 \in F_{2^m}$ se tiene la siguiente propiedad:

- La adición de dos puntos diferentes sobre la curva E es independiente de a_6.

Teorema. Para una curva elíptica E sobre el campo F_{2^m} con $j(E) = 0$, que tiene parámetros a_3, a_4 y $a_6 \in F_{2^m}$ se tiene las siguientes propiedades:

- La adición de dos puntos diferentes sobre la curva es independiente de a_4 y a_6.
- La duplicación de un punto es independiente de a_6.

5. Algoritmos para la curva elíptica.

El tratamiento de la estructura de la curva elíptica sobre campos finitos binarios, es proporcionada por los algoritmos que determinan la eficiencia y complejidad del criptosistema de curva elíptica planteado. Algunos algoritmos son ideales bajo conceptos de velocidad y desempeño. En cambio, otros algoritmos ofrecen una fácil adaptación al código que los interpreta. Lo ideal es elegir aquel algoritmo que cubra con las mejores especificaciones, que sea flexible y adaptable al código, y que proporcione la rapidez deseada sin sacrificar desempeño y seguridad.

5.1. Aritmética sobre campos binarios.

Los algoritmos encargados de la aritmética sobre campos binarios, han sido demostrados y desarrollados para la arquitectura de *32-bits*. Esto es, las palabras o bloques de bits representados con la letra W son numerados del 0 al 31, con el bit significativo a extrema derecha designado como el *bit* 0.

5.1.1. Representación de campos.

El campo F_{2^m} tiene numerosas representaciones. Sobresaliendo las *bases normales (normal bases)* y las *bases polinomiales (polinomial bases)*. La primera representación ha demostrado ser muy eficiente cuando se construyen criptosistemas directamente en hardware. En ella, la operación de elevar al cuadrado es trivial. La segunda representación, ofrece una implementación más simple y rápida en software. Los polinomios pueden ser representados como una cadena

de bits continua dejando que cada bit represente la posición del coeficiente de una potencia de x.

Como deseamos implementar un criptosistema de curva elíptica en software, sólo haremos referencia a los algoritmos encargados de la aritmética de los campos binarios que son construidos bajo la representación de bases polinomiales.

Tenemos a $f(x) = x^m + r(x)$ como el polinomio binario irreducible de grado m. Los elementos de F_{2^m} son los polinomios binarios de grado $m-1$, con la adición y multiplicación ejecutados bajo el modulo $f(x)$. Un elemento de campo $a(x) = a_{m-1}x^{m-1} + \ldots + a_2x^2 + a_1x + a_0$ es asociado con el vector binario $a = (a_{m-1}, \ldots, a_2, a_1, a_0)$ de longitud m. Tenemos que $t = \lceil m/32 \rceil$, y que $s = 32t - m$. En código, nosotros almacenamos a en un arreglo de t palabras de 32-bits: $A = (A[t-1], \ldots, A[2], A[1], A[0])$, donde el bit de extrema derecha de $A[0]$ es a_0, y los s bits de extrema izquierda de $A[t-1]$ no son usados (siempre cambian a 0).

La adición de los elementos de campo es ejecutada *bit por bit*; así, sólo se requieren t operaciones de palabras.

5.1.2. Multiplicación.

El método *shift-and-add (cambia y añade)* para la multiplicación de los ele-

Algorithm 1 Multiplicación de campo: Método shift-and-add.
Entrada: Polinomios binarios $a(x)$ y $b(x)$ de grado $m-1$.
Salida: $c(x) = a(x) \cdot b(x)$ mód $f(x)$.

1. Si $a_0 = 1$ entonces $c \leftarrow b$; de lo contrario $c \leftarrow 0$.
2. Para i de 1 hasta $m-1$ hacer
   a) $b \leftarrow b \cdot x$ mód $f(x)$.
   b) Si $a_i = 1$ entonces $c \leftarrow c + b$.
3. Regresar(c).

mentos del campo. esta basado sobre la observación de que $a \cdot b = a_{m-1}x^{m-1}b +$

$\cdots + a_2x^2b + a_1xb + a_0b$. La interacción i del algoritmo calcula x^ib mód $f(x)$ y añade el resultado al acumulador c si $a_i = 1$.

Este algoritmo tiene buen desempeño en hardware, donde el vector cambiante puede ser ejecutado en un ciclo de reloj. Sin embargo, este algoritmo no es muy viable en software, ya que su desempeño es lento porque el número de cambios de palabra es grande.

Hay otros métodos de multiplicación más rápidos que primero multiplican los elementos del campo como polinomios, y el resultado se reduce a través del modulo $f(x)$.

Multiplicación polinomial. El método *comb (panal)* para la multiplicación

---

Algorithm 2 Multiplicación polinomial: Método comb (panal).

Entrada: Polinomios binarios $a(x)$ y $b(x)$ de grado $m-1$.
Salida: $c(x) = a(x) \cdot b(x)$.

1. $C \leftarrow 0$.
2. Para k de 0 hasta 31 hacer
   a) Para j de 0 hasta $t-1$ hacer
      1) Si el k bit de $A[j]$ es 1 entonces añadir B a $C\{j\}$.
   b) Si $k \neq 31$ entonces $B \leftarrow B \cdot x$.
3. Regresar(C).

---

polinomial se basa en la observación de que $b(x) \cdot x^k$ ha sido calculado para el mismo $k \in [0, 31]$, entonces $b(x) \cdot x^{32j+k}$ puede ser fácilmente obtenido anexando j *cero* palabras a la derecha de la representación del vector de $b(x) \cdot x^k$. La siguiente notación es usada: si $C = (C[n], \ldots, C[2], C[1], C[0])$ es un vector, entonces $C\{j\}$ describe el vector truncado $(C[n], \ldots, C[j+1], C[j])$.

El *método comb* puede ser agilizado, precalculando $u(x) \cdot b(x)$ para todos los polinomios $u(x)$ de grado menor que w, donde w divide la longitud de la palabra, y considera los bits de los $A[j]$'s w al mismo tiempo. Este méto-

do modificado, con $w = 4$ se muestra como: ***algoritmo de método comb con ventanas de ancho*** $w = 4$.

Algorithm 3 Multiplicación polinomial: Método comb (panal) con ventanas de ancho $w = 4$.

Entrada: Polinomios binarios $a(x)$ y $b(x)$ de grado $m - 1$.
Salida: $c(x) = a(x) \cdot b(x)$.

1. Calcular $B_u = u(x) \cdot b(x)$ para todos los polinomios $u(x)$ de grado no más de 3.
2. $C \leftarrow 0$.
3. Para k de 7 hasta 0 hacer:
   a) Para j de 0 a $t - 1$ hacer:
      1) Teniendo $u = (u_3, u_2, u_1, u_0)$, donde u_i es el bit $(4k + i)$ de $A[j]$. Añadir B_u a $C\{j\}$.
   b) Si $k \neq 0$ entonces $C \leftarrow C \cdot x^4$.
4. Regresar(C).

El método anterior, fue descrito por Karatsuba para la multiplicación de enteros [Knuth98]. En donde se multiplican dos polinomios $a(x)$ y $b(x)$ de grado $m - 1$; suponiendo que m es par. Cada polinomio se divide en dos polinomios de grado no más de $(m/2) - 1$: $a(x) = A_1(x) X + A_0(x)$, $b(x) = B_1(x) X + B_0(x)$, donde $X = x^{m/2}$. Entonces

$$a(x) b(x) = A_1 B_1 X^2 + [(A_1 + A_0)(B_1 + B_0) + A_1 B_1 + A_0 B_0] X + A_0 B_0,$$

las cuales pueden ser derivadas de los tres productos de polinomios de grado $(m/2) - 1$.

Reducción. Sea $c(x)$ un polinomio binario de grado no más de $2m - 2$. El siguiente algoritmo reduce $c(x)$ modulo $f(x)$ un bit a la vez, comenzando con el bit de extrema izquierda. El algoritmo se basa en la observación que

$x_i \equiv x^{i-m} r(x) (\text{mód} f(x))$ para $i \geq m$. Los polinomios $x^k r(x)$, $0 \leq k \leq 31$, pueden ser precalculados. Si $r(x)$ es un polinomio de grado bajo, o si $f(x)$ es un trinomio, entonces los requerimientos de espacio son mínimos, y también las adiciones involucradas en $x^k r(x)$ son rápidas.

Algorithm 4 Reducción modular (un bit a la vez).

Entrada: Un polinomio binario $c(x)$ de grado no más de $2m-2$.
Salida: $c(x)$ mód $f(x)$.

1. Precálculo. Calcular $u_k(x) = x^k r(x)$, $0 \leq k \leq 31$.
2. Para i de $2m-2$ hasta m hacer
   a) Si $c_i = 1$ entonces
      1) Teniendo $j = \lfloor (i-m)/32 \rfloor$ y $k = (i-m) - 32j$.
      2) Añadir $u_k(x)$ a $C\{j\}$.
3. Regresar$((C[t-1], \ldots, C[1], C[0]))$.

5.1.3. El cuadrado de un polinomio (squaring).

La elevación al cuadrado de un polinomio binario, es un método más sencillo que la multiplicación de dos polinomios binarios . Es una operación lineal en F_{2^m}; en donde, si $a(x) = \sum_{i=0}^{m-1} a_i x^i$, la elevación al cuadrado resultante (squaring) es $a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i}$. La representación binaria de $a(x)^2$ es obtenida insertando un 0 entre bits consecutivos de la representación binaria $a(x)$. Para facilitar este proceso, una tabla del tamaño de 512 bytes puede ser precalculada convirtiendo polinomios de 8-bits, dentro de su contraparte expandida de 16-bits [SOOS95].

5.1.4. Inversión.

El siguiente algoritmo, calcula la inversión de un elemento diferente de cero del campo F_{2^m} . usando el algoritmo extendido de Euclides para polinomios.

Algorithm 5 El cuadrado de un polinomio (squaring).

Entrada: $a \in F_{2^m}$.
Salida: a^2 mód $f(x)$.

1. Precálculo. Por cada byte $v = (v_7, \ldots, v_1, v_0)$, calcular el valor de 16-bits $T(v) = (0, v_7, \ldots, 0, v_1, 0, v_0)$.
2. Para i de 0 hasta $t-1$ hacer
   a) Tenemos $A[i] = (u_3, u_2, u_1, u_0)$ donde cada u_j es un byte.
   b) $C[2i] \leftarrow (T(u_1), T(u_0))$, $C[2i+1] \leftarrow (T(u_3), T(u_2))$.
3. Calcular $b(x) = c(x)$ mód $f(x)$.
4. Regresar(b).

El algoritmo mantiene las invariantes $ba + df = u$ y $ca + cf = v$ para el mismo d y c las cuales no son explícitamente calculadas. Con cada interacción, si $\deg(u) \geq \deg(v)$, entonces una división parcial de u por v es desarrollada para sustraer $x^j v$ de u, donde $j = \deg(u) - \deg(v)$. De esta manera, el grado de u es decrementado como mínimo a 1, y sobre promedio por 2. La sustracción $x^j c$ de b preserva las invariantes. El algoritmo termina cuando $\deg(u) = 0$, en cualquier caso $u = 1$ y $ba + df = 1$; por lo tanto $b = a^{-1}$ mód $f(x)$.

5.2. Representación de puntos en curvas elípticas.

Mencionamos al inicio de este capitulo, que existen dos formas de representar la ecuación de Weierstrass para una curva elíptica en función de la coordenadas en un plano, ó en el espacio proyectivo.

Coordenadas en un plano (afinadas). Para la ecuación $y^2 + xy = x^3 + a_2x^2 + a_6$, en donde $a_2, a_6 \in \{0, 1\}$, tenemos a $P_1 = (x_1, y_1)$ y a $P_2 = (x_2, y_2)$ como dos puntos sobre E con $P_1 \neq -P_2$. Entonces las coordenadas del punto

Algorithm 6 Algoritmo extendido de Euclides para inversiones en F_{2^m}.
Entrada: $a \in F_{2^m}$, $a \neq 0$.
Salida: a^{-1} mód $f(x)$.

1. $b \leftarrow 1$, $c \leftarrow 0$, $u \leftarrow a$, $v \leftarrow f$.
2. Mientras que $\deg(u) \neq 0$ hacer
   a) $j = \deg(u) - \deg(v)$.
   b) Si $j < 0$ entonces: $u \leftrightarrow v$, $b \leftrightarrow c$, $j \leftarrow -j$.
   c) $u \leftarrow u + x^j v$, $b \leftarrow b + x^j c$.
3. Regresar(b).

$P_3 = P_1 + P_2 = (x_3, x_3)$ que es calculado bajo las leyes de adición de curvas elípticas sobre el campo F_{2^m}[1].

Cuando $P_1 \neq P_2$(adición general) y $P_1 = P_2$(duplicación de puntos), las fórmulas para calcular P_3 requiere de una inversión y dos multiplicaciones de campo.

Coordenadas proyectivas: Cuando se dan situaciones donde la inversión en F_{2^m} es costosa para la multiplicación, puede ser ventajoso representar puntos usando coordenadas proyectivas. Las coordenadas proyectivas estándar, con el punto proyectivo $(x : y : z)$, $z \neq 0$ corresponden a los puntos *afinados* $(x/z, y/z)$. La ecuación proyectiva de la curva elíptica es $y^2z + xyz = x^3 + a_2x^2z + a_6z^3$. En coordenadas proyectivas Jacobianas [CC87], el punto proyectivo $(x : y : z)$, $z \neq 0$, corresponde a los puntos afinados $(x/z^2, y/z^3)$ y la ecuación proyectiva de la curva es $y^2z + xyz = x^3 + a_2x^2z^2 + a_6z^6$. Introducimos otra representación de las coordenadas proyectivas [LD00], en donde $(x : y : z)$, $z \neq 0$, corresponden a los puntos afines $(x/z, y/z^2)$, y la ecuación proyectiva de la curva es $y^2z + xyz = x^3z + a_2x^2z^2 + a_6z^4$.

Las fórmulas que no requieran inversiones para adicionar y duplicar puntos

[1] Los procedimientos para la adición de puntos, son vistos en la sección *"Curvas Elípticas sobre el campo F_{2^m}"*.

Sistema de coordenadas.	Adición general	Adición general (coordenadas mezcladas)	Duplicaciones
Coordenadas afines o en el plano	11, 2M	-	11, 2M
Proyectivo Estándar $(x/z, y/z)$	13M	12M	7M
Proyectivo Jacobiano $(x/z^2, y/z^3)$	14M	10M	5M
Proyectivo $(x/z, y/z^2)$	14M	9M	4M

Cuadro 5.1.: Número de operaciones para la adición y duplicación de puntos.

en coordenadas proyectivas pueden derivarse primero convirtiendo los puntos para afinar coordenadas. luego se hacen las operaciones normales de adición de puntos para curvas elípticas sobre el campo F_{2^m} , y finalmente se crean denominadores. Es la adición de dos puntos usando coordenadas mezcladas (un punto dado en coordenadas afinadas y el otro punto en coordenadas proyectivas), los que suelen usarse en métodos de multiplicación de puntos de izquierda a derecha.

Las fórmulas de duplicación para la última ecuación proyectiva son: $2\,(x_1 : y_1 : z_1) = (x_3 : y_3 : z_3)$. donde

$$z_3 = x_1^2 \cdot z_1^2,\ x_3 = x_1^4 + a_6 z_1^4,\ y_3 = b z_1^4 \cdot z_3 + x_3 \cdot \left(a_2 z_3 + y_1^2 + b z_1^4\right)$$

Las fórmulas para la adición en coordenadas mezcladas son: $(x_1 : y_1 : z_1) + (x_2 : y_2 : 1) = (x_3 : y_3 : z_3)$, donde

$$A = y_2 \cdot z_1^2 + y_1,\ B = x_2 \cdot z_1 + x_1,\ C = z_1 \cdot B,\ D = B^2 \cdot \left(C + a_2 z_1^2\right),$$

$$z_3 = C^2,\ E = A \cdot C,\ x_3 = A^2 + D + E,\ F = x_3 + x_2 \cdot z_3,$$

$$G = x_3 + y_2 \cdot z_3,\ y_3 = E \cdot F + z_3 \cdot G.$$

El número de operaciones de campo para la adición de puntos y duplicaciones en los sistemas de coordenadas anteriores se muestran en la siguiente tabla.

5.3. Multiplicación escalar de puntos.

La multiplicación de puntos en curvas elípticas es un caso especial del problema general de potenciación en grupos abelianos. Representa la construcción básica de bloques de criptosistema de una curva elíptica sobre F_q. Se calcula de la forma:

$$Q = [k]\,P = \underbrace{P + P + \ldots + P}_{k\,-\,veces}$$

Donde P es un punto en la curva, y k es un entero arbitrario en el rango $1 \leq k < ord\,(P)$. Para algunos de los protocolos criptográficos P es un grupo compuesto que genera un subgrupo de orden primo grande de $E\,(F_q)$, mientras que para otros criptosistemas, P es un punto arbitrario en tal subgrupo. La fuerza del criptosistema radica en el hecho de que dada la curva, el punto P y $[k]\,P$, es difícil recuperar k. Este es el problema del logaritmo discreto de la curva elíptica (ECDLP).

Esto último es definido de la forma siguiente:

> Sea k un entero positivo, que comienza en el entero 1, y calculando por cada paso la suma de dos resultados previos. ¿Cual es el menor número de pasos requeridos para localizar k?.

Se han propuesto una gama muy variada de algoritmos para la realización de la multiplicación de puntos que involucran una serie de atajos para reducir la complejidad y el costo computacional de este problema.

Definimos el costo computacional de los algoritmos siguientes con las variables I y M, en donde I representa el costo de las inversiones de campo y M la multiplicación de campo respectiva.

5.3.1. Método binario.

El método binario (Algoritmo 1) requiere $l-1$ puntos múltiplos duplicados y $w-1$ puntos aditivos (las operaciones que involucran σ no son contadas). donde l es la longitud y w el peso (número de unos) de la expansión binaria de k . Asumiendo que el promedio $w = \frac{l}{2}$, que típicamente es $l \approx n$, y descuidando $O(1)$ términos, el número promedio de operaciones de campo es de

$1{,}5nI + 3nM$ en una representación afinada (Ecuación (2)), ó $10n$M en una representación proyectiva (Ecuación (1)).

Algorithm 7 Multiplicación de puntos: Método binario.

Entrada: Un punto P, un $l-bit$ entero $K = \sum_{j=0}^{l-1} k_j 2^j$, $k_j \in \{0,1\}$
Salida: $Q = [k]\,P$.

1. $Q \leftarrow \sigma$
2. Para $j = l-1$ a 0 por -1 hacer:
3. $\quad Q \leftarrow [2]\,Q$,
4. $\quad$ Si $k_j = 1$ entonces $Q \leftarrow Q + P$.
5. Regresar Q.

5.3.2. Método *m-ario.*

Es fácilmente verificable que el algoritmo anterior (Algoritmo 2) calcula $[k]\,P$, siguiendo la regla de Horner:

$$[m]\,(\ldots[m]\,([m]\,([k_{l-1}]\,P) + [k_{l-2}]\,P) + \ldots) + [k_0]\,P = [k]\,P$$

El número de duplicaciones en el cuerpo principal del algoritmo $m-ario$ es de $(d-1)\,r$ (la primera interacción no cuenta, ya que comienza con $Q = \sigma$). Donde $d = \lceil \frac{l}{r} \rceil$, donde l es la longitud de la representación binaria de k, el número de duplicaciones en el método $m-ario$ podría ser arriba de $r-1$ menos que los $l-1$ requeridos por el método *binario*. En realidad, para parámetros típicos, la ganancia para los dobleces es un poco modesta, la principal ganancia sobre el método binario es el número de puntos generales de adición.

5.3.3. Calculando la NAF de un entero positivo.

La substracción tiene virtualmente el mismo costo de cálculo como la adición en el grupo de las curvas elípticas. Para las ecuaciones de la curva

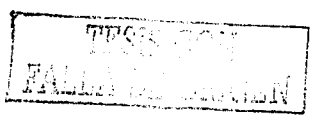
Algorithm 8 Multiplicación de puntos: Método *m-ario*.

Entrada: Un punto P, un entero $k = \sum_{j=0}^{d-1} k_j m^j$, $k_j \in \{0, 1, \ldots, m-1\}$
Salida: $Q = [k]\,P$.
Precomputación:

1. $P_1 \leftarrow P$
2. para $i = 2$ a $m-1$ hacer: $P_i \leftarrow P_{i-1} + P$ (nosotros tenemos $P_i = [i]\,P$).
3. $Q \leftarrow \sigma$

Principal:

1. Para $j = d-1$ a 0 por -1 hacer:
2. $\quad Q \leftarrow [m]\,Q$. (esto requiere r duplicaciones)
3. $\quad Q \leftarrow Q + P_{k_j}$.
4. Regresar Q.



canónicas de interés, el grupo negativo de un punto (x, y) es $(x, x + y)$ en la característica dos. y de $(x, -y)$ en características impares. Esto conduce a los métodos de multiplicación de puntos basados sobre cadenas de adición y substracción. en la cual se puede reducir el número de operaciones en la curva.

Si $P = (x, y) \in E(F_{2^m})$, entonces $-P = (x, x + y)$. Está substracción de puntos es igual de eficiente como la adición misma. Se implementa usando una *representación del dígito del signo* $k = \sum_{i=0}^{l-1} k_i 2^i$, donde $k_i \in \{0, \pm 1\}$. Una útil representación del dígito del signo es de la forma no adyacente (NAF - Non-Adjacent Form), que tiene la propiedad de que dos coeficientes no consecutivos k_i son diferentes de cero. Cada entero positivo k tiene un único NAF, que se denota como NAF(k). Además, NAF(k) tiene pocos coeficientes diferentes de cero de cualquier representación del dígito del signo de k [MO97, So00].Pueden encontrarse otras variantes del algoritmo en [AW1993, JM89, MG71, Reit60].

Algorithm 9 Calculando la NAF de un entero positivo.
Entrada: Un entero positivo k.
Salida: NAF(k).

1. $i \leftarrow 0$.
2. Mientras que $k \geq 1$ hacer
   a) Si k es impar entonces: $k_i \leftarrow 2 - (k \bmod 4)$, $k \leftarrow k - k_i$;
   b) De lo contrario: $k_i \leftarrow 0$.
   c) $k \leftarrow k/2$, $i \leftarrow i + 1$.
3. Regresar$((k_{i-1}, k_{i-2}, \ldots, k_1, k_0))$.

5.3.4. Método NAF-binario.

La adaptación del método binario para la multiplicación de puntos a NAFs es directo. Asumiendo un peso promedio para NAF de $n/3$, el costo computacional es de $\frac{1}{3}n(2M + I)$ para coordenadas en dos planos (coordenadas afinadas) [BSS1999].

Algorithm 10 Multiplicación de puntos: Método NAF-binario.

Entrada: $\mathrm{NAF}(k) = \sum_{i=0}^{l-1} k_i 2^i$, $P \in E(F_{2^m})$.
Salida: kP.

