**Universidad Autónoma Metropolitana**
**Unidad Azcapotzalco**

División de Ciencias Básicas e Ingeniería

Proyecto Terminal en Ingeniería en Computación

# Implementación en software-hardware de aritmética sobre campos finitos binarios $\mathbb{F}_{2^m}$ en curvas elípticas para aplicaciones criptográficas de llave pública

Proyecto que presenta:
**Arturo Alvarez Gaona**

para obtener el título de:
**Ingeniero en Computación**

Directores de Proyecto:

**Dr. Pedro Ricardo López Bautista** **M. en C. Oscar Alvarado Nava**

**México, D.F.** **9 de abril de 2012**

# Resumen

En la actualidad la criptografía es considerada como un aspecto muy importante para el resguardo de la información. El incremento de la capacidad de cómputo ha permitido resolver algoritmos matemáticos en un tiempo subexponencial por lo que es necesario aumentar el tamaño de llaves de cifrado, lo cual implica un mayor costo de procedimiento.
Las curvas elípticas propuestas en un inicio por Neil Koblitz y Victor Miller cuya seguridad descansa básicamente en el problema del logaritmo discreto ofrecen la creación de llaves públicas con la misma seguridad que otros métodos, pero con la ventaja de usar llaves mucho más pequeñas, permitiendo su implementación en diversas aplicaciones cuyos recursos computacionalmente son limitados.

En el presente proyecto se presenta el uso de las curvas elípticas sobre campos finitos binarios $\mathbb{F}_{2^m}$ y la manera en que estas pueden ser utilizadas para construir criptosistemas de llave pública. Como primera parte se detallan aspectos matemáticos importantes en que se basan las curvas elípticas así como una breve explicación del problema intratable del logaritmo discreto sobre curvas elípticas. Posteriormente se muestra el desarrollo en software de la generación de llaves públicas así como una implementación utilizando la multiplicación escalar para el intercambio de llaves mediante el protocolo de Diffie-Hellman elíptico. También se realiza la implementación de la multiplicación y elevar al cuadrado sobre campos finitos binarios $\mathbb{F}_{2^{233}}$ en el lenguaje de descripción de circuitos VHDL mostrando los resultados obtenidos mediante la simulación y el análisis de los resultados.

# Índice general

**Resumen** iii

**Lista de Figuras** vii

**1. Introducción** 1
1.1. Motivaciones . . . 3
1.2. Objetivos . . . 3
1.2.1. Objetivos particulares . . . 3
1.3. Organización del proyecto . . . 4

**2. Criptografía de llave pública** 5
2.1. Algoritmos asimétricos de cifrado . . . 6
2.2. Aplicaciones de algoritmos asimétricos . . . 6
2.2.1. Firma Digital . . . 6
2.3. El algoritmo RSA . . . 8
2.3.1. Generación de llaves . . . 8
2.3.2. Esquema de firma digital y verificación . . . 8
2.4. Algoritmo de encriptación basado en el Problema del Logaritmo Discreto 9

**3. Aspectos matemáticos** 11
3.1. Definiciones y teoremas . . . 12

**4. Curvas Elípticas** 17
4.1. Definición de curvas elípticas . . . 18
4.1.1. Curvas elípticas sobre el campo de números reales $\mathbb{R}$. . . 18
4.1.2. Definición de curvas elípticas sobre $\mathbb{F}_{2^m}$ . . . 21
4.1.3. Curvas elípticas sobre $\mathbb{F}_{2^m}$ . . . 21
4.2. Ley de $E(k)$ . . . 23
4.3. Curvas elípticas de Koblitz . . . 24
4.4. Problema del logaritmo discreto en curvas elípticas . . . 25
4.5. Protocolo Diffie-Hellman Elíptico ECCDH . . . 25
4.6. Ataques a criptosistemas . . . 26
4.6.1. Ataques al IFP . . . 26
4.6.2. Ataques al DLP . . . 26