1. $Q \leftarrow \sigma$.
2. Para i de $l-1$ bajando a 0 hacer
   a) $Q \leftarrow 2Q$.
   b) Si $k_i = 1$ entonces $Q \leftarrow Q + P$.
   c) Si $k_i = -1$ entonces $Q \leftarrow Q - P$.
3. Regresar(Q).

5.3.5. Método NAF-Window.

Si tenemos memoria extra disponible, el método anterior puede optimizarse usando un submétodo de ventana (*window*)[H00], que puede procesar w dígitos de k al mismo tiempo. Algunas variantes de los algoritmos de ventana, que hacen un uso del calculo NAF previo, pueden consultarse en [KT93, MTH97]. El siguiente algoritmo [So00] hace uso del submétodo de ventana, con cálculo NAF incluido en el proceso.

Una *ventana NAF de ancho (w)* de un entero k es una expresión $k = \sum_{i=0}^{l-1} k_i 2^i$, donde cada coeficiente diferente de cero k_i es impar, $|k_i| < 2^{w-1}$, y al menos uno de cualquiera de los coeficientes consecutivos de w es diferente de cero. Cada entero positivo tiene una única *ventana NAF de ancho (w)*, que se define como $\mathrm{NAF}_w(k)$. Se sabe que la longitud de $\mathrm{NAF}_w(k)$ es más grande que la binaria representación de k. También la densidad promedio de coeficientes diferentes de cero entre todas las *ventanas NAF de ancho (w)*, de longitud l es aproximadamente $1/(w+1)$. El tiempo de ejecución de este método para coordenadas afinadas es aproximadamente de $(1D + (2^{w-2} - 1)A) + (m/(w+1)A + mD)$.

Algorithm 11 Multiplicación de puntos: Método NAF-Window.

Entrada: *Ventana (window) NAF de ancho (w)*, $\mathrm{NAF}_w(k) = \sum_{i=0}^{l-1} k_i 2^i$, $P \in (F_{2^m})$.
Salida: kP.

1. Calcular $P_i = iP$. para $i \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$.
2. $Q \leftarrow \sigma$.
3. Para i de $l - 1$ hasta 0 hacer
   a) $Q \leftarrow 2Q$.
   b) Si $k_i \neq 0$ entonces:
      1) Si $k_i > 0$ entonces $Q \leftarrow Q + P_{k_i}$;
      2) De lo contrario $Q \leftarrow Q - P_{k_i}$.
4. Regresar(Q).

5.3.6. Método Window Fixed-base.

Si el punto P es fijo y algunos espacios están disponibles (locaciones en la memoria). entonces la multiplicación de puntos puede ser acelerada precalculando algunos datos que dependen sólo de P. Por ejemplo, si los puntos $2P. 2^2P. \ldots. 2^{t-1}P$ son precalculados, entonces el método binario tiene un tiempo estimado de ejecución de $(m/2)\,A$. Un refinamiento de está idea fue propuesto en [BGMW93]. Tenemos que $(k_{d-1}, \ldots, k_1, k_0)_{2^w}$ son la $2^w - aria$ representación de k. donde $d = \lceil t/w \rceil$, y considerando que $Q_j = \sum_{i:k_i=j} 2^{wi}P$. Entonces

$$kP = \sum_{i=0}^{d-1} k_i \left(2^{wi}P\right) = \sum_{j=1}^{2^w-1} \left(j \sum_{i:k_i=j} 2^{wi}P \right) = \sum_{j=1}^{2^w-1} jQ_j$$

$$= Q_{2^w-1} + (Q_{2^w-1} + Q_{2^w-2}) + \cdots + (Q_{2^w-1} + Q_{2^w-2} + \cdots + Q_1).$$

El siguiente algoritmo está basado en esta observación. Se espera que el tiempo de ejecución sea aproximadamente de $((d\,(2^w - 1)\,/2^w - 1) + (2^w - 2))$.

Algorithm 12 Multiplicación de puntos: Método Window Fixed-base.
Entrada: *Ventana (window) NAF de ancho (w)*, $d = \lceil t/w \rceil$, $k = (k_{d-1}, \ldots, k_1, k_0)\, 2^w$, $P \in E(F_{2^m})$.
Salida: kP.

1. Precálculo. Calcular $P_i = 2^{wi}P, 0 \leq i \leq d-1$.
2. $A \leftarrow \sigma, B \leftarrow \sigma$.
3. Para j de $2^w - 1$ hasta 1 hacer:
   a) Para cada i para los cuales $k_i = j$ hacer: $B \leftarrow B + P_i$. {Adicionar Q_j a B}.
   b) $A \leftarrow A + B$.
4. Regresar(A).

5.3.7. Método Comb Fixed-base.

El método propuesto en [LL94], llamado el método *comb* ("panal"), la binaria representación de k es escrita en w columnas, y las columnas del rectángulo resultante son procesadas una columna a la vez. Definimos $[a_{w-1}, \ldots, a_2, a_1, a_0]\, P = a_{w-1}2^{(w-1)d}P + \cdots + a_2 2^{2d}P + a_1 2^d P + a_0 P$, donde $d = \lceil t/w \rceil$ y $a_i \in Z_2$. El tiempo de ejecución esperado de este algoritmo es aproximadamente de $((d-1)(2^w - 1)/2^w)\, A + (d-1)\, D$.

5.4. Determinar un punto aleatorio en $E(F_q)$.

Por el teorema de Hasse, el número de puntos sobre la curva para valores grandes de q es en un limitado rango de $4\sqrt{q}$ con respecto al valor de $q+1$. Para ver porque podría ser así, note que alrededor de la mitad de todas las posibles $x-coordenadas$ de q en F_q van a obtener una solución y. Todos pero al menos tres de estas van a tener dos correspondientes $y-coordenadas$, la excepción sería los puntos de orden dos (aquellos puntos con $y-coordenadas$ igual a cero en la forma corta de la fórmula de Weierstrass para la curva). A

Algorithm 13 Multiplicación de puntos: Método Comb Fixed-base.

Entrada: *Ventana (window) NAF de ancho (w),* $d = \lceil t/w \rceil$, $k = (k_{t-1}, \ldots, k_1, k_0)_2$, $P \in E(F_{2^m})$.
Salida: kP.

1. *Precálculo.* Calcular $[a_{w-1}, \ldots, a_1, a_0]\, P \forall\, (a_{w-1}, \ldots, a_1, a_0) \in Z_2^w$.
2. Para rellenar k sobre la izquierda con ceros si es necesario, escribir $k = K^{w-1} \parallel \cdots \parallel K^1 \parallel K^0$, donde K^j es una cadena de bits de longitud d. Tenemos a K_i^j denotando el $i-avo$ bit de K^j.
3. $Q \leftarrow \sigma$.
4. Para i de $d-1$ hasta 0 hacer
   a) $Q \leftarrow 2Q$.
   b) $Q \leftarrow Q + \left[K_i^{w-1}, \ldots, K_i^1, K_i^0\right] P$.
5. Regresar(Q).

este número esperado q de puntos racionales nosotros agregamos el punto al infinito σ, haciendo un total de $q+1$ puntos racionales sobre la curva en F_q.

Con esta observación, propondremos escoger los elementos de $E(F_q)$ con una distribución casi uniforme (Algoritmo 3):

Algorithm 14 Determinar un punto aleatorio en $E(F_q)$.

Entrada: Una curva elíptica E.

Salida: Un punto *aleatorio* $P \in E(F_q)$.

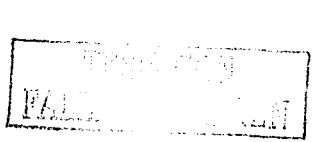
1. Hacer:
2. Escoger en forma aleatoria $x \in F_q$
3. Sustituir x en la ecuación (2).
4. Intentar resolver el resultado de la ecuación cuadrática en y.
5. Si la solución y se encuentra, lanzar entonces una moneda y decidir cual y se escoge y cambiar $P = (x, y)$.
6. Hasta que un punto P es encontrado.
7. Regresar P.

5.5. El Orden de una Curva Elíptica.

Por el teorema de Hasse, es conocido el número de puntos racionales sobre una curva elíptica sobre F_{2^m}, por medio de la siguiente ecuación:

Teorema de Hasse: Sea E la curva elíptica sobre el campo finito F_q, con $q = p^n$ una potencia del primo p. El número de puntos en E será:$\#E(F_q) = q - 1 - t$En el caso de las curvas supersingulares, una de las siguientes condiciones se cumple:

1. n par: $t = \pm 2\sqrt{q}$



2. n par y $p \neq 1$ (mód3): $t = \pm\sqrt{q}$
3. n non y $p = 2$ ó 3: $t = \pm\sqrt{pq}$
4. Cualquier n impar y $p \neq 1$ (mód4): $t = 0$

En el caso de las curvas no-supersingulares, t cumple con las siguientes condiciones:

- $|t| \leq 2\sqrt{q}$
- $\gcd(t, p) = 1$

Llamamos a t como el rastro de *Frobenius* en q.

El problema de determinar el orden del grupo de puntos racionales sobre una curva elíptica en un campo finito es de vital importancia en criptografía. Nosotros requerimos que la curva sea no-singular y el orden del grupo sea divisible por un factor primo grande; en la actualidad podría ese primo factor ser más de cien bits de tamaño (160 bits es casi siempre el mínimo requerido). Es por ello que el problema se vuelve dificultoso, ya que se requiere de soluciones innovadoras ante los retos matemáticos que sean computacionalmente más efectivos.

5.6. El Orden de un Punto.

Un punto aleatorio P en E puede ser calculado para escoger un elemento $x_1 \in F_{2^m}$, e intentar resolver la *ecuación no-singular* de la curva. Por medio del teorema de Hasse, se observa que la probabilidad de que x_1 sea la coordenada x de un punto en E es de aproximadamente 1/2 [MV93]. El orden de un punto puede ser calculado en un tiempo polinomial si la factorizacion de $\#E$ es conocida para el siguiente algoritmo [MQV95]:

5.7. Protocolos.

Los protocolos de intercambio de clave, tienen como propósito, compartir un secreto compartido entre una entidad principal, y una o varias entidades

Algorithm 15 El Orden de un punto.

Precomputación: Una curva elíptica E definido sobre F_{2^m}, donde la factorizacion prima de $\#E(F_{2^m})$ es

$$\#E(F_{2^m}) = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k},\ e_i \geq 1.$$

Entrada: Un punto $P = (x, y)$ sobre E.
Salida: n

1. $n \leftarrow \#E(F_{2^m})$.
2. Para i de 1 hasta k hacer
   a) $n \leftarrow n/p_i^{e_i}$
   b) $P_1 \leftarrow nP$
   c) Mientras que $P_1 \neq \sigma$, calcular $P_1 \leftarrow p_i P_1$ y cambiar $n \leftarrow np_i$.
3. Salida n.

involucradas. Este es el módulo principal de nuestro criptosistema de curva elíptica. Una característica importante en los criptosistemas de clave pública, es la longitud de nuestro mensaje, determinada en número de bits. Sabemos que el tratamiento de bits en un criptosistema de clave pública es una tarea compleja que involucra la totalidad de las operaciones en el criptosistema. Se recomienda entonces el uso de mensajes cortos que contengan sólo la información necesaria. El concepto de *corto* es variable y depende en gran medida de nuestras necesidades de seguridad y protección de datos.

Es por eso. que los protocolos aquí expuestos son ideales para intercambio de claves, cuando necesariamente, usamos un criptosistema de cifrado simétrico alterno para la protección interna de nuestros datos.

5.7.1. Protocolo de Diffie-Hellman.

También conocido como el protocolo de cambio de clave de Diffie-Hellman, este protocolo fue el primer esquema original de un criptosistema de clave pública, [DH76]. Su adaptación a criptosistemas de curva elíptica no es difícil, se elige la curva elíptica que satisface la ecuación: $y^2 + xy = x^3 + a_2x^2 + a_6$. En donde para $a_6 \neq 0$, y a_2 puede tener el valor de 0 o 1.

Para representar el algoritmo, inventamos a dos personas con necesidades de comunicación cifrada entre ellas: Alice y Bob (Algoritmo 16).

5.7.2. ElGamal para curvas elípticas.

En 1985, ElGamal introduce un criptosistema, basado en el problema del logaritmo discreto, [EG85, MO97]. Hoy en día, a este criptosistema se le llama *Criptosistema ElGamal* (no confundir con *Esquema de Firma ElGamal).*

Este protocolo, no se menciona en el IEEE P1363. Es un protocolo muy útil para generar aleatoriamente curvas y puntos, ya que no requiere conocer el orden de la curva, los factores de este número, o el orden del punto base. Otra ventaja, es que no esta patentado.

La versión de ElGamal para curvas elípticas, trabaja con el problema del logaritmo discreto de la curva elíptica, requiere escoger el tamaño del campo finito, la base matemática en la que se representa, y la curva pública E, con

Algorithm 16 Protocolo de Diffie-Hellman.

Descripción: Se ha elegido previamente entre Alice y Bob, un campo finito F_{2^m}, una curva elíptica E, tipo de base matemático empleado (bases normales o polinomiales), y un punto base B.

1. Alice y Bob calculan por separado sus claves privadas respectivas: k_A y k_B.
2. Bob calcula $P_B = k_B B$ sobre la curva elíptica elegida, y se lo envía a Alice.
3. Alice calcula $P_A = k_A B$ en la misma curva, y se lo envía a Bob.
4. Entonces ellos calculan el secreto compartido $P_s = k_A(k_B B) = k_B(k_A B)$.

la que se satisface la ecuación

$$y^2 + xy = x^3 + a_2 x^2 + a_6$$

en donde para $a_6 \neq 0$, se puede escoger para a_2 el valor 0 o 1. Cuando $a_2 = 0$, se incrementa la velocidad de nuestro criptosistema en un 10 por ciento.

Los únicos puntos que un atacante ve son P_h, P_A, P_B y P_r. La ventaja de este protocolo es que las claves públicas, permanecen públicas, y no es necesario cambiarlos. Cada vez que un nuevo dato es intercambiado, un nuevo valor aleatorio r es escogido. Ninguna de las entidades involucradas necesita recordar r, y si el tamaño del campo finito es lo suficientemente grande, será muy dificultoso descubrir los números secretos k_B o k_A(Algoritmos 17 y 18).

5.7.3. El esquema de acuerdo de claves Menezes-Qu-Vanstone.

Este esquema de acuerdo de claves es más avanzado que el esquema de Diffie-Hellman y ElGamal, pero incluye el esquema de Diffie-Hellman como un subconjunto. La idea de este esquema, es prevenir el ataque del *hombre*

Algorithm 17 Protocolo ElGamal: Generación de claves.

Descripción: Cada entidad crea una clave pública y su correspondiente clave privada. Cada entidad A deberá hacer lo siguiente:

1. Generar un punto base B en la curva elíptica pública.
2. Cada entidad selecciona un entero aleatorio k_A. $1 \leq k_A \leq n-1$. Esta es la clave privada.
3. Cada entidad calcula su clave pública

$$P_A = k_A B$$

Algorithm 18 Protocolo ElGamal: Encriptación - Desencriptación.

Descripción: Llamamos a la entidad que encripta y envía el mensaje como Alice. y a su clave privada k_A; a la entidad que recibe y descifra el mensaje como Bob. y su correspondiente llave privada como k_B. Alice encripta un mensaje incrustándolo como un punto P_m en la curva E pública para Bob.

1. Encriptación. Alice debe hacer lo siguiente:
   a) Alice escoge un entero aleatorio r, y computa dos puntos: $P_r = rB$ y $P_h = P_m + rP_B$.
   b) Alice envía los dos puntos P_r y P_h a Bob.
2. Desencriptación. Para recuperar el mensaje P_m, Bob debe hacer lo siguiente:
   a) Bob calcula el punto $P_s = k_B P_r$.
   b) y substrae P_s de P_h para obtener: $P_m = P_h - P_s$. Para entender como trabaja desglosamos está ecuación:
      1) $rP_B = r(k_B B)$.
      2) $P_h = P_m + r(k_B B)$.
      3) $P_m = P_m + r(k_B B) - k_B(rB)$.

en la mitad (man-in-the-middle) y desarrollar autentificación de las claves obtenidas. Este protocolo es mencionado en el estándar P1363 .

La comunicación se da en dos lados, y por cada lado tenemos dos claves. esto hace cuatro claves para considerar. Los datos claves del lado 2 serán identificados por el apóstrofe o signo primo ('); los datos claves del lado 1, no tendrán tal apóstrofe. Por ejemplo, cada lado genera una clave aleatoria efímera: R sería en el lado 1 y R' sobre el lado 2.

Todos los cálculos ocurren sobre la misma curva elíptica E, que satisface la ecuación $y^2 + xy = x^3 + a_2x^2 + a_6$. El punto base es llamado P, y nuestra clave pública (o punto público) es:

$$Q = dP$$

donde d es nuestra clave privada. El lado 2, tiene la clave pública Q', y nosotros desconocemos la clave privada del lado 2, d'.

Este esquema usa los componentes x de los puntos de la clave pública para los cálculos de la matemática modular. Esto vincula los datos efímeros (preferentemente aleatorios) a los datos de la clave pública de una forma que solo el dueño de la clave puede calcular.

Estos son los parámetros para este protocolo:

- E -los coeficientes de la curva elíptica
- P -el punto base
- n -el orden de la curva
- Q y Q' -las claves públicas permanentes del lado 1, y del lado 2, respectivamente.
- R y R' -las claves públicas efímeras del lado 1, y del lado 2, respectivamente.

Para los parámetros de la curva, nosotros especificamos que el par de datos (x, y), las cuales componen un punto, son distribuidos y simbolizados de la siguiente forma:

- $Q' = (a', b')$

- $R' = (x', y')$
- $Q = (a, b)$
- $R = (x, y) = kP$.

La salida del esquema es el secreto deseado t, pero ningún lado observara los datos de su lado contrario, ni sabrá cuales son o como son.

Si un atacante encuentra t de una transmisión particular, este no le será de ayuda.

Primera versión del protocolo MQV.

El estándar especifica el primer calculo de la siguiente manera:

$$s = k + xad \mod n. \qquad (a)$$

El valor s es un término privado y no deberá dejarse fuera del proceso de generación de claves.

El siguiente paso es calcular el punto U de los datos del lado 2, usando la fórmula:

$$U = R' + x'a'Q'. \qquad (b)$$

El secreto compartido es entonces el componente x del punto:

$$W = sU. \qquad (c)$$

En este estado, ambos lados tendrán el mismo valor W y ninguno estará habilitado para calcularlo, porque ningún lado conoce las claves privadas del contrario. Veamos entonces como esto trabaja:

Primero, multiplicamos la ecuación (a) por el punto base P:

$$sP = kP + xa\,(dP)\,. \qquad (d)$$

Nosotros podemos sustituir $Q = dP$ y $R = (x, y) = kP$ dentro de la ecuación anterior:

$$sP = R + xaQ. \qquad (e)$$

Si nosotros reemplazamos cada término sobre el lado derecho de la ecuación anterior, con su similar del lado 2, nosotros obtendremos el punto U', el cual el lado 2 calcula de nuestros datos. Pero U no es igual a U'. Para ver como ambos lados derivan la misma clave, expandimos la ecuación (c) usando las ecuaciones (a) y (b):

$$W = (k + xad)\,(R' + x'a'Q')\,. \qquad (f)$$

Nosotros sabemos que R' y Q' son derivados del punto base P, así que nosotros expandimos la ecuación anterior como sigue:

$$W = (k + xad)\,(k' + x'a'd')\,P. \qquad (g)$$

Ahora es claro que ambos lados tienen el mismo valor de W. En el núcleo de este protocolo vemos que se conforma de un simple protocolo de intercambio de claves de Diffie-Hellman. La ventaja radica en que se protegen nuestras comunicaciones de un ataque y tenemos verificación incluida. Si un atacante no conoce d o d', el atacante no podrá averiguar el secreto compartido. Además, el secreto compartido es diferente cada vez que dos lados se comunican, porque valores aleatorios k y k' han sido introducidos en cada lado. Incluso, si un atacante encuentra la clave privada, el o ella tendría que resolver el problema del logaritmo discreto para la curva elíptica para encontrar k y k' por cada mensaje.

Segunda versión del protocolo MQV.

El estándar en su segunda versión define un término que es la mitad del logaritmo base 2 del orden del punto base. Si r es el factor primo grande en el orden de la curva, entonces la ecuación aparece como:

$$h = \left\lceil \frac{(\log_2 r)}{2} \right\rceil .$$

El estándar especifica los pasos siguientes:

1. Crear un entero t, el cual es la mitad inferior de los componentes x del punto efímero público en el lado 1.

2. Crear un entero t', el cual es la mitad inferior de los componentes x del punto efímero público en el lado 2.

3. Calcular un entero e =(t · la clave secreta del lado 1 + la clave efímera del lado 1).

4. Calcular un punto en la curva elíptica $P = e \cdot$ (el punto efímero del lado2 + $t' \cdot$ el punto de clave pública del lado 2).

5. La salida de los componentes x de P como el secreto compartido.

Procedemos entonces a desglosar matemáticamente los pasos involucrados para este protocolo, usando las mismas variables de la versión anterior.

El primer paso es:

$$t = x \text{ mód } 2^h$$

y

$$t' = x' \text{ mód } 2^h.$$

Juntos t y t' tiene el bit h de acuerdo al estándar, esto añade 2^h a cada valor y fuerza a un múltiplo diferente de cero.

Las dos ecuaciones anteriores son seguidas por un cálculo entero. La clave privada es d, y la clave privada efímera es k:

$$e = td + k \text{ mód } r \qquad (a)$$

y un cálculo de la curva elíptica usando el punto efímero R' y la clave pública Q' del lado contrario:

$$W = e\,(R' + t'Q')\,. \qquad (b)$$

El secreto compartido es el componente x del punto W. Si nosotros expandimos los puntos en la ecuación anterior por múltiplos del punto base P, nosotros encontramos que:

$$R' + t'Q' = (k' + t'd')\,P. \qquad (c)$$

Colocando la ecuación (c) de regreso dentro de la ecuación (b), junto con la ecuación (a), observamos que ambos lados calculan el mismo punto W como:

$$W = (k + td)\,(k' + t'd')\,P. \qquad (d)$$

6. Elección de la Curva Elíptica.

La seguridad de un criptosistema de curva elíptica depende de la elección efectiva de la curva elíptica en la que el criptosistema se implementa. Son muy variados los parámetros que influyen en la elección de la curva elíptica ideal para nuestro criptosistema. Primordialmente, se debe considerar el nivel de seguridad que pretendemos alcanzar. La elección del grado m de nuestro campo finito (campos de característica dos), debe guardar cierta proporción con la longitud (en bits) de nuestro mensaje secreto.

Lo ideal para cualquier criptosistema, es que el número de bits m tenga el tamaño suficiente para guardar nuestro secreto, pero sin limitar drásticamente el tiempo de ejecución del criptosistema.