4.6.3. Ataques al ECDLP . . . . . 27
4.7. Eficiencia de los criptosistemas con curvas elípticas . . . . . 27
4.8. Criptografía con curvas elípticas . . . . . 28
4.8.1. Parámetros de curvas elípticas sobre $\mathbb{F}_{2^m}$ . . . . . 28
4.8.2. Pares de llaves para curvas elípticas . . . . . 29
4.8.3. Validación de llaves públicas . . . . . 30

**5. Implementación en software de la aritmética para $E(\mathbb{F}_{2^m})$ 31**
5.1. Preparación de bibliotecas . . . . . 32
5.1.1. Instalación de la biblioteca GMP . . . . . 32
5.1.2. Instalación de la biblioteca LiDIA . . . . . 32
5.2. Implementación del primer programa . . . . . 33
5.2.1. Aritmética GF2n . . . . . 33
5.2.2. Menú principal . . . . . 34
5.3. Ejecución del primer programa . . . . . 40
5.3.1. Definir m, a y b . . . . . 41
5.3.2. Impresión de puntos existentes sobre la curva elíptica . . . . . 41
5.3.3. Valores de la curva elíptica en $\mathbb{F}_{2^m}$ . . . . . 42
5.3.4. Operaciones aritméticas sobre curvas elípticas en $\mathbb{F}_{2^m}$ . . . . . 42
5.3.5. Multiplicación escalar . . . . . 44
5.4. Resultados . . . . . 44
5.4.1. Relación entre el orden de la curva elíptica y el orden del punto 44
5.4.2. Pruebas para la multiplicación escalar sobre $E(\mathbb{F}_{2^m})$ . . . . . 45
5.5. Implementación del segundo programa . . . . . 47
5.5.1. Implementación del servidor . . . . . 48
5.5.2. Implementación de clientes . . . . . 49
5.6. Ejecución del segundo programa . . . . . 51

**6. Aritmética en Hardware sobre campos finitos $\mathbb{F}_{2^m}$ 55**
6.0.1. Descripción de hardware . . . . . 56
6.0.2. Representación base polinomial . . . . . 56
6.0.3. Multiplicación sobre campos finitos $\mathbb{F}_{2^m}$ . . . . . 57
6.0.4. Algoritmo de Karatsuba-Ofman . . . . . 57
6.0.5. Reducción modular . . . . . 58
6.0.6. Elevar al cuadrado . . . . . 60
6.0.7. Implementación de multiplicación en campos finitos $\mathbb{F}_{2^m}$ . . . 61
6.0.8. Implementación de cuadrados en campo finito $\mathbb{F}_{2^{233}}$ . . . . . 71

**7. Conclusiones 75**
7.0.9. Trabajos a futuro . . . . . 77

**A. Códigos fuentes 79**

**B. Códigos fuentes 91**

**C. Códigos fuentes 101**

**D. Códigos fuentes 105**

# Índice de figuras


2.1. Transmisión de un mensaje utilizando algoritmos asimétricos. . . . . 7

4.1. $y^2 = x^3 - 11x + 4$ . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.2. $y^2 = x^3 - 3x + 2$ con $\Delta = 0$ . . . . . . . . . . . . . . . . . . . . . 19
4.3. Suma de puntos sobre la curva $y^2 = x^3 - 11x + 4$ . . . . . . . . . . . 19
4.4. Suma del punto $P$ con su inverso $-P$ sobre la curva $y^2 = x^3 - 2x + 12$ 20
4.5. Suma del punto $P$ consigo mismo de la curva $y^2 = x^3 - 2x + 12$ . . . 20
4.6. Suma del punto $P$ consigo mismo con la coordenada $y = 0$ de la curva $y^2 = x^3 + 4x + -7$ . . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.7. Gráfica del campo finito binario $\mathbb{F}_{2^4}$ con $a = z^3 + 1$ y $b = z^3 + z + 1$ . 23
4.8. Protocolo Diffie-Hellman Eliptico ECCDH, para intercambio de claves. 26