El primer paso a seguir, será necesariamente detectar aquellas curvas catalogadas como no idóneas para propósitos criptográficos. Un ejemplo de ello, es evitar u omitir las *curvas supersingulares*, ya que estas tienden a reducir el *problema del logaritmo discreto generalizado* sobre el campo F_{2^m} a el problema del *logaritmo discreto* sobre una extensión de campo K de grado menor (para mayores referencias de las curvas supersingulares, consulte [BSS1999. MOV'93]).

Las curvas ideales para propósitos criptográficos, son aquellas que cubren con las siguientes propiedades:

- La curva debe tener un orden grande ($\#E(F_{2^m})$).
- El orden es divisible por un factor primo grande.
- El factor primo grande no debe satisfacer la propiedad de la divisibilidad: $P \in F_{2^m} \mid 2^{mi} - 1$, para cualquier i de valor menor.

- La cardinalidad de la curva elíptica con el campo primo, donde $\#E(F_{2^m}) = 2^m$ debe evitarse, a este tipo de curvas se le denomina curvas anómalas[Sem98, Smart99].
- Las curvas implementadas en campos binarios compuestos, donde $m = r \cdot s$, podrían tener un grado de complejidad menor del problema del logaritmo discreto sobre porciones significativas de curvas elípticas definidas sobre campos compuestos pequeños r [GS99, GHS00].

Cuatro técnicas son consideradas ampliamente para determinar la factibilidad de que la curva elíptica sea la ideal para propósitos criptográficos:

1. Generar las curvas aleatoriamente y calcular sus ordenes, hasta que una apropiada es encontrada.
2. Generar las curvas con ordenes dadas usando la teoría de la multiplicación compleja (CM).
3. Usar curvas de *tipo Koblitz.*
4. Usar curvas conocidas y probadas por alguna entidad certificadora de estándares.

6.1. Curvas aleatorias sobre el campo F_{2^m}.

Nosotros sabemos por la teoría de Hasse que el número de puntos en una curva elíptica sobre un campo finito F_{2^m}, podría ser cualquiera entre 2^m y $2^{m/2}$. Para mayor seguridad, nosotros buscamos el numero de puntos de la curva elíptica de 2^m.

La distribución del orden de una curva está relacionada con la estructura del campo donde se implementa la curva. Para campos sobre F_{2^m}, donde m es primo, las ordenes de las curvas tienden a tener un factor primo grande; cuando m no es primo, las ordenes de las curvas tienden a tener muchos factores primos más pequeños, se dice entonces que la curva es *smooth* (suave).

Así, existe una gran probabilidad de que el orden de la curva va a contener un factor primo grande si m es primo. Ciertamente, se recalca todo el tiempo la importancia de calcular el orden de una curva.

Algorithm 19 Generando curvas elípticas: Método aleatorio.

Entrada: Un campo finito grande F_q, donde $q = 2^m$. Un entero positivo pequeño s.

Salida: Una curva elíptica E sobre F_q tal que $E(F_q) = S \cdot r$, $S \leq s$, r es número primo.

1. Trazar E aleatoriamente, con coeficientes en F_q.
2. Determinar el orden $\#E(F_q)$.
3. Checar que las curvas no caigan en la condición de curva anómala, ni en condición MOV (curvas supersingulares). Si cualquiera de estas condiciones se cumple, entonces regresar al paso 1.
4. Intentar factorizar $\#E(F_q)$ en un tiempo *razonable*. Si el intento falla, ir al paso 1.
5. Si $\#E(F_q) = S \cdot r$, $S \leq s$, r es número primo, regresar E. Si no ir al paso 1.

6. *Elección de la Curva Elíptica.*

Para estimar la probabilidad de encontrar una curva ideal en una interacción del algoritmo anterior, debemos cuantificar primero el término "grande", usado cuando nos referimos al número primo r (o "pequeño" cuando nos referimos a s). Asumiendo que el ECDLP[1] es realmente de complejidad exponencial, $\log_2 r$ es una buena medida del *número de bits de seguridad* en el criptosistema, para esto una búsqueda exponencial en está medida es necesaria para romper el criptosistema. Por otro lado, el "tamaño de la llave" es n, el tamaño de un elemento del campo, y la complejidad de las operaciones requeridas para implementar el criptosistema crece polinomialmente con n. Por lo tanto, para obtener el criptosistema más fuerte posible, nosotros podríamos hacer $\log_2 r$ tan grande como sea posible. Recalcamos esto para el teorema de Hasse, $\log_2 r$ está limitado por encima de n aproximadamente. Definimos la pérdida del criptosistema como

$$\epsilon = 1 - \frac{\log_2 r}{n}.$$

Para un entero s, H_s denota el conjunto de múltiplos de s en el intervalo $\left[q+1-2\sqrt{q},\ q+1+2\sqrt{q}\right]$, $q=2^n$ y $H_s/s = \{i : i \in H_s\}$.

Para estimar la probabilidad de trazar aleatoriamente una curva [Kob90] con perdida ϵ, dos asumciones son hechas: (i) el orden $\#E(F_q)$ es distribuida uniformemente en H_s, y (ii) para s pequeña, la distribución de primos entre enteros en H_s/s es similar a la distribución de primos entre enteros arbitrarios de el mismo orden de magnitud como q/s. Para el Teorema de Números Primos, la densidad de primos en H_s/s es hasta ahora aproximadamente de $1/\log(q/s)$.

Sea $S = \lfloor 2^{n\epsilon-1} \rfloor$. Entonces, bajo las anteriores asumciones, y usando las bien conocidas propiedades de las series harmónicas, la probabilidad de una curva de perdida ϵ es estimada por

$$\sum_{j=1}^{S} \frac{1}{j \log(q/2j)} \geq \frac{1}{\log q} \sum_{j=1}^{S} \frac{1}{j} = \frac{1}{\log q}\left(\log S + O(1)\right) = \epsilon + o(1).$$

El próximo método que aquí se explica, utiliza una búsqueda de candidatos posibles a ser el orden de una curva, y que se implementa en algunos algoritmos encargados de contar puntos; tal es el caso del algoritmo de Schoof, que

[1] "Problema del Logaritmo Discreto para la Curva Elíptica". Este término es equivalente del "Problema del Logaritmo Discreto Generalizado".

utilizaremos posteriormente como un algoritmo que nos verifica si la curva elegida es ideal para nuestro criptosistema implementado.

6.1.1. Verificando el Orden del grupo.

Tenemos una curva elíptica E definido sobre F_q, con $q = p^n$; deseamos determinar entonces cuando un entero m producido por un algoritmo contador de puntos, es el orden de $E(F_q)$. El primer test obvio es para asegurarnos que m está dentro del intervalo de Hasse

$$q + 1 - 2\sqrt{q} \leq m \leq q + 1 + 2\sqrt{q}$$

Una ves establecido, un punto p en $E(F_q)$ es seleccionado aleatoriamente, y la condición

$$m[P] = 0$$

es verificada. Ciertamente, si la condición mencionada no se da, m no es el orden del grupo. Si la condición se da, tiene una probabilidad alta de que m sea el orden del grupo; la certeza de que m sea el orden del grupo depende de la factorizacion de m. La probabilidad puede incrementarse trazando y checando puntos aleatorios adicionales. Una posible aproximación seria guardar todos los posibles candidatos de m que pasan la prueba del punto aleatorio mencionado anteriormente, y al final del algoritmo trazar y checar puntos aleatorios hasta que sólo un candidato sobreviva. Está aproximación ocurre si el algoritmo contador de puntos, garantiza producir el verdadero orden del grupo como uno de los candidatos.

Para las ordenes de grupo de interés en criptografía, nosotros sólo nos interesamos en las ordenes de grupo m, de la forma:

$$m = s \cdot r$$

donde s es un pequeño entero positivo, y r es un primo. Para valores de s usados en la práctica, está puede hacerse por pruebas de división y primalidad. Si m no es de la forma deseada, entonces se descarta, aunque podría ser el verdadero orden del grupo. Si los candidatos no sobreviven al final del algoritmo contador de puntos, la curva es demasiado inapropiada para aplicaciones criptográficas. y se intenta con alguna otra curva.

Cuando m es de la forma correcta, un punto P aleatorio es revisado. Si $[m]P = \sigma$, el múltiplo $[s]P$ es revisado. Si $[s]P = \sigma$ (muy poco frecuente), entonces P se descarta y un nuevo punto aleatorio es revisado; por otra parte, r divide la orden de P. Si $r > 4\sqrt{q}$, esta condición garantiza que m es el orden del grupo, ya que no puede ser otro múltiplo de r en el intervalo de Hasse.

6.1.2. Algoritmo de Schoof.

El número de puntos de una curva elíptica *no-supersingular* E definido sobre F_{2^m} satisface el Teorema de Hasse:

$$\#E(F_{2^m}) = 2^m + 1 - t, \quad con \mid t \mid < 2\sqrt{2^m}.$$

Antes de 1985, los únicos métodos conocidos para calcular este número consistían en probar todos los enteros posibles t de está ecuación con sus variantes *baby step-giant step* [Sha71]. La complejidad de estos algoritmos era asimptóticamente de $O(2^{m/4})$.

En 1985 Schoof presenta un algoritmo para calcular el orden de una curva elíptica usando la segunda parte del Teorema de Hasse. El corazón del algoritmo de Schoof es la determinación de t modulo primos l, para $l \leq l_{máx}$ donde $l_{máx}$ es el primo más pequeño tal que

$$\prod_{2 \leq l \leq l_{máx}} l > 4\sqrt{q}$$

Nosotros deducimos t y obtenemos el orden del grupo usando el Teorema Chino del Residuo (Chinese Remainder Theorem).

Del *Teorema de los Números Primos* fácilmente deducimos que el número de primos necesarios es $O(\log q / \log q \log q)$ y entonces el tamaño de $l_{máx} = O(\log q)$.

Nótese que se puede determinar fácilmente el determinante $t \pmod{l}$ para $l = 2$. Para curvas *no-supersingulares* tenemos que $t \equiv 1 \pmod{2}$.

Nosotros consideraremos los primos $l > 2$. El endomorfismo de Frobenius φ de la curva es el mapeo dado por

$$\varphi : \begin{cases} E(\overline{F_q}) \to E(\overline{F_q}) \\ (x, y) \longmapsto (x^q, y^q) \\ \sigma \longmapsto \sigma \end{cases}$$

y para cualquier $P \in \left(\overline{F_q}\right)$ se satisface la ecuación

$$\varphi^2(P) - [t]\varphi(P) + [q]P = \sigma \qquad (A)$$

Consideramos la ecuación para los puntos en $E[l]^* = E[l] \setminus \{\sigma\}$. Sea $q_l \equiv q \,(\text{mód}\, l)$, y $t_l \equiv t \,(\text{mód}\, l)$, donde al menos el representativo no negativo de la clase de congruencia es tomada como q_l y t_l. Si el valor $\tau \in \{0, 1, \ldots, l-1\}$ es encontrado, tal que para un punto $P = (x, y) \in E[l]^*$ tenemos

$$\left(x^{q^2}, y^{q^2}\right) + [q_l](x, y) = [\tau]\left(x^q, y^q\right) \qquad (B)$$

entonces nosotros podríamos tener $\tau = t_l$, como ejemplo t mód l es obtenido. La adición en la fórmula denota la adición de puntos sobre la curva. El valor de τ que satisface la ecuación (B) es única donde l es primo y $P \neq \sigma$.

Para determinar un valor semejante de τ, asumimos por el momento que todos los valores $\tau \in \{0, 1, \ldots, l-1\}$ son probados en turno. Primero, las x-coordenadas sobre ambos lados de la ecuación (B) son calculadas. Las x-coordenadas de los múltiplos del punto $[q_l](x, y)$ y $[\tau]\left(x^q, y^q\right)$, para los primos l dados y los valores de τ siendo probados, son funciones racionales de x y y, que envuelven las divisiones polinomiales. La fórmula de adición de puntos son usadas simbólicamente para calcular las x-coordenadas de $\left(x^{q^2}, y^{q^2}\right) + [q_l](x, y)$. Por denominadores despejados y, si es necesario, eliminando potencias de y mayores que uno para reducir el modulo de la ecuación de la curva (cualquier $y^2 = x^3 + ax + b$ o $y^2 = xy + x^3 + a_6$), resulta en una ecuación de la forma $a(x) - yb(x) = 0$ o $y = a(x)/b(x)$. Así, en su momento, puede ser sustituido dentro de la ecuación de la curva para eliminar y, y tener una ecuación de la forma $h_X(x) = 0$. Una observación crucial en determinar la complejidad del procedimiento es que, desde que el punto P postulado satisface la ecuación (B) que esta en $E[l]^*$, todos los cálculos polinomiales pueden ser llevados fuera del modulo de la división polinomial f_l, el cual es de grado $O(l^2)$. En particular, los polinomios x^{q^2}, y^{q^2}, x^q, y^q son reducidos, usando f_l y la ecuación de la curva, de grado exponencial en $\log q$ a grado polinomial en este parámetro. El grado de $h_X(x)$ es por lo tanto $O(l^2)$.

Para checar si $h_X(x) = 0$ tiene una solución para la x-coordenada de un punto en $E[l]^*$, el *mayor común divisor* (GCD[2]) de $h_X(x)$ y f_l es calculado. Si

[2]Siglas en Inglés de : "Greatest Common Divisor".

el GCD es uno, entonces no hay solución en $E[l]^*$, el cual satisfaga la ecuación (B). y el próximo valor de τ se pone a prueba. Si el GCD es un *non común*, entonces existe un punto en $E[l]^*$ tal que

$$\left(x^{q^2}, y^{q^2}\right) + [q_l](x, y) = \pm[\tau]\left(x^q, y^q\right). \qquad (C)$$

El signo del punto sobre el lado derecho de la ecuación es ambiguo, donde las x-coordenadas no varían para cualquier signo. Para determinar el signo, lo asumimos con signo positivo en la ecuación (C). las y-coordenadas de ambos lados de la ecuación son calculadas y, como con las x-coordenadas, los denominadores son despejados y la variable y eliminada para obtener una ecuación de la forma $h_{\mathcal{Y}}(x) = 0$, con $h_{\mathcal{Y}}$ reducido a grado $O(l^2)$. De nuevo, si $\gcd(h_{\mathcal{Y}}, f_l) \neq 1$. hay un punto que satisface la ecuación y el correcto signo es positivo; si este es negativo no existiría un punto que satisfaga tal ecuación.

Nótese que para un τ dado, el procedimiento actualmente hace la prueba de $\pm\tau$. y esto es sólo necesario para tener a τ ejecutándose entre $0 \leq \tau \leq (l-1)/2$. Generalmente los puntos de $E[l]$ tiene coordenadas en una extensión de campo de F_q. La actual computación de estos puntos, para lo cual podría ser en general muy dificultoso, es aliviado en parte por la computación del GCD.

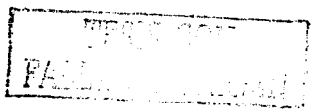
Para examinar la complejidad del algoritmo, nosotros notamos que el volumen de los cálculos se incrementa encontrando x^{q^2}, y^{q^2}, x^q, y^q (convenientemente reducidos al modulo de la ecuación de la curva) modulo f_l, un grado polinomial $O(l^2)$. En el caso de x^q y x^{q^2}, hay operaciones de potenciación en el anillo $F_q[x]/\langle f_l(x)\rangle$, que requieren de $O(\log q)$ multiplicaciones en el anillo. El modulo es de grado $O(l^2) = O\left(\log^2 q\right)$. Ahora, asumiendo que las rutinas de multiplicación no son rápidas, cada multiplicación en el anillo requiere $O\left(\log^2 q\right)$ multiplicaciones de elementos de F_q, cada requerimiento en turno de $O\left(\log^2 q\right)$ operaciones de bit. La complejidad de los elementos y^q, y^{q^2} calculados es similar. Nótese que x^q, y^q, x^{q^2} y y^{q^2}son calculados una vez por cada primo l y usados por todos los valores de τ probados para este primo. Por consiguiente, el número de operaciones de bit necesarios para obtener la traza[3] de un primo simple l es de $O\left(\log^7 q\right)$. Desde que el número de tales primos es $O(\log q)$ (en efecto, $O(\log q/\log\log q)$), la complejidad total para determinar el orden del grupo es $O(\log 8q)$ operaciones de bit.

[3]Se interpreta como "huella o rastro".

Algorithm 20 Algoritmo de Schoof básico.

Entrada: Una curva elíptica E sobre un campo finito F_q. Donde $q = 2^m$.
Salida: El orden de $E(F_q)$.

1. $M \leftarrow 2$. $l \leftarrow 3$ y $S \leftarrow \{(t \,(\text{mód} 2), 2)\}$.
2. Mientras $M < 4\sqrt{q}$ hacer
   a) Para $\tau = 0, \ldots, (l-1)/2$ hacer
      1) Usando las fórmulas descritas anteriormente para tener $P \in E[l]$
      $$\varphi^2(P) + [q]P = \pm[\tau]\varphi(P)$$
      Exactamente cualquier τ que pase la prueba.
3. $S \leftarrow S \bigcup \{(\tau, l)\}$ o $S \leftarrow S \bigcup \{(-\tau, l)\}$, como sea apropiado.
4. $M \leftarrow M \times l$.
5. $l \leftarrow$ primo_próximo l.
6. Recuperar t usando el conjunto S y el *Chinese Remainder Theorem*.
7. Regresar $q + 1 - t$.



6.2. Curvas Koblitz.

Las *curvas binarias anómalas* (también conocidas como *Curvas Koblitz*) son curvas elípticas sobre F_{2^m}, cuyos coeficientes a y b puede ser cualquiera de 0 ó 1. Para propósitos criptográficos se requiere que $b \neq 0$. Estas curvas están definidas por cualquiera de estas ecuaciones:

$$y^2 + xy = x^3 + 1$$

o la ecuación

$$y^2 + xy = x^3 + x^2 + 1.$$

Estas curvas ofrecen muy eficientes implementaciones en criptosistemas de curva elíptica. Las curvas binarias anómalas son un caso especial de curvas de subcampo. Su implementación se justifica de la siguiente manera [WZ98].

Si $m = c \cdot d$ para $c, d \in Z_{>0}$, entonces $F_{2^c} \subset F_{2^m}$. Si a y b son los elementos actuales de F_{2^c}, entonces nosotros decimos que E es una *curva de subcampo*. Nótese en este caso que $E_{(a,b)}(F_{2^c}) \subset E_{(a,b)}(F_{2^m})$.

Si c es pequeño, tanto que el número de puntos en $E_{(a,b)}(F_{2^c})$ pueda ser fácilmente contado, existe una forma fácil para determinar el número de puntos en $E_{(a,b)}(F_{2^m})$. Sabemos que $\#E_{(a,b)}(F_{2^c}) = 2^c + 1 - t$, para $t \leq 2\sqrt{2^c}$, conocido t como la *traçe* de la curva. Si α y β son las dos raíces de la ecuación $X^2 - tX + 2^c = 0$, entonces $\#E_{(a,b)}(F_{2^m}) = 2^m + 1 - \alpha^d - \beta^d$. A esto se le conoce como el *Teorema de Weil*.

El Endomorfismo de Frobenius para las curvas binarias anómalas.

Una interesante propiedad de las curvas binarias anómalas es que si $P = (x, y)$ es un punto sobre la curva, entonces también lo es (x^2, y^2). De hecho $(x^2, y^2) = \lambda P$ para la misma constante λ.. Para observar esta propiedad en el caso general de curvas de subcampo usamos el Endomorfismo de Frobenius.

El Endomorfismo de Frobenius es la función φ que toma x a x^{2^c} para todo $x \in F_{2^m}$. Nótese que $\varphi(r(x)) = r(\varphi(x))$ para todo $x \in F_{2^m}$ y cualquier función racional r con coeficientes en F_{2^c}. Si $P = (x, y)$ es un punto sobre la curva de subcampo E, definimos $\varphi(P) = (\varphi(x), \varphi(y))$. Igualmente definimos $\varphi(\sigma) = \sigma$. Esto puede mostrarnos que la ecuación definida de la curva y el hecho de que $(a + b)^{2^c} = a^{2^c} + b^{2^c}$ para cualquier $a, b \in F_{2^c}$ se logra si $P \in E$

entonces $\varphi(P) \in E$. Así, si E es una curva de subcampo y $P, Q \in E$, entonces $\varphi(P+Q) = \varphi(P) + \varphi(Q)$.

Ahora, considerando un punto $P \in E$ donde E es una curva de subcampo y P tiene un orden primo p con $p^2 = \#E$. Por lo anterior visto, nosotros tenemos que $p\varphi(P) = \varphi(pP) = \varphi(\sigma) = \sigma$. Aquí $\varphi(P)$ podría también ser un punto de orden P. Desde $\varphi(P) \in E$, podemos tener $\varphi(P) = \lambda P$ para cualquier $\lambda \in Z$, $1 \leq \lambda \leq p-1$. El valor λ es una constante entre todos los puntos del subgrupo generado por P y es conocido como el *eigenvalor* del endomorfismo de Frobenius.

Si esta es conocida para cualquier punto $P \in E$, el *endomorfismo de Frobenius* satisface

$$\varphi^2(P) - t\varphi(P) + 2^e P = \sigma$$

donde t es la trace. Por eso, se puede también mostrar que λ es una de las raíces de la congruencia cuadrática

$$X^2 - tX + 2^e \equiv 0 \,(\text{mód} p)\,.$$

Aquí, λ puede ser calculado eficientemente.

6.3. Curvas Elípticas estandarizadas.

La búsqueda de una curva elíptica que proporcione a un criptosistema del mismo tipo, una seguridad óptima y una notable ganancia en la velocidad de los cálculos, suele ser una tarea compleja. Una alternativa viable es la utilización de curvas elípticas creadas por organizaciones estandarizadoras (como ejemplo: NIST). De está manera, tenemos la seguridad de que una curva elíptica recomendada por estas organizaciones, ha sido probada previamente. De lo anterior, resaltamos la siguiente referencia [NIST99] para información más detallada.

La curvas elípticas estándar elegidas, son aquellas para los campos finitos $F_{2^{283}}$ y $F_{2^{409}}$. Estos son sus parámetros:

Para curvas en el campo $F_{2^{283}}$, que tienen la forma

$$E : y^2 + xy = x^3 + x^2 + b$$

en donde b es el coeficiente, r es el orden del punto base, G_x el punto base de la x-coordenada y G_y el punto base de la y-coordenada.

r = 7770675568902916283677847627294075626569625924376904
8891091965267700442777873786928 71

Para la base polinomial:

b = 27b680a c8b8596d a5a4af8a 19a0303f ca97fd76 45309fa2 a581485a f6263e31 3b79a2f5

G_x = 5f93925 8db7dd90 e1934f8c 70b0dfec 2eed25b8 557eac9c 80e2e198 f8cdbecd 86b12053

G_y = 3676854 fe24141c b98fe6d4 b20d02b4 516ff702 350eddb0 826779c8 13f0df45 be8112f45 be8112f4

Para las curvas Koblitz en el campo $F_{2^{283}}$, que tienen la forma

$$E_a : y^2 + xy = x^3 + ax^2 + 1$$

en donde el coeficiente $a = 0$ ó 1, r es el orden del punto base, G_x el punto base de la x-coordenada y G_y el punto base de la y-coordenada.