5.1. Menú principal del programa. . . . . . . . . . . . . . . . . . . . . . . 41
5.2. Definir m, a y b en la ecuación $y^2 = x^3 + ax^2 + b$ sobre $\mathbb{F}_{2^m}$. . . . . . 41
5.3. Impresión de puntos sobre la curva $y^2 = x^3 + 8x^2 + 9$ para campos finitos $\mathbb{F}_{2^m}$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.4. Gráfica de puntos sobre la curva $y^2 = x^3 + 8x^2 + 9$ para campos finitos $\mathbb{F}_{2^m}$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.5. Valores definidos sobre la curva elíptica. . . . . . . . . . . . . . . . . 42
5.6. Generación de puntos al azar. . . . . . . . . . . . . . . . . . . . . . . 43
5.7. Suma de puntos sobre la curva. . . . . . . . . . . . . . . . . . . . . . 43
5.8. Duplicar punto sobre la curva. . . . . . . . . . . . . . . . . . . . . . 43
5.9. Multiplicación escalar. . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5.10. Multiplicación escalar sobre $E(\mathbb{F}_{2^{23}})$. . . . . . . . . . . . . . . . . . 45
5.11. Multiplicación escalar sobre $E(\mathbb{F}_{2^{89}})$. . . . . . . . . . . . . . . . . . 46
5.12. Multiplicación escalar sobre $E(\mathbb{F}_{2^{233}})$. . . . . . . . . . . . . . . . . 48
5.13. Conexión física de los sistemas de computo. . . . . . . . . . . . . . . 51
5.14. Servidor. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.15. Host 1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.16. Host 2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

6.1. Diagrama de bloques para la multiplicación Karatsuba-Ofman. . . . 59
6.2. Proceso de paralelización para la multiplicación Karatsuba-Ofman. . 60
6.3. Reducción modular: $z^{233} + z^{74} + 1$. . . . . . . . . . . . . . . . . . . . 60

6.4. Arquitectura completa para la multiplicación sobre $\mathbb{F}_{2^m}$. . . . . . . . . 61
6.5. Elevar al cuadrado un elemento $a \in \mathbb{F}_{2^m}$. . . . . . . . . . . . . . . . . 61
6.6. Multiplicador K-O de 4 bits. . . . . . . . . . . . . . . . . . . . . . . . . 63
6.7. Multiplicador K-O de 8 bits. . . . . . . . . . . . . . . . . . . . . . . . . 65
6.8. Multiplicador K-O de 233 bits. . . . . . . . . . . . . . . . . . . . . . . . 66
6.9. Multiplicador K-O de 233 bits. . . . . . . . . . . . . . . . . . . . . . . . 66
6.10. Multiplicador K-O de 233 bits. . . . . . . . . . . . . . . . . . . . . . . . 67
6.11. Multiplicador K-O de 233 bits. . . . . . . . . . . . . . . . . . . . . . . . 67
6.12. Multiplicador de 233 bits. . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.13. Multiplicador de 233 bits. . . . . . . . . . . . . . . . . . . . . . . . . . 69
6.14. Multiplicador de 233 bits. . . . . . . . . . . . . . . . . . . . . . . . . . 71
6.15. Cuadrado de 233 bits. . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
6.16. Cuadrado de 233 bits. . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.17. Cuadrado de 233 bits. . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

# Capítulo 1

# Introducción

Hoy en día se maneja una gran cantidad de información que viaja a través de distintos sistemas de cómputo, donde dicha información puede ser de carácter confidencial, para lo cual se requieren de algoritmos que sean capaces de guardar su integridad y su privacidad, teniendo como objetivo principal que la información no sea visible ante cualquier persona.

La criptografía es la ciencia de cifrar y descifrar información utilizando técnicas matemáticas que hagan posible el intercambio de mensajes de manera que sólo puedan ser leídos por las personas correspondientes. La palabra criptogría proviene del griego Kryptos (ocultar) y grafos(escribir), que literalmente significa escritura oculta.

En la critografía, el termino *cifrar* significa ocultar el mensaje aplicando técnicas matemáticas para que no sea visible ante cualquier persona y el termino *descifrar* indica el proceso para obtener el mensaje en claro realizando de manera inversa operaciones matemáticas [1].