$a = 0$

r = 3885533778445145814183892381364703781328481173379306132429587499752981582970442260 3873

Para la base polinomial:

G_x = 503213f 78ca4488 3f1a3b81 62f188e5 53cd265f 23c1567a 16876913 b0c2ac24 58492836

G_y = 1ccda38 0f1c9e31 8d90f95d 07e5426f e87e45c0 e8184698 e4596236 4e341161 77dd2259

Ambas curvas pueden implementarse sobre una representación del campo en base polinomial determinado por el pentanomio $p(t) = t^{283} + t^{12} + t^7 + t^5 + 1$ para el campo $F_{2^{283}}$.

Para curvas en el campo $F_{2^{409}}$, que tienen la forma

$$E : y^2 + xy = x^3 + x^2 + b$$

en donde b es el coeficiente, r es el orden del punto base, G_x el punto base de la x-coordenada y G_y el punto base de la y-coordenada.

r = 66105596879024859895191530803277103982840468296428121
928464879830415777482737480520814372376217911096597986
72288366567526771.

b = 021a5c2 c8ee9feb 5c4b9a75 3b7b476b 7fd6422e f1f3dd67 4761fa99
d6ac27c8 a9a197b2 72822f6c d57a55aa 4f50ae31 7b13545f.

G_x =15d4860 d088ddb3 496b0c60 64756260 441cde4a f1771d4d
b01ffe5b 34e59703 dc255a86 8a118051 5603aeab 60794e54
bb7996a7.

G_y = 061b1cf abbe5f3 2bbfa783 24ed106a 7636b9c5 a7bd198d 0158aa4f
5488d08f 38514f1f df4b4f40 d2181b36 81c364ba 0273c706.

Para las curvas Koblitz en el campo $F_{2^{409}}$, que tienen la forma

$$E_a : y^2 + xy = x^3 + ax^2 + 1$$

en donde el coeficiente $a = 0$ ó 1, r es el orden del punto base, G_x el punto base de la x-coordenada y G_y el punto base de la y-coordenada.

$a = 0$

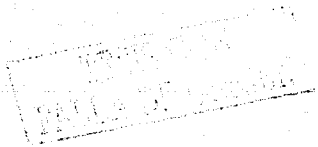
r = 33052798439512929947595765401638551991420234148214060
964232439502288071128924919105067325843777745801409636659909617731358671

G_x =060f05f 658f49c1 ad3ab189 0f718421 0efd0987 e307c84c 27accfb8
f9f67cc2 c460189e b5aaaa62 ee222eb1 b35540cf e9023746

G_y = 1e36905 0b7c4e42 acba1dac bf04299c 3460782f 918ea427 e6325165
e9ea10e3 da5f6c42 e9c55215 aa9ca27a 5863ec48 d8e0286b

Para el campo $F_{2^{409}}$, ambas curvas pueden implementrase sobre una representación del campo finito en base polinomial determinado por el polinomio $p(t) = t^{109} + t^{87} + 1$.

Los parámetros r, G_x, G_y pueden ser diferentes, si el criptosistema de curva elíptica implementado los determina aleatoriamente. Sin embargo, los parámetros a y b respectivos de cada ecuación permanecen constantes y son determinantes de la seguridad del criptosistema.



7. Implementación del Criptosistema.

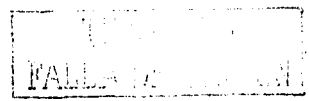
En 1985, V. Miller y N. Koblitz sugirieron las curvas elípticas para propósitos criptográficos; en aquella época, era difícil desarrollar e implementar los cálculos necesarios, las matemáticas en que se fundamentaban no eran muy eficientes, y su interés quedaba como ejercicio académico.

Desde entonces, la evolución de las matemáticas con aplicaciones computacionales, y la creación de código programable que validara estas matemáticas, han experimentado un notable crecimiento.

Se cuentan por cientos, la cantidad de publicaciones que tratan sobre la criptografía de curva elíptica, y de las matemáticas que la conforman. También, la existencia de una amplia variedad de código libre escrito en diversos lenguajes de programación confirman un crecimiento continuo en está área.

La utilización de código reutilizable de forma parcial o total, enfoca nuestros esfuerzos al tratamiento de los detalles de implementación más sobresalientes, citando la elección del campo finito y de la curva elíptica ideal como algunos ejemplos. Considerando, en todo momento, el aprovechamiento de los algoritmos óptimos para el mecanismo de nuestro criptosistema.

Resaltamos el trabajo realizado para la implementación de criptosistemas de curva elíptica realizado por Michael Rosing [Ros99] en lenguaje C, de entre una variedad de código existente para propósitos similares. El código contenido en el trabajo al que hacemos referencia, se encuentra perfectamente documentado, utilizando la mayoría de los algoritmos que en está investigación se detallan, contemplando las características que el estándar IEEE P1363 dictamina para este tipo de implementaciones [IEEE00]. El autor del código proporciona y usa referencias de otros autores.



Las pruebas de los criptosistemas de curva elíptica implementados para está investigación, son realizadas bajo este código con algunas modificaciones.

El algoritmo de Schoof, del que hacemos referencia en el capitulo anterior. se implementa en está investigación haciendo uso de una librería para la multiprecisión aritmética llamada MIRACL proporcionada libremente por *"Shamus Software Ltd."* [MIRACL]. De hecho, el código que conforma este algoritmo, es incluido por MIRACL como parte de un conjunto de herramientas previamente compiladas.

7.1. Componentes generales.

Los criptosistemas implementados fuerón conformados siguiendo las especificaciones de tres protocolos que difieren entre ellos en el tratamiento para el secreto compartido. Los algoritmos y características matemáticas de los protocolos empleados en nuestros criptosistemas, son tratados en el capitulo 5. Los protocolos usados son: el protocolo de Diffie-Hellman, ElGamal para curvas elípticas y el esquema de acuerdo de claves de Menezes-Qu-Vanstone (MQV). Nombramos respectivamente a cada criptosistema de la siguiente forma: ECDH. ECElGamal y ECMQV.

Todos los criptosistemas cuentan con los siguientes parámetros comunes: la mitad de estos criptosistemas cuentan con un campo finito de característica dos - $F_{2^{283}}$. una base polinomial determinado por el polinomio irreducible $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$ para el campo finito $F_{2^{283}}$; la mitad restante, cuenta con un campo finito de característica dos - $F_{2^{409}}$, y una base polinomial determinado por el polinomio irreducible $f(x) = x^{409} + x^{87} + 1$ para el campo finito $F_{2^{409}}$.

Cada criptosistema, es implementado y probado con curvas elípticas E generadas por estas dos técnicas: curvas elípticas aleatorias, y curvas elípticas de tipo Koblitz. Para cada criptosistema creado, una curva de cada técnica es probada.

Los criptosistemas implementados, están conformados por un conjunto de rutinas específicas escritas en lenguaje C, los cuales interpretan los algoritmos que aquí se representan. A su vez, estas rutinas están estructuradas en archivos de código (subprogramas), que integran cada uno de los sistemas

criptográficos.

Estos son los subprogramas para cada criptosistema:

- *field2n.h* .- Este archivo de cabecera nos ayuda a definir el campo finito.
- *poly.h* .- Este archivo de cabecera es requerido para la sección de matemáticas polinomiales. Su propósito es definir el tamaño de estructuras intermedias para uso con las rutinas de división y multiplicación.
- *eliptic.h* .- Aquí se definen la estructuras usadas para crear los puntos en la curva elíptica y sus parámetros.
- *poly_func.c* .- Las rutinas necesarias para resolver la ecuación de la curva elíptica, se encuentran aquí.
- *polymain.c* .- Aquí se especifica el polinomio irreducible tratado con las matemáticas de base polinomial. Este archivo realiza una prueba al polinomio irreducible, verificando que realmente sea polinomio primo. Incluye también las rutinas necesarias para las operaciones básicas en base polinomial.
- *poly_protocol.c* .- Especifica las rutinas necesarias para implementar el protocolo elegido.
- *eliptic_poly.c* .- Funciones de la curva elíptica implementados en la base polinomial del campo finito binario elegido.

7.2. Representación del campo finito F_{2^m}.

El archivo de cabecera *field2n.h*, proporciona en código, la representación del campo finito binario elegido para la operación de los criptosistemas:

```
*** field2n.h ***/

#define WORDSIZE      (sizeof(int)*8)
#define NUMBITS       283
```

```
#defineÑUMWORD      (NUMBITS/WORDSIZE)
#define UPRSHIFT     (NUMBITS%WORDSIZE)

#define MAXLONG      (NUMWORD−1)

#define MAXBITS      (MAXLONG*WORDSIZE)
#define MAXSHIFT     (WORDSIZE-1)
#define MSB          (1L<<MAXSHIFT)

#define UPRBIT       (1L<<(UPRSHIFT-1))
#define UPRMASK      (~(-1L<<UPRSHIFT))
#define SUMLOOP(i)    for(i=0; i<MAXLONG; i++−)

typedef short int INDEX;

typedef unsigned long ELEMENT;

typedef struct {
      ELEMENT        e[MAXLONG];
} FIELD2N;
```

El parámetro WORDSIZE lo determina la computadora en donde el código es ejecutado. Este parámetro define palabras de datos en plataformas de 32 bits. Si los criptosistemas representados son transportados y compilados en plataformas de 64 bits, necesariamente este parámetro tendrá que ser modificado bajo las especificaciones del tamaño de un byte para estas plataformas.

El total de número de bits es determinado por NUMBITS, esto representa el número de elementos contenidos dentro de un campo finito binario. NUMWORD es el índice máximo dentro de un arreglo de palabras de máquina usados en un arreglo que contiene NUMBITS. MAXLONG es el número de palabras de máquina necesarios para guardar un polinomio. Por último, el parámetro ELEMENT es una palabra de máquina representativa del campo finito binario, representado dentro de la estructura definida por FIELD2N.

7.3. Representación de la curva elíptica.

Representamos en nuestros criptosistemas a la curva elíptica elegida, con el archivo de cabecera *eliptic.h*. Como la curva elíptica está definido sobre un campo finito binario, todos los valores de la ecuación son elementos del campo. Los coeficientes de la curva se representan así:

```
typedef struct
{
    INDEX  form;
    FIELD2N a2;
    FIELD2N a6;
} CURVE;
```

El valor a2 y a6 representan los coeficientes de una curva elíptica a_2 y a_6. En donde *form* es 0 para a2 = 0; cuando *form* sea 1, entonces a2 = 1. El elemento a6 obtiene su valor conforme a nuestra elección de la curva elíptica apropiada, y de las técnicas usadas para obtener dicha curva. Nótese que estos elementos. son estructuras de FIELD2N; es decir, son elementos del campo elegido.

Las coordenadas de un punto de una curva elíptica (x, y), forman parte del campo finito binario elegido (estructura FIELD2N). Se representa el par de coordenadas de un punto en la curva de la siguiente forma:

```
typedef struct
{
    FIELD2N x;
    FIELD2N y;
} POINT;
```

7.4. Representación de la base polinomial.

Los cálculos inmersos en la operación de los criptosistemas, son implementados en la base polinomial que representa los elementos conformados en el campo finito binario elegido. Para las rutinas de división y multiplicación,

es necesario implementar estructuras adicionales intermedias. El archivo de cabecera *poly.h*, contiene estructuras las cuales son el doble de grandes (en bits) que el tamaño del campo elegido.

```
#define DBLBITS        2*NUMBITS
#define DBLWORD         (DBLBITS/WORDSIZE)
#define DBLSHIFT       (DBLBITS%WORDSIZE)
#define MAXDBL         (DBLWORD+1)

#define DERIVMASK       0x55555555

#define DBLLOOP(i)      for(i=0; i<MAXDBL; i--)

typedef struct {
      ELEMENT        e[MAXDBL];
} DBLFIELD;
```

El término DERIVMASK es usado para checar polinomios primos. Un campo de doble tamaño llamado DBLFIELD se usa para definir el arreglo de palabras máquina que contienen el resultado de una multiplicación.

7.4.1. El polinomio irreducible.

El polinomio irreducible o primo, es definido en el archivo *polymain.c* de la siguiente forma:

Para el campo finito $F_{2^{283}}$, se define así:

```
FIELD2N poly_prime = {0x08000000,0x00000000,0x00000000,0x00000000,
0x00000000.0x00000000,0x00000000,0x00000000,0x000010a1}; /*283*/
```

Para el campo finito $F_{2^{409}}$, se define así:

```
FIELD2N poly_prime = {0x02000000,0x00000000,0x00000000,0x00000000,
0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,
0x00800000,0x00000000,0x00000001}; /*409*/
```

La representación de cada palabra máquina está en código hexadecimal; es decir, cada dígito representa a partir de 0x de cada palabra, cuatro bits. En donde, cada posición de bit, representa el coeficiente de una potencia de la literal del polinomio. El indicador 0x, nos informa que la representación de cada palabra máquina, está en formato hexadecimal.

La representación del polinomio elegido, debe guardar la proporción adecuada al número de bits (tamaño del campo) escogido (NUMBITS).

Para nuestro ejemplo con el campo finito $F_{2^{283}}$, se implementa un polinomio irreducible o primo, en nueve palabras máquina; como cada palabra representa 32 bits, entonces se tiene un número de total de bits de 288, en donde solo los 283 bits son los representativos de la longitud del campo.

7. Implementación del Criptosistema.

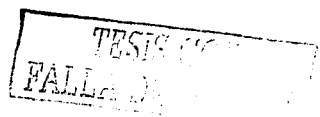
Conclusiones.

Consideraciones Previas.

Los criptosistemas de curva elíptica fueron construidos con tres protocolos diferentes: ECCElGamal, ECCDH y ECCMQV. Empleando en su construcción dos tipos de curvas: las del tipo Koblitz y curvas aleatorias, ambas con sus respectivas variantes (cuando el coeficiente a_2 de la ecuación de la curva es uno o cero). Las curvas propuestas por NIST para campos $F_{2^{283}}$ y $F_{2^{409}}$ tienen la misma complejidad como aquellas curvas aleatorias generadas para los campos mencionados; por estas razones, y para propósitos de nuestras pruebas, curvas propuestas por NIST u otro organismo estandarizador no son considerados de forma particular.

Para documentar aún más los resultados obtenidos, fueron considerados dos campos finitos binarios: $F_{2^{283}}$ y $F_{2^{409}}$. El propósito de construir criptosistemas de curva elíptica y realizar pruebas de ejecución de estas mismas en tales campos, es comprobar el tiempo de ejecución de estos criptosistemas con respecto a la longitud en bits del mensaje a encriptar.

Los criptosistemas de curva elíptica generados para estas pruebas, fueron desarrollados en lenguaje C, con el compilador **gcc versión 2.96-110 (GNU-C)**. El equipo de computo utilizado para la construcción y ejecución de los criptosistemas, es una PC - Pentium III a 800 MHz, cuyo sistema operativo es Linux con kernel 2.4.18.



```
ECCGamal/a2=0/a6=1/m=409          eccgam-a3.txt          Página 1/1
poly_prime =
  2000000         0         0         0         0         0         0         0         0
        0    800000         0         1
create Base curve and point

Public curve
form: 0
a6: 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Base point
x:   7cf965 63323eab 10fc68f8 254d4d11 d2d518f2 9979dd24 f3814df5 5d0be8bb a913f
cdb 91edee60 4da6d486 295d85ac 44043ee1
y:   b7300b acf2f289 c709a5d3 ba9d5263 1d9f90dd  250cd43 dbd49a61  ebcd574 2f470
983 812d7059 c4551241 b4342118 e84557dd

create side 2's private key

Side 2 secret:
   f94f6 e0caf9a7 2d189f04 8591c5e5 3935d4dc 3d80d402 741aa073 fee656e5 b5b92389
 1e124560 5c6e42e9 8ca58aef b9bb4e77

Generate side 2's public key

Side 2 public key
x:   14d888d f4cecb80  84503db b021d805 42de2930 ab0f4788  f16d5ab 9f84612c b4709
7de 679fedf2 a173ad89 7af471ca 17d5754b
y:   5e732d  168d287 da7b8edf 9dcddbf9 368bd40f 9e4a737a 1df564bd de95c874 b279c
aa5 cdfa4f4d eb1bc94c e4e0b9d3 4dfc939c

Create message data


Hide data on curve and send from side 1 to side 2

Hidden data
x:   5744d44 d9c0a41f 6bc0509f 9235b260  ad30a99 2e56e2f7 a38f0d1f ddca0ce2 46648
a0f d1cd2ed4 42fb2938 9c2c54b0  22a1cca
y:   19fa357 b9f1041e a356677e d0375ee5 bb63c49b cca81a7c b6ac2cb7 426c6f81 5b570
8f9 e8befeb7 23cab1b1 2d5aa1b0 a02cfd08
Random point
x:   1d695b8 ab2acccb c4c189b9 33716c6f 16fb9e35 39bbb129 78eadb35 cbb354c7 5749b
1ed db9be01c 40662b27 d31bd99c 761bf9ed
y:   7f7727 690c5ddd 64dc3876 c95bd546 1a011970 f0e2039b 2c5319b8 daea7ac6 1accf
a65  9e470f6 b98a1944 920967d6 d30d1866

Recover transmitted message

sent data
  b3823b ac2a507e 1d932b35 60357d85 cdd1283e eb1b17df fefb88d6 dd3c9088 93862b65
 56504d14 2fbb468d 9eddb171 46894bc1
received data
  b3823f ac2a507e 1d932b35 60357d85 cdd1283e eb1b17df fefb88d6 dd3c9088 93862b65
 56504d14 2fbb468d 9eddb171 46894bc1

(1)--> 0 segs, 706889 usgs. Creando curva y punto aleatorio
(2)--> 30 segs, -387347 usgs. Generando llaves para lado 2
(3)--> 58 segs, 507256 usgs. Ocultando datos y enviandolos
(4)--> 30 segs, -59286 usgs. Recuperando mensaje y decodificandolo
```

Figura 7.1.: Vista del criptosistema ECCGamal con $a_2 = 0$, y $a_6 = 1$.

```
ECCDH/a2=1/a6=1/m=409                    eccdh-a4.txt                    Página 1/1
poly prime =
 2000000          0          0          0          0          0          0          0          0
       0     800000          0          1
create Base curve and point

random curve
form: 1
a2: 0 0 0 0 0 0 0 0 0 0 0 0 0 1
a6: 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Base point
x:   7cf965 63323eab 10fc68f8 254d4d11 d2d518f2 9979dd24 f3814df5 5d0be8bb a913f
cdb 91edee60 4da6d486 295d85ac 44043edd
y:   1231bba f290cfe4 bf75df7b 4f8cd839 807d68b6 83becb6c f8d0aa04 8ef843ef db5e2
750 3de0a3c7 ee8aee35  4ca3e1a 97223432

create each sides private key

Side 1 secret:
   f94f6 e0caf9a7 2d189f04 8591c5e5 3935d4dc 3d80d402 741aa073 fee656e5 b5b92389
 1e324560 5c6e42e9 8ca58aef b9bb4e77
Side 2 secret:
  b3823b ae2a507e 1d932b35 60357d85 cdd1283e eb1b17df fefb88d6 dd3c9088 93862b65
 56564d14 2fbb4c8d 9eddb171 46894bc1

Generate each sides public key

Side 1 public key
x:   1271dfb d5f7a674 524b4ca5 c1c7bd41 c197db62 c02a4768 63d9102f b64fe8a5 ff896
fa3 3fc654a8 ec752411 8384901b 17509767
y:   42294f e1102d40 52b4428f f5644504 5aa52148 d2f7df1b  f1f1c51 4d28e5fd 70b82
a36 261ca3c1 6746704b 4c6acd46 8aa65774
Side 2 public key
x:   fca348 7c84a029 b45a6c96 e85e698e 709b1163 8e065dd8 226856ce  a92bc92 f625b
89b fbcc4055 45f0e7d4 3c9741cd 22669a01
y:   f070ca 84c9d207  86f6fcc 18979d36 358d9556 e0af98a4 f8be47f1 53319227 c9bb9
5a3 96c5beeb 52c8cbf4 5c3b6aab 4655c0d7

Show that each side gets the same shared secret

key 1 is
 19a12ef 96645d73 d1b3a86a  10d0554 d22e8447 8d1e88c1 3ef45e4d 725d9ff6 62078d0d
 f228a946 abca5bb6 27a0e611 82538775
key 2 is
 19a12ef 96645d73 d1b3a86a  10d0554 d22e8447 8d1e88c1 3ef45e4d 725d9ff6 62078d0d
 f228a946 abca5bb6 27a0e611 82538775

(1)--> 1 segs, 487006 usgs. Creando curva y punto aleatorio
(2)--> 0 segs, 51 usgs. Generando llaves privadas de ambos lados
(3)--> 60 segs, 18129 usgs. Generando llaves publicas de ambos lados
(4)--> 60 segs, 62793 usgs. Mostrando el mensaje de ambos lados
```

Figura 7.2.: Vista del criptosistema ECCDH con $a_2 = 1$, y $a_6 = 1$.

Campo tamaño.	Curva Tipo.	Creando curva y punto aleatorio.	Generando claves privadas. Ambos lados.	Generando claves públicas. Ambos lados.	Decodificando mensaje. Ambos lados.
$F_{2^{283}}$	$a_2 = 0$, $a_6 =$ *aleatorio*	0 segs, 207479 usegs.	0 segs, 38 usegs.	15 segs, 380552 usegs.	16 segs, 562494 usegs.
$F_{2^{283}}$	$a_2 = 1$, $a_6 =$ *aleatorio*	0 segs, 213104 usegs.	0 segs, 40 usegs.	15 segs, 310393 usegs.	16 segs, 675273 usegs.
$F_{2^{283}}$	$a_2 = 0$, $a_6 = 1$	0 segs, 245946 usegs.	0 segs, 38 usegs.	15 segs, 383467 usegs.	16 segs, 592316 usegs.
$F_{2^{283}}$	$a_2 = 1$, $a_6 = 1$	1 segs, 804979 usegs.	0 segs, 37 usegs.	16 segs, 51655 usegs.	16 segs, 68708 usegs.
$F_{2^{409}}$	$a_2 = 0$, $a_6 =$ *aleatorio*	1 segs, 473615 usegs.	0 segs, 52 usegs.	61 segs, 136102 usegs.	61 segs, 4697 usegs.
$F_{2^{409}}$	$a_2 = 1$, $a_6 =$ *aleatorio*	1 segs, 358282 usegs.	0 segs, 50 usegs.	61 segs, 49744 usegs.	61 segs, 13886 usegs.
$F_{2^{409}}$	$a_2 = 0$, $a_6 = 1$	0 segs, 742718 usegs.	0 segs, 50 usegs.	62 segs, 557604 usegs.	61 segs, 620646 usegs.
$F_{2^{409}}$	$a_2 = 1$, $a_6 = 1$	1 segs, 487006 usegs.	0 segs, 51 usegs.	60 segs, 18129 usegs.	60 segs, 62793 usegs.

Cuadro 7.1.: Tiempos estimados de ejecución en ECCDH.

Resultados.

Los protocolos criptográficos que esta tesis resume, cada uno de ellos, contiene un conjunto de tareas necesarias para el cumplimiento del objetivo básico de cada protocolo. Los resultados mostrados son los tiempos de ejecución de cada una de estas tareas por protocolo.

Cada tarea correspondiente a cada protocolo criptográfico de los aquí mostrados, consume un determinado tiempo de ejecución que varía dependiendo del tamaño del campo finito, y el tipo de curva implementado por protocolo. Algunas de estas tareas muestran tiempos de ejecución más rápidos que otros ante determinados tipos de curvas:

- Para ECCDH . dividimos las tareas en cuatro: creación de curva base y

punto aleatorio, la generación de claves privadas en ambos lados, generación de claves públicas en ambos lados, y la decodificación del mensaje en ambos lados.

1. **Creación de curva base y punto aleatorio.** El tiempo de cálculo es más rápido en las curvas de tipo aleatorias; sin embargo, cuando $a_2 = 1$, los tiempos de ejecución para esta tarea son discretamente más rápidas, cuando el campo finito crece de tamaño a 409 bits. En el caso de las curvas de tipo Koblitz, los tiempos de cálculo son mejorados notablemente cuando $a_2 = 0$.
2. **Generación de claves privadas en ambos lados.** No se encuentran diferencias importantes en los tiempos de cálculo para está tarea.
3. **Generación de claves públicas en ambos lados.** Para el campo finito de tamaño a 283 bits, el mejor tiempo de ejecución es para las curvas de tipo aleatorias cuando $a_2 = 1$. Cuando el campo finito es del tamaño de 409 bits, el mejor desempeño en tiempo para esta tarea lo muestra la curva de tipo Koblitz con $a_2 = 1$.
4. **Decodificación del mensaje en ambos lados.** Las curva de tipo Koblitz con $a_2 = 1$, proporciona los mejores tiempos de ejecución en campos finitos de tamaño a 283 bits y 409 bits.

- Para medir el desempeño en tiempos de ejecución del protocolo EC-CGamal , dividimos este protocolo en cuatro tareas: creación de curva base y punto aleatorio, generación de las claves pública y privada para el lado dos, encriptación y envió de datos, recuperación y decodificación del mensaje.

1. **Creación de curva base y punto aleatorio.** Se muestra una ventaja discreta de las curvas de tipo aleatorias con respecto a la curva de tipo Koblitz, cuando $a_2 = 1$ en el campo finito de tamaño a 283 bits. En el campo finito de tamaño a 409 bits, la curva de tipo Koblitz con $a_2 = 1$ mostró mejor desempeño en tiempos de ejecución. La ventaja de este tipo de curva es modesta con respecto a las curvas de tipo aleatorias cuando $a_2 = 0$.

Campo tamaño.	Curva Tipo.	Creando curva y punto aleatorio.	Generando claves. Lado dos	Encriptando datos y enviandolos.	Recuperando y decodificando mensaje.
$F_{2^{283}}$	$a_2 = 0$, $a_6 =$ *aleatorio*	0 segs, 200184 usegs.	8 segs, 16995 usegs.	16 segs, 236199 usegs.	8 segs, 63422 usegs.
$F_{2^{283}}$	$a_2 = 1$, $a_6 =$ *aleatorio*	0 segs, 175337 usegs.	8 segs, 97029 usegs.	15 segs, 550052 usegs.	8 segs, 32079 usegs.
$F_{2^{283}}$	$a_2 = 0$, $a_6 = 1$	0 segs, 222865 usegs.	8 segs, 185463 usegs.	16 segs, 426514 usegs.	8 segs, 197546 usegs.
$F_{2^{283}}$	$a_2 = 1$, $a_6 = 1$	0 segs, 173485 usegs.	7 segs, 650124 usegs.	16 segs, 748881 usegs.	7 segs, 671030 usegs.
$F_{2^{409}}$	$a_2 = 0$, $a_6 =$ *aleatorio*	0 segs, 514709 usegs.	30 segs, 136824 usegs.	61 segs, 630957 usegs.	30 segs, 95505 usegs.
$F_{2^{409}}$	$a_2 = 1$, $a_6 =$ *aleatorio*	0 segs, 625403 usegs.	30 segs, 232570 usegs.	60 segs, 215108 usegs.	30 segs, 278762 usegs.
$F_{2^{409}}$	$a_2 = 0$, $a_6 = 1$	0 segs, 706889 usegs.	30 segs, 387347 usegs.	58 segs, 507056 usegs.	30 segs, 59286 usegs.
$F_{2^{409}}$	$a_2 = 1$, $a_6 = 1$	1 segs, 500397 usegs.	31 segs, 237847 usegs.	60 segs, 319256 usegs.	31 segs 256935 usegs.

Cuadro 7.2.: Tiempos estimados de ejecución en ECCGamal.

2. **Generación de las claves pública y privada para el lado dos.** Cuando el tamaño del campo finito es de 283 bits, la curva de tipo Koblitz con $a_2 = 1$, muestra el mejor tiempo de ejecución. Pero, cuando el tamaño del campo finito es de 409 bits, la curva mencionada es la de peor desempeño; resaltando el desempeño eficiente de las curvas de tipo aleatorio, en tiempos de ejecución cuando $a_2 = 0$.
3. **Encriptación y envió de datos.** Las curvas de tipo aleatorias con $a_2 = 1$ muestran tiempos de ejecución rápidos en ambos tamaños del campo finito. Cuando el tamaño de campo es de 409 bits, supera a la curva anterior la curva de tipo Koblitz con $a_2 = 0$.
4. **Recuperación y decodificación del mensaje.** Las curvas de tipo Koblitz ofrecen el mejor desempeño en tiempos de ejecución para esta tarea. Cuando $a_2 = 1$ en curvas de tipo Koblitz, el mejor desempeño en tiempos de ejecución es para el campo de tamaño a 283 bits; el efecto es contrario cuando el tamaño del campo es de 409 bits. Cuando $a_2 = 0$ en curvas de tipo Koblitz, el mejor desempeño en tiempos de ejecución se muestra en campos de tamaño a 409 bits.

Campo tamaño	Curva tipo	Creación de curva y punto aleatorio	Generando claves privadas. Ambos lados	Generando claves públicas. Ambos lados.	Generación de datos efímeros. Ambos lados.	Decodificación e Intercambio de datos
$F_{2^{283}}$	$a_2 = 0, a_6 =$ *aleatorio*	0 segs, 183937 usegs.	0 segs, 41 usegs.	16 segs, 89394 usegs.	16 segs, 320775 usegs.	95 segs, 2073 usegs.
$F_{2^{283}}$	$a_2 = 1, a_6 =$ *aleatorio*	0 segs, 194787 usegs.	0 segs, 39 usegs.	16 segs, 309828 usegs.	15 segs, 526941 usegs.	95 segs, 735548 usegs.
$F_{2^{283}}$	$a_2 = 0, a_6 =$ 1	0 segs, 196333 usegs.	0 segs, 39 usegs.	15 segs, 635815 usegs.	16 segs, 332315 usegs.	94 segs, 480671 usegs.
$F_{2^{283}}$	$a_2 = 1, a_6 =$ 1	0 segs, 175020 usegs.	0 segs, 39 usegs.	15 segs, 253561 usegs.	15 segs, 341846 usegs.	92 segs, 554248 usegs.
$F_{2^{409}}$	$a_2 = 0, a_6 =$ *aleatorio*	1 segs, 428216 usegs.	0 segs, 51 usegs.	60 segs, 133440 usegs.	60 segs, 149232 usegs.	358 segs, 315035 usegs.
$F_{2^{409}}$	$a_2 = 1, a_6 =$ *aleatorio*	1 segs, 504108 usegs.	0 segs, 50 usegs.	59 segs, 359712 usegs.	59 segs, 23872 usegs.	358 segs, 114699 usegs.
$F_{2^{409}}$	$a_2 = 0, a_6 =$ 1	0 segs, 556860 usegs.	0 segs, 51 usegs.	60 segs, 840934 usegs.	59 segs, 67594 usegs.	357 segs, 703395 usegs.
$F_{2^{409}}$	$a_2 = 1, a_6 =$ 1	1 segs, 498177 usegs.	0 segs, 52 usegs.	61 segs, 64548 usegs.	61 segs, 176221 usegs.	370 segs, 75571 usegs.

Cuadro 7.3.: Tiempos estimados de ejecución en ECCMQV.

- Como en los casos anteriores, para medir el desempeño en tiempos de ejecución del protocolo ECCMQV , dividimos este protocolo en cinco tareas: creación de curva base y punto aleatorio, generación de claves privadas en ambos lados, generación de claves públicas en ambos lados, generación de datos efímeros en ambos lados, decodificación e intercambio de datos.

1. **Creación de curva base y punto aleatorio.** Las curvas de tipo Koblitz muestran los mejores tiempos de ejecución. Para el campo finito de tamaño a 283 bits, la curva de tipo Koblitz con $a_2 = 1$ tiene el mejor tiempo de ejecución. Para el campo finito de tamaño a 409 bits, la curva de tipo Koblitz con $a_2 = 0$ ofrece el mejor tiempo de ejecución.

2. **Generación de claves privadas en ambos lados.** No se observan diferencias importantes en tiempos de ejecución para esta tarea.
3. **Generación de claves públicas en ambos lados.** Para el campo finito de tamaño a 283 bits, las curvas de tipo Koblitz muestran los mejores tiempos de ejecución, sobresaliendo la curva de tipo Koblitz con $a_2 = 1$. Las curvas de tipo aleatorias destacan con lo mejores resultados en tiempos de ejecución cuando el tamaño del campo es de 409 bits, sobresaliendo en resultados las curvas de tipo aleatorias con $a_2 = 1$.
4. **Generación de datos efímeros en ambos lados.** En el campo finito de tamaño a 283 bits, las curvas con $a_2 = 1$ de ambos tipos (Koblitz y aleatorias) muestran el mejor desempeño en tiempos de ejecución. Cuando el campo finito es de tamaño a 409 bits, destaca ligeramente la curva aleatoria con $a_2 = 1$.
5. **Decodificación e intercambio de datos.** Cuando $a_2 = 1$, se muestra el mejor resultado en tiempos de ejecución en la curva de tipo Koblitz para esta tarea con el tamaño de campo a 283 bits. Cuando $a_2 = 0$, el mejor desempeño en tiempos de ejecución es en la curva de tipo Koblitz con el tamaño de campo a 409 bits.

Discusión.

Los resultados anteriores mostraron que en algunas tareas de los protocolos propuestos, las curvas de tipo Koblitz resaltan ligeramente en el desempeño de los tiempos de ejecución. Sin embargo, el conocimiento de las propiedades de este tipo de curvas, son bien conocidas por ser las primeras curvas propuestas para la criptografía de curvas elípticas. Esta simple razón, puede ser suficiente como para no elegir este tipo de curvas, por razones de seguridad, suponiendo que se desea obtener el mayor nivel de seguridad posible. Aunque aún no existen bases solidas que avalen esta aseveración, el mayor conocimiento obtenido de estas curvas supondría una solución más rápida al ECDLP.

Las curvas de tipo Koblitz se usarían entonces. en los casos que se requiera en comunicaciones de datos, seguridad media.

Por el contrario, las curvas aleatorias, suponen menor conocimiento de sus propiedades de cálculo; pero tienen el inconveniente de que existen algunas curvas de este tipo, no óptimas para propósitos criptográficos, aunque la probabilidad de elegir aleatoriamente una curva así, es muy baja. También se recomienda su uso para entornos que requieran un nivel de seguridad media.

Para niveles de seguridad altos, se recomienda curvas probadas por algún organismo estandarizador, citando a NIST como ejemplo; o alguna curva seleccionada aleatoriamente y verificada por algún algoritmo contador de puntos, como el algoritmo de Schoof. También las curvas generadas por algún algoritmo que haga uso de la teoría de la multiplicación compleja, podría ser un recurso viable en la búsqueda de la máxima seguridad.

Recientemente, matemáticos de la Universidad de Notre Dame encabezados por Chris Monico dieron solución a un problema de encriptación de claves ECCp-109, propuesto como reto por Certicom Corporation de Canada.[1] (6 de Noviembre del 2002). El problema de encriptación de claves ECCp-109 fue resuelto usando el poder de computo de 10,000 computadoras (la mayoría de ellas, computadoras personales PC's), trabajando en el problema durante 24 horas al día durante 549 días.

Certicom es una empresa canadiense pionera en el desarrollo y fortalecimiento de la criptografía de curva elíptica, y fue la primera empresa en poner está tecnología al mercado.

Existe otro reto similar propuesto por Certicom para encontrar la solución al problema de encriptación de claves ECCp-131, en donde el equipo que llegase a ser ganador ganaría $20,000 U.S. La solución a este problema requiere de cientos de veces más poder de computo con respecto al problema ECCp-109.

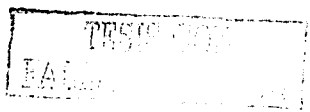
Estos retos no demeritan la efectividad de la criptografía de curva elíptica: por el contrario, destacan la dificultad de resolver el problema ECDLP empleando recursos computacionales numerosos y caros.

Para resolver, el problema de ECDLP de un criptosistema de curva elíptica con un tamaño de campo finito a 283 bits, empleando el concepto de "fuerza bruta", sería necesario miles de veces más poder de computo que el empleado para resolver el ECDLP del problema ECCp-109 propuesto por Certicom.

[1]Para mayores informes de este caso, refierase a : http://www.certicom.com/

Parte I.

Listado de Programas.

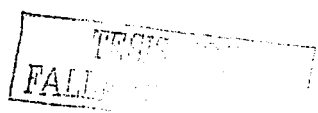


```
/*** field2n.h ***/

#define WORDSIZE          (sizeof(int)*8)
#define NUMBITS           283

#define NUMWORD           (NUMBITS/WORDSIZE)
#define UPRSHIFT          (NUMBITS%WORDSIZE)

#define MAXLONG           (NUMWORD+1)

#define MAXBITS           (MAXLONG*WORDSIZE)
#define MAXSHIFT          (WORDSIZE-1)
#define MSB                     (1L<<MAXSHIFT)

#define UPRBIT            (1L<<(UPRSHIFT-1))
#define UPRMASK           (~(-1L<<UPRSHIFT))
#define SUMLOOP(i)        for(i=0; i<MAXLONG; i++)

typedef short int INDEX;

typedef unsigned long ELEMENT;

typedef struct {
        ELEMENT           e[MAXLONG];
}  FIELD2N;
```

eliptic.h Página 1/1

```
/******   eliptic.h   *****/
/*************************************************************
*                                                            *
*        These are structures used to create elliptic curve  *
*   points and parameters.  "form" is a just a fast way to check *
*   if a2 == 0.                                              *
*                  form              equation                *
*                   0                y^2 + xy = x^3 + a_6    *
*                   1                y^2 + xy = x^3 + a_2*x^2 + a_6 *
*                                                            *
*************************************************************/

typedef struct
{
        INDEX   form;
        FIELD2N  a2;
        FIELD2N  a6;
} CURVE;

/*  coordinates for a point  */

typedef struct
{
        FIELD2N  x;
        FIELD2N  y;
} POINT;
```

```
/*  poly.h  */

/*  This header is required for the polynomial math section.
        The purpose is to define intermediate structure sizes
        for use with the multiply and divide routines.
        Place include header after field2n.h
*/

#define DBLBITS             2*NUMBITS
#define DBLWORD             (DBLBITS/WORDSIZE)
#define DBLSHIFT            (DBLBITS%WORDSIZE)
#define MAXDBL              (DBLWORD+1)

#define DERIVMASK           0x55555555

#define DBLLOOP(i)          for(i=0; i<MAXDBL; i++)

typedef struct {
        ELEMENT             e[MAXDBL];
} DBLFIELD;

 * prototypes */

void rot_left();
void rot_right();
void mul_shift();
void null();
void copy();
void poly_mul_partial();
void div_shift();
INDEX log_2();
INDEX degFreeof();
void poly_div();
void poly_inv();
void sngltodbl();
void dbltosngl();
void dblnull();
void poly_gcd();
void mul_x_mod();
INDEX irreducible();
INDEX poly_matrix_invert();
INDEX init_poly_math();
void poly_mul();
INDEX poly_quadratic();
void poly_fofx();
void matrix_print();
void poly_embed();
void poly_esum();
void poly_edbl();
void poly_esub();
void copy_point ();
void poly_elptic_mul();
void print_field();
void print_dbl();
void print_point();
void print_curve();
void poly_gf2();
void Mother();
void random_field();
void rand_curve();
void rand_point();
void DH_gen_send_key();
void DH_key_share();
void send_elgamal();
void receive_elgamal();
void ECKGP();
void authen_secret();
```



7.

polymain.c Página 1/9

```
/*  This is the main test driver for polynomial math routines*/

#include "field2n.h"
#include "poly.h"

/*  Define irreducible polynomial for field here.  Use table, or evaluate your
        own, but the math won't work if it' isn't really a prime polynomial.
        NOTE: the value you pick must be consistent with NUMBITS.
*/

FIELD2N poly_prime = {0x08000000,0x00000000,0x00000000,0x00000000,0x00000000,
    0x00000000,0x00000000,0x00000000,0x000010a1}; /* 283 */
/*FIELD2N  poly_prime = {0x08000000,0x00000000,0x00000000,0x00000000,0x000000B1}
;  *155*/
/*FIELD2N  poly_prime = {0x08000000,0x00000000,0x00000000,0x40000000,0x00000001}
;  /*155*/
/*FIELD2N poly_prime = {0x00000040,0x00000000,0x00000000,0x02000000,0x00000001};
 /*134*/
/*FIELD2N poly_prime = {0x00000008,0x00000000,0x00000000,0x00000000,0x0000010D};
 /*131*/
/*FIELD2N poly_prime = {0x00000004,0x00000000,0x00000000,0x00000000,0x00000009};
 /*130*/
/*FIELD2N poly_prime = {0x00000002,0x00000000,0x00000000,0x00000000,0x00000021};
 /*129*/
/*FIELD2N poly_prime = {0x00000001,0x00000000,0x00000000,0x00000000,0x00000087};
 /*128*/
/*FIELD2N poly_prime = {0x00008000,0x00000000,0x00000000,0x00000401};  /* 111*/
/*FIELD2N  poly_prime = {0x20000000,0x00000000,0x00000005};      /*93*/
/*FIELD2N  poly_prime = {0x0000800,0x00000000,0x0000004b};*/     /*75*/
/*FIELD2N  poly_prime = {0x00000020,0x00000000,0x00000065};*/    /*69*/
/*FIELD2N  poly_prime = {0x00000002,0x00000000,0x0000001b};      /*65*/
/*FIELD2N  poly_prime = {0x00000001,0x00000000,0x0000001b};*/    /*64*/
/*FIELD2N poly_prime = {0x80000000,0x00000001};*/ /*63*/
/*FIELD2N poly_prime = {0x00000002,0x00002001};  /*33*/
/*FIELD2N poly_prime = {0x00000001,0x0000000c5}; /*32*/
/*FIELD2N poly_prime = {0x80000009}; /*31*/
/*FIELD2N poly_prime = {0x800021};*/ /* 23*/
/*FIELD2N poly_prime = {0x100009};*/ /* 20*/
/*FIELD2N poly_prime = {0x20021};*/ /* 17*/
/*FIELD2N poly_prime =  {0x1002B};*/ /* 16*/
/*FIELD2N poly_prime = {0x8003};  */ /* 15*/
/*FIELD2N poly_prime = {0x89};*/ /* 7*/
/*FIELD2N poly_prime = {0x43}; /*6*/
/*FIELD2N poly_prime = {0x29}; /*5*/
/*FIELD2N poly_prime = {0x13};*/ /*4*/

/*  rotation routines copied from normal basis.  Useful
        in general for high level operations.
*/

void rot_left(a)
        FIELD2N *a;
{
        INDEX i;
        ELEMENT bit,temp;

        bit = (a->e[0] & UPRBIT) ? 1L : 0L;
        for (i=NUMWORD; i>=0; i--) {
                temp = (a->e[i] & MSB) ? 1L : 0L;
                a->e[i] = ( a->e[i] << 1) | bit;
                bit = temp;
        }
        a->e[0] &= UPRMASK;
}

void rot_right(a)
        FIELD2N *a;
```

```
{
        INDEX i;
        ELEMENT bit,temp;

        bit = (a->e[NUMWORD] & 1) ? UPRBIT : 0L;
SUMLOOP(i) {
        temp = ( a->e[i] >> 1)  | bit;
        bit = (a->e[i] & 1) ? MSB : 0L;
        a->e[i] = temp;
}
a->e[0] &= UPRMASK;
}

/*  Shift left one bit used by multiply routine.  Make inline for speed.  */

void mul_shift(a)
        DBLFIELD *a;
{
        ELEMENT *eptr, temp, bit;       /* eptr points to one ELEMENT at a time
 */
        INDEX   i;

        eptr = &a->e[DBLWORD];          /* point to end, note: bigendian process
ing  */
        bit = 0;                                        /*  initial carry bit is
 clear */
        DBLLOOP (i)
{
        temp = (*eptr << 1) | bit;              /* compute result as temporary
*/
        bit = (*eptr & MSB) ? 1L : 0L;  /* get carry bit from shift  */
        *eptr-- = temp;                                 /* save new result  */
}
}

/*  null out a FIELD2N variable.  Make inline for speed.  */

void null(a)
        FIELD2N *a;
{
        INDEX i;

        SUMLOOP(i) a->e[i] = 0L;
}

void dblnull(a)
        DBLFIELD *a;
{
        INDEX i;

        DBLLOOP(i) a->e[i] = 0L;
}

/*  copy one FIELD2N variable to another.  Make inline for speed.  */

void copy(from, to)
        FIELD2N *from, *to;
{
        INDEX   i;

        SUMLOOP(i) to->e[i] = from->e[i];
}

/*  copy a FIELD2N variable into a DBLFIELD variable */

void sngltodbl(from, to)
        FIELD2N *from;
```

7.