La criptogría es la herramienta que comúnmente se emplea para lograr la seguridad en distintas aplicaciones para el envío y transferencia de datos.

Existen dos grandes métodos criptográficos para ocultar la información, estos dos métodos son conocidos como: criptografía simétrica y la criptografía asimétrica.

La criptografía simétrica es aquel criptosistema en el que la llave de cifrado, puede ser calculada a partir de la llave de descifrado y viceversa. En la mayoría de estos sistemas, ambas llaves coinciden por lo que el emisor y el receptor deben mantener en secreto la llave, pero si esta es atacada y se descubre la llave utilizada en la comunicación, el criptosistema se ha roto. De todos los sistemas de llave secreta, el único que se utiliza en la actualidad es el DES (Data Encryption Standard)[2].

La critografía asimétrica desarrollada por Whitfield Diffie y Martin Hellman en la Universidad de Stanford, demostraron la posibilidad de crear un criptosistema en el cual no se precisara la llave secreta durante la transferencia del mensaje, utilizando para esto dos llaves: la llave pública que se hace de conocimiento general y la llave privada que es utilizada para el cifrado del mensaje. Esto hace que para descifrar el mensaje conociendo la llave pública necesariamente sea utilizada la llave privada [3].

La criptografía de llave pública se basa en problemas matemáticos que son computacionalmente difíciles de resolver. Los dos problemas matemáticos mas utilizados para generar la llave pública son la factorización de enteros y el problema del logaritmo discreto.

Un ejemplo de criptosistema de llave pública es el algoritmo RSA desarrollado en el MIT en 1978, ideado por Rivest, Shamir y Adleman, donde se utilizan números primos y la aritmética modular la cual trabaja con subconjuntos finitos de números enteros: los conjuntos de todos los números enteros que tienen el mismo residuo al dividirlos por $n$.

Las curvas elípticas permiten definir problemas similares al logaritmo discreto, al cual se le denomina Problema del Logaritmo Discreto en curvas elípticas (PLDCC). Estas solo se pueden resolver en la actualidad con un algoritmo en tiempo exponencial, con lo cual nos permite construir un criptosistema igualmente seguro a otros, como el RSA, pero con una llave inferior reduciendo así los recursos de memoria, velocidad, ancho de banda e integración en dispositivos móviles, etc.

## 1.1. Motivaciones

Las curvas elípticas son capaces de competir en velocidad con otros criptosistemas como lo son el RSA, además de que la aritmética sobre ellas puede ser implementada con gran eficiencia en hardware y software. En general se cree que estos criptosistemas son bastante seguros, pero no han sido demostrados en su totalidad, se sabe que existe un tipo de curvas que recientemente se ha revelado su vulnerabilidad, por lo que, estas curvas en particular, no son aconsejables para su uso en criptografía.

La seguridad de los sistemas de criptografía basados en curvas elípticas es una buena opción, pese al esfuerzo realizado para intentar atacarlos, hasta el momento no ha habido ninguna sorpresa.

El algoritmo XTR introducido recientemente por Lenstra y Verheul podría convertirse en una competencia para las curvas elípticas, pero sin duda funcionan ligeramente mejor en la práctica y presentan una ventaja definitiva en cuanto al tamaño de las llaves.

Actualmente existen varios intentos por estandarizar los criptosistemas de curvas elípticas que estan sobradamente conocidas aunque su uso en la práctica no está muy extendido.

## 1.2. Objetivos

Realizar en un lenguaje de alto nivel la generación de llaves públicas mediante la utilización de curvas elípticas, e implementar en un lenguaje de descripción de circuitos digitales algunas operaciones aritméticas necesarias para trabajar con curvas elípticas sobre campos finitos $\mathbb{F}_{2^m}$, obteniendo así una mejor optimización.

### 1.2.1. Objetivos particulares

- Implementar en software la operación de doblado, suma de puntos y multiplicación escalar en curvas elípticas para la generación de llaves públicas.

- Diseñar la operación aritmética de la multiplicación sobre campos finitos $\mathbb{F}_{2^m}$, utilizando el algoritmo de Karatsuba-Ofman.

- Implementar el algoritmo de Karatsuba-Ofman sobre campos finitos $\mathbb{F}_{2^m}$ en lenguaje VHDL †[1].
- Diseñar la operación aritmética de elevar al cuadrado sobre campos finitos $\mathbb{F}_{2^m}$.
- Implementar la operación de elevar en lenguaje VHDL.
- Realizar el reporte final de los resultados obtenidos.

## 1.3. Organización del proyecto

Como primera parte se muestran los conceptos básicos de critografía de llave pública y los diferentes tipos de algoritmos que se utilizan para realizar el cifrado. Posteriormente se describen aspectos matemáticos que son muy importantes para el desarrollo de la aritmética sobre curvas elíticas.

Para ver las aplicaciones que se tiene sobre la criptografía con las curvas elípticas, primero se desarrolla un programa en lenguaje de alto nivel que tendrá como objetivo mostrar la forma de como se generan las llaves públicas.

Una vez conociendo las operaciones necesarias para generar llaves públicas, se implementará un segundo programa, para lo cual se emplean tres sistemas de cómputo simulando un servidor y dos clientes los cuales hacen el intercambio de llaves realizando una petición de parámetros al servidor.

Al trabajar con curvas elípticas son necesarias las operaciones de suma y producto en aritmética de campos finitos. La adición y sustración son equivalentes en $\mathbb{F}_{2^m}$.

Aunque actualmente existen diferentes programas de software que trabajan con curvas elípticas, la desventaja que existe al trabajar sobre software es que el tiempo de ejecución es muy costoso a medida que se trabajan en campos finitos muy grandes. Esto nos lleva a diseñar nuevos algoritmos para desarrollar las operaciones aritméticas en campos finito que permitan ser implementados en hardware para tener un mejor rendimiento en cuanto a recursos de tiempo y así lograr el objetivo principal al trabajar con curvas elípticas que es la generación de llaves fuertes.

Por tal motivo en el presente trabajo también se implementará la operación de multiplicación sobre campos finitos $\mathbb{F}_{2^m}$ en lenguaje VHDL, ya que esta es considerada unas de las operaciones más difíciles (aparte de la operación del inverso aditivo) de implementar en hardware.

Adicionalmente también se implementará la operación de elevar al cuadrado sobre campos finitos $\mathbb{F}_{2^{233}}$ que es una operación más sencilla que la multiplicación.

[1]†VHSIC Hardware Description Language. Lenguaje para describir circuitos digitales.

# Capítulo 2

# Criptografía de llave pública

## 2.1. Algoritmos asimétricos de cifrado

Los algoritmos de llave pública o algoritmos asimétricos, han sido de gran interés para ser empleados en redes de comunicación inseguras como es Internet. Estos fueron ideados por Whitfield Diffie y Martin Hellman [4] , donde su novedad fundamental con respecto a la criptografía simétrica es que las llaves no son únicas, si no que forman pares. Existe una gran cantidad de algoritmos asimétricos, la mayoría de los cuales son inseguros y en la práctica muy pocos algoritmos son realmente utilizados. El más utilizado es el RSA, ya que este ha sobrevivido a diversos ataques aunque la longitud de sus llaves es considerable. Otros algoritmos que también son utilizados son los de ElGamal y Rabin [5].
Los algoritmos asimétricos emplean generalmente longitudes de llave mucho mayores que los simétricos así mismo la complejidad de cálculo que se necesita es considerablemente mucho mayor.

## 2.2. Aplicaciones de algoritmos asimétricos

Estos algoritmos poseen dos llaves diferentes $K_A$ y $K_a$, llamadas llave privada y llave pública respectivamente. Una de estas llaves se emplea para cifrar y la otra para decodificar. Para asegurar su fortaleza estos criptosistemas deben cumplir que a partir de una de las llaves resulte computacionalmente difícil calcular la otra.

La aplicación más interesantes al utilizar algoritmos asimétricos, es el cifrado de información sin la necesidad de transmitir la llave de decodificación en canales inseguros (Internet).