polymain.c Página 3/9

```
        DBLFIELD *to;
{
        INDEX i;

        dblnull (to);
        SUMLOOP(i) to->e[DBLWORD - NUMWORD + i] = from->e[i];
}

/*  copy the bottom portion of DBLFIELD to FIELD2N variable  */

void dbltosngl(from, to)
        FIELD2N *to;
        DBLFIELD *from;
{
        INDEX i;

        SUMLOOP(i) to->e[i] = from->e[DBLWORD - NUMWORD + i];
}

/*  Simple polynomial multiply.
        Enter with 2 single FIELD2N variables to multiply.
        Result is double length DBLFIELD variable.
        user supplys all storage space, only pointers used here.
*/

void poly_mul_partial(a, b, c)
        FIELD2N *a, *b;
        DBLFIELD *c;
{
        INDEX i, bit_count, word;
        ELEMENT mask;
        DBLFIELD B;

/*  clear all bits in result  */

        dblnull(c);

/*  initialize local copy of b so we can shift it  */

        sngltodbl(b, &B);

/*  for every bit in 'a' that is set, add present B to result  */

        mask = 1;
        for (bit_count=0; bit_count<NUMBITS; bit_count++)
        {
                word = NUMWORD - bit_count/WORDSIZE;
                if (mask & a->e[word])
                {
                        DBLLOOP(i) c->e[i] ^= B.e[i];
                }
                mul_shift( &B);                 /* multiply copy of b by x  */
                mask <<= 1;                     /* shift mask bit up  */
                if (!mask) mask = 1;    /* when it goes to zero, reset to 1  */
        }
}

/*  Shift right routine for polynomial divide.  Convert to inline for speed.
        Enter with pointer to DBLFIELD variable.
        shifts whole thing right one bit;
*/

void div_shift(a)
        DBLFIELD *a;
{
        ELEMENT *eptr, temp, bit;
        INDEX   i;
```

```
        eptr = (ELEMENT*) &a->e[0];
        bit = 0;
        DBLLOOP (i)
{
                temp = (*eptr>>1) | bit;                /*  same as shift left but  */
        bit = (*eptr & 1) ? MSB : 0L;    /*  carry bit goes off other end  */
        *eptr++ = temp;
}
}

/*  binary search for most significant bit within word */

INDEX log_2 (x)
        ELEMENT x;
{
        ELEMENT ebit, bitsave, bitmask;
        INDEX   k, lg2;

        lg2 = 0;
        bitsave = x;                                    /* grab bits we're inter
ested in.  */
        k = WORDSIZE/2;                                 /* first see if msb is i
n top half */
        bitmask = -1L<<k;                               /* of all bits  */
        while (k)
        {
                ebit = bitsave & bitmask;       /* did we hit a bit?  */
                if (ebit)                                       /* yes  */
                {
                        lg2 += k;                               /* increment deg
ree by minimum possible offset */
                        bitsave = ebit;                 /* and zero out non usef
ul bits  */
                }
        /*  The following two lines are slick and tricky.  We are binary searchi
ng,
                so k is divided by 2.  It's also the base amount we can add at a
ny given
                search point.  The mask continuously searches for upper blocks b
eing set.
                Once found, the remaining lower blocks are zeroed, to make sure
we don't
                use bits which aren't the most significant.
        */
                k /= 2;
                bitmask ^= (bitmask >> k);
        }
        return( lg2);
}

/*  find most significant bit in multiple ELEMENT array.
        Enter with pointer to array (t) and number of elements (dim).
        Returns degree of polynomial == position count of most signicant bit set
*/

INDEX degreeof(t, dim)
        ELEMENT *t;
        INDEX   dim;
{
        INDEX   degree, k;

/*  This is generic routine for arbitrary length arrays, use ELEMENT
        pointer to find first non-zero ELEMENT.  We will add degree from some ba
se,
        so initial base is at least one WORDSIZE smaller than maximum possible.
*/
```

```
        degree = dim * WORDSIZE;
        for (k=0; k<dim; k++)                                   /* search for a
non-zero ELEMENT  */
        {
                if (*t) break;
                degree -= WORDSIZE;
                t++;
        }

/* for inversion routine, need to know when all bits zero.  */

        if (!*t) return(-1);

/*  find most significant bit in this element */

        degree += log_2( *t);
        return(degree);
}

/*  division of two polynomials.  Major use is reduction modulo
        irreducible polynomials.
        Enter with pointers to top, bottom, quotient and remainder.
          Assumes top is DBLFIELD and all other arrays are
          FIELD2N variables.
        Returns with top destroyed, and the following satisfied:
          top = quotient * bottom + remainder.
*/

void poly_div(top, bottom, quotient, remainder)
        DBLFIELD *top;
        FIELD2N  *bottom, *quotient, *remainder;
{
        INDEX    deg_top;
        INDEX    deg_bot;
        INDEX    deg_quot;
        INDEX    bit_count;
        INDEX    i;
        INDEX    equot;
        ELEMENT topbit, *tptr;
        DBLFIELD         shift;

/*  Step 1: find degree of top and bottom polynomials.  */

        deg_top = degreeof( top, DBLWORD);
        deg_bot = degreeof( bottom, NUMWORD);

/*  prepare for null return and check for null quotient */

        null (quotient);
        if (deg_top < deg_bot)
        {
                dbltosngl(top, remainder);
                return;
        }

/*  Step 2: shift bottom to align with top.  Note that there
                are much more efficient ways to do this. */

        deg_quot = deg_top - deg_bot;
        bit_count = deg_quot + 1;
        sngltodbl(bottom, &shift);
        for (i = 0; i<deg_quot; i++)
                mul_shift(&shift);

/* Step 3: create bit mask to check if msb of top is set  */
```

```
        topbit = 1L << (deg_top % WORDSIZE);
        tptr = (ELEMENT*) top + DBLWORD - deg_top/WORDSIZE;

/*  for each possible quotient bit, see if bottom can be subtracted
        (added, it's modulo 2!) from top.  If it can, set that bit in
        quotient.  If it can't, clear that bit in quotient.  Shift bottom
        right once and keep going for total bit_count.
*/

        while (bit_count)
        {

/*  Step 4: determine one bit of the quotient.  */

                if (*tptr & topbit)             /* is bit set in top?  */
                {
                        DBLLOOP (i)                     /* yes, subtract shift f
rom top  */
                                top->e[i] ^= shift.e[i];

        /* find word and bit in quotient to be set  */

                        equot =         NUMWORD - deg_quot/WORDSIZE;
                        quotient->e[equot] |= 1L << (deg_quot % WORDSIZE);
                }

/*  Step 5: advance to the next quotient bit and perform the necessary
                        shifts.  */

                bit_count--;                    /*  number of bits is one more
*/
                deg_quot--;                             /*  than polynomial degr
ee  */
                div_shift(&shift);              /*  divide by 2  */
                topbit >>= 1;                   /*  move mask over one bit also
 */
                if (!topbit)
                {
                        topbit = MSB;           /*  reset mask bit to next word
 */
                        tptr++;                         /*  when it goes to zero
 */
                }
        }

/*  Step 6: return the remainder in FIELD2N size  */

        dbltosngl (top, remainder);
}

/*  Polynomial multiplication modulo poly_prime.  */

poly_mul(a, b, c)
        FIELD2N *a, *b, *c;
{
        DBLFIELD temp;
        FIELD2N  dummy;

        poly_mul_partial(a, b, &temp);
        poly_div(&temp, &poly_prime, &dummy, c);
}

/*  Polynomial inversion routine.  Computes inverse of polynomial field element
        assuming irreducible polynomial "poly_prime" defines the field.
*/

void poly_inv(a, inverse)
```

```
        FIELD2N *a, *inverse;
{
        FIELD2N pk, pk1, pk2;
        FIELD2N rk, rk1;
        FIELD2N qk, qk1, qk2;
        INDEX   i;
        DBLFIELD rk2;

/*  initialize remainder, quotient and product terms  */

        sngltodbl (&poly_prime, &rk2);
        copy( a, &rk1);
        null( &pk2);
        null( &pk1);
        pk1.e[NUMWORD] = 1L;
        null( &qk2);
        null( &qk1);
        qk1.e[NUMWORD] = 1L;

/*  compute quotient and remainder for Euclid's algorithm.
    when degree of remainder is < 0, there is no remainder, and we're done.
    At that point, pk is the answer.
*/
        null( &pk);
        pk.e[NUMWORD] = 1L;
        poly_div(&rk2, &rk1, &qk, &rk);

        while (degreeof(&rk, NUMWORD) >= 0)
        {
                poly_mul_partial( &qk, &pk1, &rk2);
                SUMLOOP(i) pk.e[i] = rk2.e[i+DBLWORD-NUMWORD] ^ pk2.e[i];

/*  set up variables for next loop */

                sngltodbl(&rk1, &rk2);
                copy( &rk, &rk1);
                copy( &qk1, &qk2);
                copy( &qk, &qk1);
                copy( &pk1, &pk2);
                copy( &pk, &pk1);
                poly_div( &rk2, &rk1, &qk, &rk);
        }
        copy( &pk, inverse);                    /* copy answer to output  */
}

/*  polynomial greatest common divisor routine.  Same as Euclid's algorithm.  */

void poly_gcd( u, v, gcd)
        FIELD2N *u, *v, *gcd;
{
        DBLFIELD        top;
        FIELD2N         r, dummy, temp;

        sngltodbl( u, &top);
        copy( v, &r);

        while( degreeof( &r, NUMWORD) >= 0)
        {
                poly_div( &top, &r, &dummy, &temp);
                sngltodbl( &r, &top);
                copy( &temp, &r);
        }
        dbltosngl( &top, gcd);
}

/*  multiply a polynomial u by x modulo polynomial v.  Useful in
        several places.  Enter with u and v, returns polynomial w.
```

```
        w can equal to u, so this will work in place.
*/

void mul_x_mod( u, v, w)
        FIELD2N *u, *v, *w;
{
        DBLFIELD mulx;
        INDEX    i, deg_v;

        deg_v = degreeof(v, NUMWORD);
        sngltodbl( u, &mulx);
        mul_shift( &mulx);  /*  multiply u by x  */
        dbltosngl( &mulx, w);
        if (w->e[ NUMWORD - (deg_v/WORDSIZE)] & ( 1L << (deg_v % WORDSIZE)))
                SUMLOOP (i) w->e[i] ^= v->e[i];
}

/*  check to see if input polynomial is irreducible.
        Returns 1 if irreducible, 0 if not.

        This method explained by Richard Pinch.  The idea is to use
        gcd algorithm to test if x^2^r - x has input v as a factor
        for all r <= degree(v)/2.  If there is any common factor,
        then v is not irreducible.  This works because x^2^r - x
        contains all possible irreducible factors of degree r, and
        because we only need to find the smallest degree such factor
        if one exists.
*/

INDEX irreducible( v)
        FIELD2N *v;
{
        FIELD2N         vprm, gcd, x2r, x2rx, temp;
        FIELD2N         sqr_x[NUMBITS+1];
        INDEX           i, r, deg_v, k;

 *  check that gcd(v, v') = 1.  If not, then v not irreducible.  */

        SUMLOOP(i) vprm.e[i] = (v->e[i] >> 1) & DERIVMASK;
        poly_gcd( v, &vprm, &gcd);
        if (gcd.e[NUMWORD] > 1) return(0);
        for (i=0; i<NUMWORD; i++)  if (gcd.e[i]) return(0);

/*  find maximum power we have to deal with  */

        deg_v = degreeof(v, NUMWORD);

/*  create a vector table of powers of x^2k mod v.
        this will be used to compute square of x^2^r
*/

        null (&sqr_x[0]);
        sqr_x[0].e[NUMWORD] = 1;
        for (i=1; i<= deg_v; i++)
        {
                mul_x_mod( &sqr_x[i-1], v, &temp);
                mul_x_mod( &temp, v, &sqr_x[i]);
        }

/*  check that gcd( x^2^r - x, v) == 1 for all r <= degreeof(v)/2.
        set x^2^r = x for r = 0 to initialize.
*/
        null( &x2r);
        x2r.e[NUMWORD] = 2;
        for ( r=1; r <= deg_v/2; r++)
        {
/*  square x^2^r mod v.  We do this by seeing that s^2(x) = s(x^2).
```

7.

polymain.c Página 9/9

```
        for each bit j set in x^2^(r-1) add in x^2j mod v to find x^2^r.
*/
                null( &x2rx);
                for (i=0; i <= deg_v; i++)
                {
                        if ( x2r.e[NUMWORD - (i/WORDSIZE)] & ( 1L << (i%WORDSIZE
) ) )
                                SUMLOOP(k) x2rx.e[k] ^= sqr_x[i].e[k];
                }

/*  save new value of x^2^r mod v and compute x^2^r - x mod v */

                copy( &x2rx, &x2r);
                x2rx.e[NUMWORD] ^= 2;

/*  is gcd( x^2^r - x, v) == 1?  if not, exit with negative result  */

                poly_gcd( &x2rx, v, &gcd);
                if (gcd.e[NUMWORD] > 1) return (0);
                for (i=0; i<NUMWORD; i++) if ( gcd.e[i]) return (0);
        }

/*  passed all tests, provably irreducible  */

        return (1);
}
```

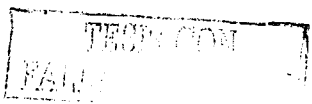
```
/*****************************************************************
*                                                                *
*                periferal functions useful for setting up polynomial     *
*       math but not called routinely.                           *
*                                                                *
*                                                                *
*****************************************************************/

#include "field2n.h"
#include "poly.h"

/*  for given irreducible polynomial solve g^3 = g + 1.
        Useful for GF(8^n) Koblitz curves.
        Trick is to lineraize equation into form g^4 + g^2 + g = 0.
        This is linear because (a + b)^(2^n) = a^(2^n) + b^(2^n).
        There are four solutions: one is zero, two are found as
        independent vectors and the third is their sum.

        Input:  pointers to irreducble polynomial and g[3]
        Output: g[0], g[1], and g[3] = g[0] + g[1] filled in
*/

void poly_gf8( prime, g)
        FIELD2N   *prime, g[3];
{
        FIELD2N         gamma_matrix[NUMBITS], solve[NUMBITS];
        FIELD2N         power_table[4*NUMBITS], temp;
        INDEX           row, column, i, j, found, vector;
        ELEMENT         tobit, frombit, bit, null_check;

/*  step 1:  compute all powers of x modulo prime polynomial
        with simple function */

        null( &power_table[0]);
        power_table[0].e[NUMWORD] = 1L;
        for (row = 1; row < 4*NUMBITS; row++)
                mul_x_mod( &power_table[row-1], prime, &power_table[row]);

/*  step 2:  sum powers of rows to create g^4 + g^2 + g
        coefficients matrix.
*/
        for( row=0; row < NUMBITS; row++)
        {
                copy( &power_table[row], &gamma_matrix[row]);
                SUMLOOP (i) gamma_matrix[row].e[i] ^=
                        power_table[row<<1].e[i] ^ power_table[row<<2].e[i];
        }

/*  step 3:  transpose matrix and work with single powers of x */

        for ( row=0; row < NUMBITS; row++)  null( &solve[row]);
        for ( row=0; row < NUMBITS; row++)
        {
                bit = 1L << (row % WORDSIZE);
                i = NUMWORD - (row/WORDSIZE);
                frombit = 1;
                j = NUMWORD;
                for ( column = 0; column < NUMBITS; column++)
                {
                        if (gamma_matrix[row].e[j] & frombit)
                                solve[column].e[i] |= bit;
                        frombit <<= 1;
                        if ( !frombit)
                        {
                                frombit = 1;
```

7.

poly_func.c Página 2/3

```
                        j--;
                }
        }
}

/*  step 4: lower diagonalize solve matrix.
        2 columns will go null, search for null rows instead
        of diagnalizing.  */

        vector = 0;             /*  set up solution space  */
        null( &g[0]);
        null( &g[1]);

        for (column = NUMBITS - 1; column > 0; column--)
        {
                bit = 1L << (column % WORDSIZE);
                i = NUMWORD - column/WORDSIZE;
                if ( !(bit & solve[column].e[i]) )
                {
                /*  go look for a bit set in this column  */

                        found = 0;
                        for (row = column - 1; row >= 0; row--)
                        {
                                if ( bit & solve[row].e[i])
                                {
                                        copy ( &solve[row], &temp);
                                        copy ( &solve[column], &solve[row]);
                                        copy ( &temp, &solve[column]);
                                        found = 1;
                                        break;
                                }
                        }
                }
                else found = 1;
        /*  and eliminate any set bits below it  */

                if (found)
                {
                        for ( row = column-1; row >=0; row--)
                        {
                                if (solve[row].e[i] & bit)
                                        SUMLOOP (j) solve[row].e[j] ^= solve[col
umn].e[j];
                        }
                }
                else
                {
        /*  if null column, see if we have null row to match it.  */

                        null_check = 0;
                        SUMLOOP (j) null_check |= solve[column].e[j];
                        if ( null_check)
                        {
                        /*  search for a null row to put here  */

                                for ( row = column-1; row >=0; row--)
                                {
                                        null_check = 0;
                                        SUMLOOP (j) null_check |= solve[row].e[j
];
                                        if ( !null_check)
                                        {
                                                copy ( &solve[row], &temp);
                                                copy ( &solve[column], &solve[ro
w]);
                                                copy ( &temp, &solve[column]);
```

```
                                        break;
                                }
                        }
                }

        /*  mark this vector with column bit, and use next vector */

                        g[vector].e[i] |= bit;
                        vector++;
                }
        }

/*  last row may be null and not marked.  if INDEX vector < 2,
        set g[1] = 1
*/
        if (vector < 2) g[1].e[NUMWORD] = 1;

/*  find two solution vectors by back solving matrix.  use bits
        previoiusly solved to find bits not yet solved.  Initial vectors
        come from assuming g0 and g1 are variables.
*/

        for ( row=1; row<NUMBITS; row++)
        {
                tobit = 1L << (row % WORDSIZE);
                j = NUMWORD - (row/WORDSIZE);

                /*  check to see if diagonal bit is set  */

                if (solve[row].e[j] & tobit)
                {
                        for (column = row-1; column >= 0; column--)
                        {
                                frombit = 1L << (column % WORDSIZE);
                                i = NUMWORD - column/WORDSIZE;
                                if ( solve[row].e[i] & frombit)
                                {
                                        if (g[0].e[i] & frombit) g[0].e[j] ^= to
bit;
                                        if (g[1].e[i] & frombit) g[1].e[j] ^= to
bit;
                                }
                        }
                }
        }

/*  last step:  g[2] = g[1] + g[0]  */

        SUMLOOP (i) g[2].e[i] = g[0].e[i] ^ g[1].e[i];
}
```



elliptic_poly.c Página 1/13

```
/*  elliptic_poly.c  */
/*****************************************************************************
*****
*
*
*                    Polynomial version of elliptic curve functions.  essentially the
 same as
*  optimal normal basis but uses polynomial functions.  Biggest difference is th
at
*  once poly_prime is picked (see polymain.c) you need to compute the "T matrix"
 to
*  help embed data onto a curve by solving a quadratic equation.
*
*
*                                                          Author = mike rosing
*                                                           date  = June 27, 1997
*
*
*****************************************************************************
****/

#include <stdio.h>
#include "field2n.h"
#include "poly.h"
#include "elliptic.h"

/*  Global parameters for polynomial math.  poly_prime is the irreducible polyno
mial
        for the Galois field arithmetic.  Tmatrix is used to solve quadratic equ
ations
        and Smatrix is used to compute square roots.
*/

extern  FIELD2N poly_prime;
FIELD2N Tmatrix[NUMBITS], Smatrix[NUMBITS], Trace_Vector;
INDEX   Null_Row;

/*  Subroutine to invert a binary matrix.  The method is brute force but simple.
        Start with a diagonal matrix in I, and for each row operation in mat_in
that
        eliminates non diagonal terms, do the same in I.  The result is that mat
_in
        = 1 and I = 1/mat_in (transpose).
        If the input transpose is set to any value, we then have to transpose I
back
        into mat_out to get the correct result.  If transpose is zero, I is copi
ed to
        mat_out and the value of error is returned to indicate any null row.

        Returns 0 if matrix invertible, row number of zero column if not inverti
ble.
*/

INDEX poly_matrix_invert( mat_in, mat_out)
        FIELD2N mat_in[NUMBITS], mat_out[NUMBITS];
{
        INDEX   row, col, i, j, rowdex, found, error;
        ELEMENT src_mask, dst_mask;
        FIELD2N dummy, Imatrix[NUMBITS];

/*  Create Imatrix as diagonalized  */
```

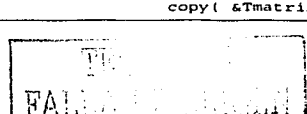
```
        for ( row=0; row<NUMBITS; row++)
        {
                null( &Imatrix[row]);
                Imatrix[row].e[NUMWORD - row/WORDSIZE] = 1L << (row%WORDSIZE);
        }
        error = 0;  /* hope this is return value  */

/*  Diagonalize input matrix.  Eliminate all other bits in each column.
        First find a column bit that is set, and swap with diagonal
        row if needed.
*/

        for ( row = 0; row < NUMBITS; row ++)
        {
                rowdex = NUMWORD - row/WORDSIZE;
                src_mask = 1L << (row % WORDSIZE);

/*  find a row of input matrix which has col = row bit set.
        First see if we get lucky, then do search.
*/
                found = 0;
                if ( !(mat_in[row].e[rowdex] & src_mask))
                {
                        for (j = row+1; j<NUMBITS; j++)
                        {
                                if ( mat_in[j].e[rowdex] & src_mask)  /*  found
one, swap rows  */
                                {
                                        copy( &Imatrix[j], &dummy);
                                        copy( &Imatrix[row], &Imatrix[j]);
                                        copy( &dummy, &Imatrix[row]);
                                        copy( &mat_in[j], &dummy);
                                        copy( &mat_in[row], &mat_in[j]);
                                        copy( &dummy, &mat_in[row]);
                                        found = 1;
                                        break;
                                }
                        }
                }
                else found = 1;

/*  eliminate all other terms in this column  */

                if (found)
                {
                        for ( i=0; i<NUMBITS; i++)
                        {
                                if ( i == row) continue;
                                if ( mat_in[i].e[rowdex] & src_mask)
                                {
                                        SUMLOOP(j)

        Imatrix[i].e[j] ^= Imatrix[row].e[j];
        mat_in[i].e[j] ^= mat_in[row].e[j];
}
}
                        /*  end column eliminate  */
else error = row;
                        /*  end diagonalization  */

/*  Next step is to transpose diagonalized matrix.  This completes the
        inversion process.  Clear Tmatrix to begin with. */

for (row=0; row<NUMBITS; row++) null(&mat_out[row]);
```

7.

eliptic_poly.c

```
for (col = 0; col<NUMBITS; col++)
{
        j = NUMWORD - col/WORDSIZE;
        src_mask = 1L << (col % WORDSIZE);
        for (row = 0; row<NUMBITS; row++)
        {
                if (Imatrix[row].e[j] & src_mask)
                {
                        i = NUMWORD - row/WORDSIZE;
                        dst_mask = 1L << (row  % WORDSIZE);
                        mat_out[col].e[i] |= dst_mask;
                }
        }
}
return(error);
}

/*  initialize polynomial math for elliptic curve calculations.

        Creates Tmatrix and Smatrix to help solve quadratic equations as
        well as Trace_Vector.  The Smatrix is only invoked to compute the
        square root of a polynomial modulo poly_prime.  The Tmatrix is a
        set of basis vectors which are summed assuming a solution to the
        quadratic exists.  The Trace_Vector is used to determine if the
        solution exists  If an error occurs in creating the Smatrix the
        routine returns the row it could not diagonalize, otherwise the
        routine returns 0 for no errors.
*/

INDEX init_poly_math()
{
        INDEX   i, j, error, k, sum, row, rowdex, found, nulldex;
        ELEMENT src_mask;
        FIELD2N x, c, dummy;
        FIELD2N Trace[2*NUMBITS], Tz[NUMBITS], T2[NUMBITS];

/*  Create Trace_Vector for computing the trace in polynomial basis.
        Given any input c = c_m*x^m + ... + c_1*x + c_0  the Trace_Vector
        will mask off and sum the correct coefficients of a_i.  The
        following method was suggested by Richard Pinch (rgep@dpmms.cam.ac.uk)
*/

/*  step 1 Build a list of powers of x modulo poly_prime from 0 to 2n-2 */

        null( &Trace[0] );
        Trace[0].e[NUMWORD] = 1L;
        for (i=1; i<2*NUMBITS-1; i++)
                mul_x_mod( &Trace[i-1], &poly_prime, &Trace[i]);

/*  step 2: Sum diagonals of nxn matrix using each row as a starting
        point.  Each sum amounts to the trace vector coefficient for that
        power of x.
*/

        null( &Trace_Vector);
        for (i=0; i<NUMBITS; i++)
        {
                sum = 0;
                j = NUMWORD;
                src_mask = 1;
                for ( k=i; k<i+NUMBITS; k++)
                {
                        if ( Trace[k].e[j] & src_mask) sum ^= 1;
                        src_mask <<= 1;
                        if (!src_mask)
                        {
                                src_mask = 1;
```

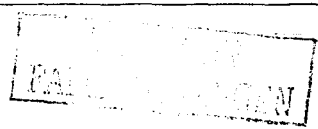
```
                        j--;
                }
        }
        if (sum) Trace_Vector.e[NUMWORD - i/WORDSIZE] |= 1L << (i % WORD
SIZE);
}

/*  Next compute Tz = z^2 + z matrix.  We already have every power of x in
        the Trace matrix, so each row of Tz is just two rows from Trace.  Do
        partial inversion to get Tmatrix.
*/
        for ( i=0; i<NUMBITS; i++)
        {
                SUMLOOP(j)
{
        Tz[i].e[j] = Trace[i].e[j] ^ Trace[i<<1].e[j];
        T2[i].e[j] = Trace[i<<1].e[j];
}
}

/*  Invert T2 matrix to get special square root matrix (called Smatrix)  */

if (error = poly_matrix_invert( T2, Smatrix))
{
        printf("Can not invert square root matrix.  Null row = %d\n", error);
        return(error);
}

/*  Create a set of basis vectors for use in finding roots of z^2 + z = c
        This is similar to the matrix inversion, but there is no transpose and
        the row order is different.  Exactly how this works is not clear to me,
        but Prof. Pinch gets it and it works.
*/

/*  Create Tmatrix as diagonalized  */

for ( row=0; row<NUMBITS; row++)
{
        null( &Tmatrix[row]);
        Tmatrix[row].e[NUMWORD - row/WORDSIZE] = 1L << ( row % WORDSIZE);
}

/*  Semi diagonalize input matrix.
        First find a column bit that is set, and swap with diagonal
        row if needed.  Then eliminate all bits below it.
*/

Null_Row = 0;
nulldex = 0;
for ( row = 0; row < NUMBITS; row++)
{
        rowdex = NUMWORD - row/WORDSIZE;
        src_mask = 1L << (row % WORDSIZE);

/*  Find a row of input matrix which has col = row bit set.
        First see if we get lucky, then do search.
*/
        found = 0;
        if ( !(Tz[row].e[rowdex] & src_mask))
        {
                for (j = row+1; j<NUMBITS; j++)
                {
                        if ( Tz[j].e[rowdex] & src_mask)  /*  found one, swap ro
ws  */
                        {
                                copy( &Tmatrix[j], &dummy);
                                copy( &Tmatrix[row], &Tmatrix[j]);
```



7.


```
                                copy( &dummy, &Tmatrix[row]);
                                copy( &Tz[j], &dummy);
                                copy( &Tz[row], &Tz[j]);
                                copy( &dummy, &Tz[row]);
                                found = 1;
                        /*  keep track of original null row */
                                if (row == nulldex) nulldex = j;
                                break;
                        }
                }
        }
        else found = 1;

/*  eliminate all other terms below diagonal in this column  */

        if (found)
        {
                for ( i=row+1; i<NUMBITS; i++)
                {
                        if ( Tz[i].e[rowdex] & src_mask)
                        {
                                SUMLOOP(j)
{
        Tmatrix[i].e[j] ^= Tmatrix[row].e[j];
        Tz[i].e[j] ^= Tz[row].e[j];
}
}
}
                        /*  end column eliminate  */
else
{
                /*  mark null row and swap position with original  */
        Null_Row = row;
        copy( &Tmatrix[nulldex], &dummy);
        copy( &Tmatrix[row], &Tmatrix[nulldex]);
        copy( &dummy, &Tmatrix[row]);
        copy( &Tz[nulldex], &dummy);
        copy( &Tz[row], &Tz[nulldex]);
        copy( &dummy, &Tz[row]);
}
}
                                /*  end diagonalization  */

/*  finally eliminate all other terms above diagonal except for
        Null_Row.  Result is a set of basis vectors which converts
        c to z and z^2 + z = c.
*/

for ( row = NUMBITS-1; row > 0; row--)
{
        if (row == Null_Row) continue;

        rowdex = NUMWORD - row/WORDSIZE;
        src_mask = 1L << (row % WORDSIZE);

        for (i = row-1; i>=0; i--)
        {
                if (Tz[i].e[rowdex] & src_mask)
                {
                        SUMLOOP(j)
{
        Tmatrix[i].e[j] ^= Tmatrix[row].e[j];
        Tz[i].e[j] ^= Tz[row].e[j];
}
}
}
                        /*  end column eliminate  */
return(0);
}
```

```
/*****************************************************************
*                                                                *
*       solve a quadratic equation over an irreducible polynomial field.  *
*   Enter with parameters for the equation y^2 + xy + f = 0, where y *
*   is the unknown, x is usually the x coordinate of a point and         *
*   f = f(x) of a curve.                                         *
*                                                                *
*       Assumes init_poly_math already run with Trace_Vector, Null_Row   *
*   and Tmatrix filled out.                                      *
*                                                                *
*       computes c = f/x^2 and tests if the trace condition is met.  If  *
*   not returns an error code of 1.  Otherwise it means we can solve *
*   the reduced equation z^2 + z + c = 0.  The change of variable is *
*   y = xz.                                                      *
*                                                                *
*       returns error code zero, y[0] = xz, y[1] = y[0] + x      *
*                                                                *
*****************************************************************/

INDEX poly_quadratic( x, f, y)
        FIELD2N *x, *f, y[2];
{
        FIELD2N c, z, dummy;
        FIELD2N test1, test2;
        INDEX   i, j, k;
        ELEMENT sum, mask;

  *  first check to see if x is zero  */

        sum = 0;
        SUMLOOP(i) sum |= x->e[i];

        if (sum)
        {
  *  compute c = x^-2 * f  */

                copy (x, &c);
                poly_mul(&c, x, &dummy);                /*  get x^2  */
                poly_inv(&dummy, &z);
                poly_mul( f, &z, &c);

  *  verify that trace of c = 0.  If not, no solution
        is possible.  */

                sum = 0;
                SUMLOOP(i) sum ^= c.e[i] & Trace_Vector.e[i];
                mask = ~0;
                for (i = WORDSIZE/2; i > 0; i >>= 1)
                {
                        mask >>= i;
                        sum = ((sum & mask) ^ (sum >> i));
                }

        /* if last bit is set, there is no solution to equation.
        This eliminates half the points in the field, which might
        make sense to a mathematician.
        */
                if (sum)
                {
```



```
                        null( &y[0]);
                        null( &y[1]);
                        return(1);
                }

/*  clear out null row bit.  that part of matrix will not work.  */

                j = NUMWORD - Null_Row / WORDSIZE;
                c.e[j] &= ~( 1L << (Null_Row % WORDSIZE));

/*  for every bit set in c, add that row of Tmatrix to solution.  */

                null( &z);
                mask = 1;
                j = NUMWORD;
                for (i=0; i<NUMBITS; i++)
                {
                        if ( c.e[j] & mask)
                                SUMLOOP(k) z.e[k] ^= Tmatrix[i].e[k];
                        mask <<= 1;
                        if ( !mask)
                        {
                                mask = 1;
                                j--;
                        }
                }

/*  compute final solution using input parameters.  */

                poly_mul( x, &z, &y[0]);
                SUMLOOP(i) y[1].e[i] = y[0].e[i] ^ x->e[i];
                return(0);
        }
        else
        {
/*  x input was zero.  Return y = square root of f. Process
        involves ANDing each row of Smatrix with f and summing
        all bits to find coefficient for that power of x */

                null( &z);
                for (j = 0; j<NUMBITS; j++)
                {
                        sum = 0;
                        SUMLOOP(i) sum ^= Smatrix[j].e[i] & f->e[i];
                        if (sum)
                        {
                                mask = -1L;
                                for (i = WORDSIZE/2; i > 0; i >>= 1)
                                {
                                        mask >>= i;
                                        sum = ((sum & mask) ^ (sum >> i));
                                }
                        }
                        if (sum) z.e[NUMWORD - j/WORDSIZE] |= (1L << j%WORDSIZE)
;
                }
                copy( &z, &y[0]);
                copy( &z, &y[1]);
                return(0);
        }
}

/*  print field matrix.  an array of FIELD2N of length NUMBITS is assumed.  The
        bits are printed out as a 2D matrix with an extra space every 5 characte
rs.
*/
```

```
void matrix_print( name, matrix, file)
        FIELD2N matrix[NUMBITS];
        char *name;
        FILE *file;
{
        INDEX i,j;

        fprintf(file,"%s\n",name);
        for (i = NUMBITS-1; i>=0; i--)
        {
                if (i%5==0) fprintf(file,"\n");  /* extra line every 5 rows  */
                for (j = NUMBITS-1; j>=0; j--)
                {
                        if ( matrix[i].e[NUMWORD - j/WORDSIZE] & (1L<<(j%WORDSIZ
E)))
                                fprintf(file,"1");
                        else fprintf(file,"0");
                        if (j%5==0) fprintf(file," ");  /* extra space every 5 c
haracters  */
                }
                fprintf(file,"\n");
        }
        fprintf(file,"\n");
}

/*  compute f(x) = x^3 + a_2*x^2 + a_6 for non-supersingular elliptic curves */

void poly_fofx( x, curv, f)
        FIELD2N *x, *f;
        CURVE *curv;
{
        FIELD2N         x2, x3;
        INDEX           i;

        copy ( x, &x3);
        poly_mul( x, &x3, &x2);         /*  get x^2  */
        if (curv->form) poly_mul ( &x2, &curv->a2, f);
        else null( f);
        poly_mul( x, &x2, &x3);         /*  get x^3  */
        SUMLOOP (i) f->e[i] ^= ( x3.e[i] ^ curv->a6.e[i] );
}

/*  embed data onto a curve.
        Enter with data, curve, ELEMENT offset to be used as increment, and
        which root (0 or 1).
        Returns with point having data as x and correct y value for curve.
        Will use y[0] for last bit of root clear, y[1] for last bit of root set.
        If ELEMENT offset is out of range, default is 0.
*/

void poly_embed( data, curv, incrmt, root, pnt)
        FIELD2N *data;
        CURVE   *curv;
        INDEX   incrmt, root;
        POINT   *pnt;
{
        FIELD2N         f, y[2];
        INDEX           inc = incrmt;
        INDEX           i;

        if ( (inc < 0) || (inc > NUMWORD) ) inc = 0;
        copy( data, &pnt->x);
        poly_fofx( &pnt->x, curv, &f);
        while (poly_quadratic( &pnt->x, &f, y))
        {
                pnt->x.e[inc]++;
                poly_fofx( &pnt->x, curv, &f);
```

```
        }
        copy ( &y[root&1], &pnt->y);
}

/*****************************************************************************
*                                                                            *
*   Implement elliptic curve point addition for polynomial basis form.       *
*                                                                            *
*  This follows R. Schroeppel, H. Orman, S. O'Mally, "Fast Key Exchange with *
*  Elliptic Curve Systems", CRYPTO '95, TR-95-03, Univ. of Arizona, Comp.    *
*  Science Dept.                                                             *
*****************************************************************************/

void poly_esum (p1, p2, p3, curv)
        POINT   *p1, *p2, *p3;
        CURVE   *curv;
{
        INDEX   i;
        FIELD2N  x1, y1, theta, onex, theta2;
        ELEMENT  check;

/*  check if p1 or p2 is point at infinity  */

        check = 0;
        SUMLOOP(i) check |= p1->x.e[i] | p1->y.e[i];
        if (!check)
        {
                printf("summing point at infinity\n");
                copy_point( p2, p3);
                return;
        }
        check = 0;
        SUMLOOP(i) check |= p2->x.e[i] | p2->y.e[i];
        if (!check)
        {
                printf("summing point at infinity\n");
                copy_point( p1, p3);
                return;
        }

/*  compute theta = (y_1 + y_2)/(x_1 + x_2)  */

        null(&x1);
        null(&y1);
        check = 0;
        SUMLOOP(i)
{
        x1.e[i] = p1->x.e[i] ^ p2->x.e[i];
        y1.e[i] = p1->y.e[i] ^ p2->y.e[i];
        check |= x1.e[i];
}
if (!check)   /* return point at infinity  */
{
        printf("input to elliptic sum has duplicate x values\n");
        null(&p3->x);
        null(&p3->y);
        return;
}
poly_inv( &x1, &onex);
poly_mul( &onex, &y1, &theta);  /*  compute y_1/x_1 = theta */
poly_mul(&theta, &theta, &theta2);   /* then theta^2  */

/*  with theta and theta^2, compute x_3  */

if (curv->form)
        SUMLOOP (i)
        p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ x1.e[i] ^ curv->a2.e[i];
```

```
else
        SUMLOOP (i)
        p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ x1.e[i];

/*  next find y_3  */

SUMLOOP (i) x1.e[i] = p1->x.e[i] ^ p3->x.e[i];
poly_mul( &x1, &theta, &theta2);
SUMLOOP (i) p1->y.e[i] = theta2.e[i] ^ p3->x.e[i] ^ p1->y.e[i];
}

/*  elliptic curve doubling routine for Schroeppel's algorithm over polymomial
    basis.  Enter with p1, p3 as source and destination as well as curv
    to operate on.  Returns p3 = 2*p1.
*/

void poly_edbl (p1, p3, curv)
        POINT *p1, *p3;
        CURVE *curv;
{
        FIELDN  x1, y1, theta, theta2, t1;
        INDEX   i;
        ELEMENT check;

        check = 0;
        SUMLOOP (i) check |= p1->x.e[i];
        if (!check)
        {
                printf("doubling point at infinity\n");
                null(&p3->x);
                null(&p3->y);
                return;
        }

/*  first compute theta = x + y/x  */

        poly_inv( &p1->x, &x1);
        poly_mul( &x1, &p1->y, &y1);
        SUMLOOP (i) theta.e[i] = p1->x.e[i] ^ y1.e[i];

/*  next compute x_3  */

        poly_mul( &theta, &theta, &theta2);
        if(curv->form)
                SUMLOOP (i) p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ curv->a2.e[i
];
        else
                SUMLOOP (i) p3->x.e[i] = theta.e[i] ^ theta2.e[i];

/*  and lastly y_3  */

        theta.e[NUMWORD] ^= 1;                  /*  theta + 1 */
        poly_mul( &theta, &p3->x, &t1);
        poly_mul( &p1->x, &p1->x, &x1);
        SUMLOOP (i) p3->y.e[i] = x1.e[i] ^ t1.e[i];
}

/*  subtract two points on a curve.  just negates p2 and does a sum.
    Returns p3 = p1 - p2 over curv.
*/

void poly_esub (p1, p2, p3, curv)
        POINT   *p1, *p2, *p3;
        CURVE   *curv;
{
        POINT   negp;
        INDEX   i;
```

```
        copy ( &p2->x, &negp.x);
        null (&negp.y);
        SUMLOOP(i) negp.y.e[i] = p2->x.e[i] ^ p2->y.e[i];
        poly_esum (p1, &negp, p3, curv);
}

/*  need to move points around, not just values.  Optimize later.  */

void copy_point (p1, p2)
        POINT *p1, *p2;
{
        copy (&p1->x, &p2->x);
        copy (&p1->y, &p2->y);
}

/*  Routine to compute kP where k is an integer (base 2, not normal basis)
        and P is a point on an elliptic curve.  This routine assumes that K
        is representable in the same bit field as x, y or z values of P.  Since
        the field size determines the largest possible order, this makes sense.
    Enter with: integer k, source point P, curve to compute over (curv)
    Returns with: result point R.

  Reference: Koblitz, "CM-Curves with good Cryptografic Properties",
        Springer-Verlag LNCS #576, p279 (pg 284 really), 1992
*/

void  poly_elptic_mul(k, p, r, curv)
        FIELD2N *k;
        POINT   *p, *r;
        CURVE   *curv;
{
        char            blncd[NUMBITS+1];
        INDEX           bit_count, i;
        ELEMENT         notzero;
        FIELD2N         number;
        POINT           temp;

/*  make sure input multiplier k is not zero.
        Return point at infinity if it is.
*/
        copy( k, &number);
        notzero = 0;
        SUMLOOP (i) notzero |= number.e[i];
        if (!notzero)
        {
                null (&r->x);
                null (&r->y);
                return;
        }

/*  convert integer k (number) to balanced representation.
        Called non-adjacent form in "An Improved Algorithm for
        Arithmetic on a Family of Elliptic Curves", J. Solinas
        CRYPTO '97. This follows algorithm 2 in that paper.
*/
        bit_count = 0;
        while (notzero)
        {
        /*  if number odd, create 1 or -1 from last 2 bits  */

                if ( number.e[NUMWORD] & 1 )
                {
                        blncd[bit_count] = 2 - (number.e[NUMWORD] & 3);

        /*  if -1, then add 1 and propagate carry if needed  */
```

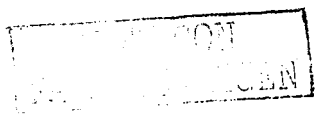
```
                        if ( blncd[bit_count] < 0 )
                        {
                                for (i=NUMWORD; i>=0; i--)
                                {
                                        number.e[i]++;
                                        if (number.e[i]) break;
                                }
                        }
                }
                else
                        blncd[bit_count] = 0;

        /*  divide number by 2, increment bit counter, and see if done  */

                number.e[NUMWORD] &= ~0 << 1;
                rot_right( &number);
                bit_count++;
                notzero = 0;
                SUMLOOP (i) notzero |= number.e[i];
        }

/*  now follow balanced representation and compute kP  */

        bit_count--;
        copy_point(p,r);                        /* first bit always set */
        while (bit_count > 0)
        {
                poly_edbl(r, &temp, curv);
                bit_count--;
                switch (blncd[bit_count])
                {
                        case 1: poly_esum (p, &temp, r, curv);
                                break;
                        case -1: poly_esub (&temp, p, r, curv);
                                break;
                        case 0: copy_point (&temp, r);
                }
        }
}

void print_field( string, x)
        char *string;
        FIELD2N *x;
{
        INDEX i;

        printf("%s\n",string);
        SUMLOOP(i) printf("%8x ",x->e[i]);
                printf("\n");
}

void print_point( string, point)
        char *string;
        POINT *point;
{
        INDEX i;

        printf("%s\n",string);
        printf("x: ");
        SUMLOOP(i) printf("%8x ",point->x.e[i]);
                printf("\n");
                printf("y: ");
                SUMLOOP(i) printf("%8x ",point->y.e[i]);
                printf("\n");
}

void print_curve( string, curv)
```

7.

elliptic_poly.c Página 13/13

```
        char *string;
        CURVE *curv;
{
        INDEX i;

        printf("%s\n",string);
        printf("form: %d\n",curv->form);
        if (curv->form)
        {
                printf("a2: ");
                SUMLOOP(i) printf("%lx ",curv->a2.e[i]);
                        printf("\n");
        }
        printf("a6: ");
        SUMLOOP(i) printf("%lx ",curv->a6.e[i]);
                printf("\n");
}
```

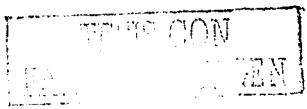
```
/*****************************************************************************
*
*
*                                                                           *
*               Routines to implement protocols, Diffie-Hellman, Massey-Omura an
d
*       ElGamal for elliptic curve analogs.  Data transfer routines not included
*
*                                                                           *
*****************************************************************************
/

#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include "field2n.h"
#include "poly.h"
#include "eliptic.h"

/*  random seed is accessable to everyone, not best way, but functional.  */

unsigned long random_seed;
extern  FIELD2N poly_prime;

/*  below is from Mother code, till end of mother.  Above is all my fault.  */

#include <string.h>

static short mother1[10];
static short mother2[10];
static short mStart=1;

#define m16Long 65536L                          /* 2^16 */
#define m16Mask 0xFFFF          /* mask for lower 16 bits */
#define m15Mask 0x7FFF                  /* mask for lower 15 bits */
#define m31Mask 0x7FFFFFFF      /* mask for 31 bits */
#define m32Double  4294967295.0   /* 2^32-1 */

 * Mother *************************************************************
        George Marsaglia's The mother of all random number generators
                producing uniformly distributed pseudo random 32 bit values with
                period about 2^250.

        The arrays mother1 and mother2 store carry values in their
                first element, and random 16 bit numbers in elements 1 to 8.
                These random numbers are moved to elements 2 to 9 and a new
                carry and number are generated and placed in elements 0 and 1.
        The arrays mother1 and mother2 are filled with random 16 bit values
                on first call of Mother by another generator.  mStart is the swi
tch.

        Returns:
        A 32 bit random number is obtained by combining the output of the
                two generators and returned in *pSeed.  It is also scaled by
                2^32-1 and returned as a double between 0 and 1

        SEED:
        The inital value of *pSeed may be any long value

        Bob Wheeler 8/8/94

        removed double return since I don't need it.  mgr
*

void Mother(unsigned long *pSeed)
```



7.


```
{
        unsigned long  number,
        number1,
        number2;
        short          n,
        *p;
        unsigned short sNumber;

                /* Initialize mother1 with 9 random values the first time */
        if (mStart) {
                sNumber= *pSeed&m16Mask;    /* The low 16 bits */
                number= *pSeed&m31Mask;    /* Only want 31 bits */

                p=mother1;
                for (n=18;n--;) {
                        number=30903*sNumber+(number>>16);
                                /* One line multiply-with-cary */
                        *p++=sNumber=number&m16Mask;
                        if (n==9)
                                p=mother2;
                }
                /* make cary 15 bits */
                mother1[0]&=m15Mask;
                mother2[0]&=m15Mask;
                mStart=0;
        }

                /* Move elements 1 to 8 to 2 to 9 */
        memmove(mother1+2,mother1+1,8*sizeof(short));
        memmove(mother2+2,mother2+1,8*sizeof(short));

                /* Put the carry values in number1 */
        number1=mother1[0];
        number2=mother2[0];

                /* Form the linear combinations */

        number1+=1941*mother1[2]+1860*mother1[3]+1812*mother1[4]+1776*mother1[5]
+
                1492*mother1[6]+1215*mother1[7]+1066*mother1[8]+12013*mother1[9]
;

        number2+=1111*mother2[2]+2222*mother2[3]+3333*mother2[4]+4444*mother2[5]
+
                5555*mother2[6]+6666*mother2[7]+7777*mother2[8]+9272*mother2[9];

                /* Save the high bits of number1 as the new carry */
        mother1[0]=number1/m16Long;
        mother2[0]=number2/m16Long;
                /* Put the low bits of number1 into mother1[1] */
        mother1[1]=m16Mask&number1;
        mother2[1]=m16Mask&number2;

                /* Combine the two 16 bit random numbers into one 32 bit */
        *pSeed=(((long)mother1[1])<<16)+(long)mother2[1];

                /* Return a double value between 0 and 1
        return ((double)*pSeed)/m32Double;   */
}

/*  Generate a random bit pattern which fits in a FIELD2N size variable.
        Calls Mother as many times as needed to create the value.
*/

void random_field( value)
        FIELD2N *value;
{
```

```
        INDEX   i;

        SUMLOOP(i)
{
        Mother( &random_seed);
        value->e[i] = random_seed;
}
value->e[0] &= UPRMASK;
}

/*  generate a random curve for a given field size.
        Enter with pointer to storage space for returned curve.
        Returns with curve.form = 0,  curve.a2 = 0 and curve.a6
        as a random bit pattern.   This is for the equation

                y^2 + xy = x^3 + a_2x^2 + a_6
*/

void rand_curve ( curv)
        CURVE *curv;
{
        curv->form = 0;
        random_field( &curv->a6);
        null( &curv->a2);
}

/*  generate a random point on a given curve.
        Enter with pointer to curve and one pointer
        to storage space for returned point.   Returns
        one of solutions to above equation. Negate point
        to get other solution.
*/

void rand_point( point, curve)
        POINT   *point;
        CURVE   *curve;
{
        FIELD2N rf;

        random_field( &rf);
        poly_embed( &rf, curve, NUMWORD, rf.e[NUMWORD]&1, point);
}

/*  Compute a Diffie-Hellman key exchange.

        First routine computes senders public key.
        Enter with public point Base_point which sits on public curve E and
        senders private key my_private.
        Returns public key point My_public = my_private*Base_point to be sent
        to other side.
*/

void DH_gen_send_key( Base_point, E, my_private, My_public)
        POINT *Base_point, *My_public;
        CURVE *E;
        FIELD2N *my_private;
{
        poly_elptic_mul( my_private, Base_point, My_public, E);
}

/*      Second routine computes shared secret that is same for sender and
        receiver.
        Enter with public point Base_point which sits on public curve E along wi
th
        senders public key their_public and receivers private key k.
        Returns shared_secret as x component of kP
*/
```



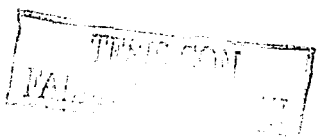
```
void DH_key_share(Base_point, E, their_public, my_private, shared_secret)
        POINT *Base_point, *their_public;
        CURVE *E;
        FIELD2N *my_private, *shared_secret;
{
        POINT   temp;

        poly_elptic_mul( my_private, their_public, &temp, E);
        copy (&temp.x, shared_secret);
}

/*  Send data to another person using ElGamal protocol. Send Hidden_data and
        Random_point to other side. */

void send_elgamal(
        Base_point, Base_curve,
        Their_public, raw_data,
        Hidden_data, Random_point)
        FIELD2N *raw_data;
        POINT   *Base_point, *Their_public, *Hidden_data, *Random_point;
        CURVE   *Base_curve;
{
        FIELD2N random_value;
        POINT   hidden_point, raw_point;

/*  create random point to help hide the data  */

        random_field (&random_value);
        poly_elptic_mul (&random_value, Base_point, Random_point, Base_curve);

/*  embed raw data onto the chosen curve,  Assume raw data is contained in
        least significant ELEMENTs of the field variable and we won't hurt anyth
ing
        using the most significant to operate on.  Uses the first root for y val
ue.
*/

        poly_embed( raw_data, Base_curve, 0, 0, &raw_point);

/*  Create the hiding value using the other person's public key  */

        poly_elptic_mul( &random_value, Their_public, &hidden_point, Base_curve)
;
        poly_esum( &hidden_point, &raw_point, Hidden_data, Base_curve);
}

/*  Recieve data from another person using ElGamal protocol. We get
        Hidden_data and Random_point and output raw_data. */

void receive_elgamal(
        Base_point, Base_curve,
        my_private, Hidden_data, Random_point,
        raw_data)
        FIELD2N *my_private, *raw_data;
        POINT   *Base_point, *Hidden_data, *Random_point;
        CURVE   *Base_curve;
{
        POINT   hidden_point, raw_point;

/*  compute hidden point using my private key and the random point  */

        poly_elptic_mul( my_private, Random_point, &hidden_point, Base_curve);
        poly_esub( Hidden_data, &hidden_point, &raw_point, Base_curve);
        copy(&raw_point.x, raw_data);
}
```

```
/*  MQV method to establish shared secret.
        Enter with other servers permenent (other_Q) and
        ephemeral (other_R) keys,
        this servers permenent key (skey, pkey) and
        ephemeral (dkey, dpoint) keys.
        Returns shared secret point W.  Only W.x is useful as key,
        and further checks may be necessary to confirm link established.
*/
CURVE   Base_curve;

void authen_secret( d, Q, k, R, other_Q, other_R, W)
        FIELD2N *d, *k;
        POINT  *Q, *R, *other_Q, *other_R, *W;
{
        POINT S, T, U;

/*  compute U = R' + x'a'Q' from other sides data */

        poly_elptic_mul(&other_Q->x, other_Q, &S, &Base_curve);
        poly_elptic_mul(&other_R->x, &S, &T, &Base_curve);
        poly_esum(other_R, &T, &U, &Base_curve);

/* compute (k + xad)U the hard way.  Need modulo math routines to make this
        quicker, but no need to know point order this way.
*/

        poly_elptic_mul(d, &U, &S, &Base_curve);
        poly_elptic_mul(&Q->x, &S, &T, &Base_curve);
        poly_elptic_mul(&R->x, &T, &S, &Base_curve);
        poly_elptic_mul(k, &U, &T, &Base_curve);
        poly_esum(&S, &T, W, &Base_curve);
}

/*  Generate a key pair, a random value plus a point.
        This was called ECKGP for Elliptic Curve Key Generation
        Primitive in an early draft of IEEE P1363.

        Input:  Base point on curve (pnt, crv)

        Output: secret key k and random point R
*/

void ECKGP( pnt, crv, k, R)
        POINT *pnt, *R;
        CURVE *crv;
        FIELD2N *k;
{
        random_field( k);
        poly_elptic_mul( k, pnt, R, crv);
}


void print_dbl( string, field)
        char *string;
        DBLFIELD *field;
{
        INDEX i;

        printf("%s: ", string);
        DBLLOOP(i) printf("%8x ", field->e[i]);
                printf("\n");
}

main()
{
        FIELD2N private1, private2, key1, key2;
        CURVE rnd_crv;
```

7.


```
POINT Base, P1, P2;
INDEX   error;

/* Estructuras para calcular el tiempo */
struct timeval inic, fin, temps, inic2, fin2, temps2,
inic3, fin3, temps3, inic4, fin4, temps4;

random_seed = 0x932b15fe;

gettimeofday(&inic,NULL);/*inicio de contador (1) */

if (!irreducible(&poly_prime)) return(0);
print_field("poly_prime = ", &poly_prime);

if (error = init_poly_math())
{
        printf("Can't initialize S matrix, row = %d\n", error);
        return(-1);
}

printf("create Base curve and point\n\n");

rand_curve(&rnd_crv);
print_curve("random curve", &rnd_crv);
rand_point(&Base, &rnd_crv);
print_point("Base point", &Base);

gettimeofday(&fin,NULL); /* fin de contador (1) */

printf(" ncreate each sides private key\n\n");

gettimeofday(&inic2,NULL);/*inicio de contador (2) */

random_field(&private1);
print_field("Side 1 secret:", &private1);
random_field(&private2);
print_field("Side 2 secret:", &private2);

gettimeofday(&fin2,NULL); /* fin de contador (2) */

printf(" nGenerate each sides public key\n\n");

gettimeofday(&inic3,NULL); /* inicio de contador (3) */

DH_gen_send_key( &Base, &rnd_crv, &private1, &P1);
print_point("Side 1 public key", &P1);
DH_gen_send_key( &Base, &rnd_crv, &private2, &P2);
print_point("Side 2 public key", &P2);

gettimeofday(&fin3,NULL); /* fin de contador (3) */

printf(" nShow that each side gets the same shared secret\n\n");

gettimeofday(&inic4,NULL); /* inicio de contador (4) */

DH_key_share( &Base, &rnd_crv, &P2, &private1, &key1);
print_field("key 1 is", &key1);
DH_key_share( &Base, &rnd_crv, &P1, &private2, &key2);
print_field("key 2 is", &key2);

gettimeofday(&fin4,NULL); /* fin de contador (4)*/

/* Calculo del tiempo transcurrido */
temps.tv_sec = fin.tv_sec - inic.tv_sec;
temps.tv_usec = fin.tv_usec - inic.tv_usec;
temps2.tv_sec = fin2.tv_sec - inic2.tv_sec;
temps2.tv_usec = fin2.tv_usec - inic2.tv_usec;
```

```
        temps3.tv_sec = fin3.tv_sec - inic3.tv_sec;
        temps3.tv_usec = fin3.tv_usec - inic3.tv_usec;
        temps4.tv_sec = fin4.tv_sec - inic4.tv_sec;
        temps4.tv_usec = fin4.tv_usec - inic4.tv_usec;

        printf(" n(1)  ->%ld segs. %ld usgs. Creando curva y punto aleatorio", temps.tv_sec, temps.t
v_usec);
        printf(" n(2)  ->%ld segs. %ld usgs. Generando llaves privadas de ambos lados", temps2.tv_sec,
 temps2.tv_usec);
        printf(" n(3)  ->%ld segs. %ld usgs. Generando llaves publicas de ambos lados", temps3.tv_sec,
 temps3.tv_usec);
        printf(" n(4)  ->%ld segs. %ld usgs. Mostrando el mensaje de ambos lados", temps4.tv_sec, te
mps4.tv_usec);
}
```



7.

Bibliografía

[A99] ANSI X9.62, *Public Key Cryptography for the Financial services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, 1999.

[A99-2] ANSI X9.63, *Public Key Cryptography for the Financial services Industry: Elliptic Curve Key Agreement and Key Transport Protocols*, working draft, August 1999.

[AM93] A. Atkin and F. Morain, "Elliptic curves and primality proving". *Mathematics of Computation*, Vol. 61, pp. 29-68. (1993).

[AW1993] S. Arno and F.S. Wheeler. *"Signed digit representations of minimal Hamming weight". IEEE Trans. Comp.*, 42, pp. 1007-1010. 1993.

[Bea96] Dan Beauregard, *"Efficient Algorithms for Implementing Elliptic Curve Public-Key Schemes"; A Thesis submitted to the Faculty of the Worcester Polytechnic Institute.* 1996.

[Bel00] Gabriel Belingueres, *"Introducción a los Criptosistemas de Curva Elíptica"*, Whitepaper, Chacabuco - Buenos Aires - Argentina. http://www.geocities.com/belingueres, (2000).

[BGMW93] E. Brickell, D. Gordon, K. McKurley and D. Wilson, *"Fast exponentiation with precomputation", Advances in Cryptology - Eurocrypt '92*, LNCS 658, pp. 200-207. 1993.

[Borst97] J. Borst, *"Public Key Cryptosystems using Elliptic Curves"*, Master's Thesis; TECHNISCHE UNIVERSITEIT EINDHOVEN. Belgium, (1997).

[BSS1999] Ian Blake, Gadiel Seroussi and Nigel Smart, *"Elliptic Curves in Cryptography", London Mathematical Society, Lecture Note Series 265.* Cambridge University Press. 1999.

[CC87] D. Chudnovsky and G. Chudnovsky, *"Sequences of numbers generated by addition in formal groups and new primality and factoring tests", Advances in Applied Mathematics*, 7, pp. 385-434. 1987.

[DH76] W. Diffie y M. E. Hellman., *"New directions in cryptography", IEEE, Transactions of Information Theory*, IT-22 (1976), pp. 664-654.

[EGS5] T. ElGamal. *"A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". IEEE Transactions on Information Theory, vol. IT-31*, pp. 469-472. 1985.

[FGH01] A. Fúster Sabater, D. de la Guía Martínez, L. Hernández Encinas, F. Montoya Vitini y J. Muñoz Masqué. *"Técnicas Criptográficas de protección de datos"*. RA - MA Editorial, 2a. Ed, 2001.

[GHS00] P. Gaudry, F. Hess and N. Smart, *"Constructive and destructive facets of Weil descent on elliptic curves"*, preprint, January 2000.

[GS99] S. Galbraith and N. Smart, *"A criptography aplication of Weil descent", Codes and Cryptography*, LNCS 1746, Springer-Verlag. 1999.

[H00] D. Hankerson, J. López Hernandez, and A. Menezes, *"Software Implementation of Elliptic Curve Cryptography Over Binary Fields"*. Whitepaper, (2000).

[IEEE00] IEEE P1363, *Standard Specifications for Public-Key Cryptography*. 2000.

[ISO98] ISO/IEC 14888-3, *Information Technology - Security Techniques - Digital Signatures with Appendix - Part 3: Certificate Based-Mechanisms*. 1998.

[ISO99] ISO/IEC 15946, *Information Technology - Security Techniques - Cryptographic Techniques Based on Elliptic Curves,* Committee Draft (CD), 1999.

[JM89] J. Jedwab and C.J. Mitchell. *"Minimum weight modified signed-digit representations and fast exponentiation". Electronic Letters.* 25, pp. 1171-1172, 1989.

[K87] N. Koblitz., "Elliptic curve cryptosystems", *Mathematics of Computation,* Vol. 48, no. 177 (1987), pp. 203-209.

[K99] N. Koblitz., "Algebraic Aspects of Cryptography", *Algorithms and Computation in Mathematics,* Vol. 3, Springer-Verlag, (1999).

[Kaz92] O. Kazarin, *"Use of properties of elliptic curves". Automatic Control and Computer Sciences.* 26(5), pp. 19-26. 1992.

[Knuth98] D. Knuth, *"The Art of Computer Programming - Seminumerical Algorithms",* Addison-Wesley, 3rd edition. 1998.

[Kob90] N. Koblitz. *"Constructing elliptic curve cryptosystems in characteristic 2". In.* C90, pp. 156-167. 1990.

[KT93] K. Koyama and Y. Tsuruoka, *"Speeding up elliptic cryptosystems by using a signed binary window method", Advances in Cryptology - Crypto '92,* LNCS 740; pp. 345-357. 1993.

[L87] H. W. Lenstra Jr., "Factoring integers with elliptic curves", *Annals of Mathematics,* Vol. 126 (1987), pp. 649-673.

[LD00] J. López and R. Dahab, *"Improved algorithms for elliptic curve arithmetic in GF*(2^n)*", Selected Areas in Cryptography - SAC '98.* LNCS 1556, pp. 201-212. 1999.

[LL94] C. Lim and P. Lee, *"More flexible exponentiation with precomputation", Advances in Cryptology - Crypto '94,* LNCS 839. pp. 95-107. 1994.

[LN86] R. Lidl and H. Niederreiter, *"Introduction to finite fields and their applications"*, Cambridge University Press, 1986, Revised Edition 1994.

[M85] V. S. Miller, "Use of elliptic curves in cryptography", *Advances in Cryptology Proc. Crypto'85, LNCS 218*, H. C. Williams, Ed., Springer-Verlag, 1985, pp. 417-426.

[MG71] J.L. Massey and O. N. Garcia. *"Error correcting codes in computer arithmetic". In Anvances in Information Systems Science*, J. L. Tou editor, pp. 273-326. Plenum, New York, 1971.

[MIRACL] *"M.I.R.A.C.L. Users Manual"*. Shamus Software Ltd. Ireland. January 2002.

[MO97] A. Menezes, P. van Oorschot and S. Vanstone. *"Handbook of Applied Cryptography"*, CRC Press, 1997.

[MOV93] A. Menezes, T. Okamoto and S. Vanstone. *"Reducing elliptic curve logarithms to a finite field". IEEE Trans. Info. Theory*, 39, pp. 1639-1646. 1993.

[MQV95] A. Menezes, M. Qu, and S. Vanstone. *"Elliptic curve systems". Proposed IEEE p1363 Standard*, pp. 1-42. 1995.

[MTH97] A. Miyaji, T. Ono and H. Cohen, *"Efficient elliptic curve exponentiation", Proceedings of ICICS '97*, LNCS 1334; pp. 282-290. 1997.

[MV93] A. Menezes and S. Vanstone. *"Elliptic curve cryptosystems and their implementation". Journal of Cryptography*, 6. pp. 209-224. 1993.

[NIST99] National Institute of Standards and Technology, *"RECOMENNDED ELLIPTIC CURVES FOR FEDERAL GOVERNMENT USE"*, http://csrc.nist.gov/csrc/fedstandards.html. July 1999.

[NIST00] National Institute of Standards and Technology, *Digital Signature Standard*. FIPS Publication 186-2, February 2000.

[Reit60] G. Reitwiesner. *"Binary arithmetic". Adv. in Comp.*, 1, 231-308, 1960.

[Ros99] Michael Rosing. *"Implementing Elliptic Curve Cryptography"*, Manning Publications Co. 1999.

[SCH96] Bruce Schneier, *"Applied Cryptography: Protocols, Algorithms and Source Code in C"*, John Wiley & Sons, Inc., (1996).

[Schoof95] R. Schoof. *"Counting point on elliptics curves over finite fields". J. Théorie des Nombres de Bordeaux*, 7. pp. 219-254. 1995.

[Sem98] I. Semaev, *"Evaluatión of discrete logarithms in a group of p-torsion points of an an elliptic curve in characteristic p", Mathematics of Computation*, 67, pp. 353-356. 1998.

[Sha71] D. Shanks. *"Class number, a theory of factorization , and genera". In Proc. Symp. Pure Math. vol. 20*, pp. 415-440. AMS, 1971.

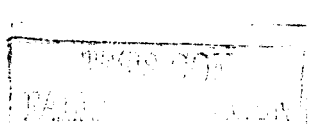
[Sim92] G. J. Simmons (Ed.), *"Contemporary Cryptology, The Science of Information Integrity"*, IEEE Press, 1992.

[Smart99] N. Smart. *"The discrete logarithm problem on elliptic curves of trace one", Journal of Cryptology*, 12, pp. 193-196. 1999.

[So00] J. Solinas, *"Eficient arithmetic on Koblitz curves", Designs, Codes and Cryptography*, 19, pp. 195-249. (2000),

[SOOS95] R. Schroeppel, H. Orman, S. O'Malley and *"Fast key exchange with elliptic curve systems", Advances in Cryptology - Crypto '95*, LNCS 963. pp. 43-56. 1995.

[WZ98] Michael J. Wiener and Robert J. Zuccherato. *"Faster Attacks on Elliptic Curve Cryptosystems". Entrust Technologies.* Draft. 1998.



México. D. F. , a 14 de Enero del 2003.

ING. LEOPOLDO SILVA GUTIERREZ.
DIRECTOR GENERAL DE ADMINISTRACIÓN ESCOLAR.
P R E S E N T E .

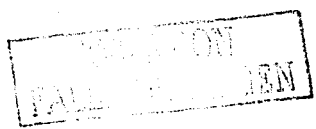
He leído detenidamente la tesis que presenta **Alejandro Andrade Alvarez**, titulada **"Curvas Elípticas sobre Campos Finitos de Característica Dos y sus aplicaciones en Criptografía de Clave Pública"**. Considero que dicha tesis satisface plenamente los requisitos necesarios para optar por el grado de Maestro en Ciencias.

Sin más por el momento, aprovecho la ocasión para enviarle un cordial saludo.

A t e n t a m e n t e .

B. Khapchenko

Dr. Vladislav Khartchenko.



154

México, D. F. , a 16 de Enero del 2003.

ING. LEOPOLDO SILVA GUTIERREZ.
DIRECTOR GENERAL DE ADMINISTRACIÓN ESCOLAR.
P R E S E N T E .

He leído detenidamente la tesis que presenta **Alejandro Andrade Alvarez**, titulada **"Curvas Elípticas sobre Campos Finitos de Característica Dos y sus aplicaciones en Criptografía de Clave Pública"**. Considero que dicha tesis satisface plenamente los requisitos necesarios para optar por el grado de Maestro en Ciencias.

Sin más por el momento, aprovecho la ocasión para enviarle un cordial saludo.

A T E N T A M E N T E.

DR. FRANCISCO JAVIER GARCÍA UGALDE

México, D. F. , a 14 de Enero del 2003.

ING. LEOPOLDO SILVA GUTIERREZ.
DIRECTOR GENERAL DE ADMINISTRACIÓN ESCOLAR.
P R E S E N T E .

He leído detenidamente la tesis que presenta **Alejandro Andrade Alvarez**, titulada **"Curvas Elípticas sobre Campos Finitos de Característica Dos y sus aplicaciones en Criptografía de Clave Pública"**. Considero que dicha tesis satisface plenamente los requisitos necesarios para optar por el grado de Maestro en Ciencias.

Sin más por el momento, aprovecho la ocasión para enviarle un cordial saludo.

A t e n t a m e n t e .

Dr. Vladimir Tchijov.

México, D. F. , a 22 de Enero del 2003.

ING. LEOPOLDO SILVA GUTIERREZ.
DIRECTOR GENERAL DE ADMINISTRACIÓN ESCOLAR.
P R E S E N T E .

He leído detenidamente la tesis que presenta **Alejandro Andrade Alvarez**, titulada **"Curvas Elípticas sobre Campos Finitos de Característica Dos y sus aplicaciones en Criptografía de Clave Pública"**. Considero que dicha tesis satisface plenamente los requisitos necesarios para optar por el grado de Maestro en Ciencias.

Sin más por el momento, aprovecho la ocasión para enviarle un cordial saludo.

A T E N T A M E N T E.

DR. ENRIQUE DALTABUIT GODAS

157

México, D. F. , a 14 de Enero del 2003.

ING. LEOPOLDO SILVA GUTIERREZ.
DIRECTOR GENERAL DE ADMINISTRACIÓN ESCOLAR.
P R E S E N T E .

He leído detenidamente la tesis que presenta **Alejandro Andrade Alvarez**, titulada **"Curvas Elípticas sobre Campos Finitos de Característica Dos y sus aplicaciones en Criptografía de Clave Pública"**. Considero que dicha tesis satisface plenamente los requisitos necesarios para optar por el grado de Maestro en Ciencias.

Sin más por el momento, aprovecho la ocasión para enviarle un cordial saludo.

A T E N T A M E N T E.

M. EN C. GUILLERMO ARNULFO VAZQUEZ COUTIÑO.