Supongamos que existe una entidad **A** que desea enviar un mensaje cifrado a la entidad **B** (fig. 2.1). Para ello la entidad **A** solicita a **B** su llave pública $K_b$. De esta manera **A** genera el mensaje cifrado $E_{K_p}(m)$, donde la llave $K_p$ es calculada a partir de la llave pública de **B** y la llave privada de **A**. Posteriormente únicamente quien posea la llave $K_B$, en este caso solo la entidad **B**, podrá recuperar el mensaje original $m$. Entonces la llave pública permite codificar los mensajes, mientras que la llave privada es aquella que permite decodificarlos.

### 2.2.1. Firma Digital

Otra aplicación que tienen los algoritmos asimétricos es la autenticación de mensajes mediante las firmas digitales [6] a partir del mensaje, esta firma es mucho más pequeña que el mensaje original la cual es conocida como el hash del mensaje.

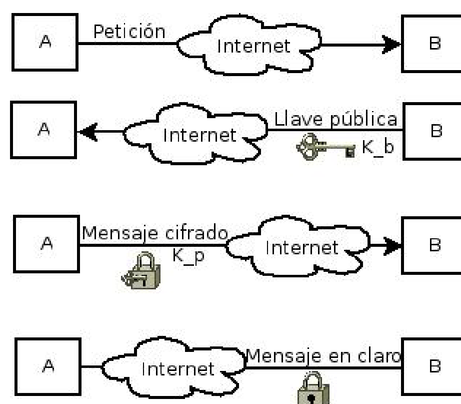


Figura 2.1: Transmisión de un mensaje utilizando algoritmos asimétricos.

### Procedimiento

Supongamos que la entidad **A** recibe un mensaje $m$ de **B** y quiere comprobar su autenticidad, entonces **B** genera el hash del mensaje $r(m)$ y lo codifica empleando la llave de cifrado, es este caso será la llave privada.
La llave de descifrado se habrá hecho pública anteriormente y el cual debe conocer **A**. A partir de esto **B** envía a **A** el criptograma correspondiente a $r(m)$. La entidad **A** genera su propia hash $r'(m)$ y compara con el valor hash $r(m)$ obteniendo el criptograma enviado por $B$. Si en este caso coinciden, el mensaje será auténtico, ya que el único que puede codificar el mensaje de la misma forma es la entidad **B**. Cabe notar que en este caso la llave que se emplea para cifrar es la llave privada.

La firma digital se puede definir como un método que se emplea para la autentificación de información digital. La firma digital en la transmisión de mensajes electrónicos se asegura que:

- Autenticidad: Un critosistema de llave pública permite que cualquier persona envíe un mensaje empleando criptografía de llave pública. Una firma digital permite al receptor de un mensaje estar seguro de que el transmisor del mensaje, es el auténtico.

- Integridad: Esta propiedad refiere a que tanto el receptor como el emisor puedan estar seguros de que la información no ha sido alterada de alguna forma por una tercera persona. El cifrado de mensajes podría proporcionar esta propiedad, ya que la información no es legible ante terceras personas, sin embargo la información puede ser alterada de alguna forma.

- No repudio: Hablando en termino criptográfico se refiere a que si un mensaje es enviado, el emisor no puede negar el envío del mensaje.

## 2.3. El algoritmo RSA

Su nombre se debe a sus tres inventores: Ronald Rivest, Adi Shamir y Leonard Adleman, estuvo bajo la patente de los Laboratorios RSA [7] hasta el 20 de septiembre del 2000, por lo que su uso comercial estuvo restringido hasta esa fecha. RSA basa su fuerza en la dificultad de factorizar números grandes. La llave pública y privada se calculan a partir de un número que se obtiene como producto de dos números primos grandes, así, si un atacante pretende romper la llave tendrá que enfrentarse al problema de factorización, el cúal hasta la fecha no ha sido resuelto.

### 2.3.1. Generación de llaves

Para generar un par de llaves $(k_x, k_y)$, primero se eligen dos números primos grandes $p$ y $q$ de manera aleatoria y la misma longitud en bits, para posteriormente calcular $n = pq$.

Sea $\phi$ de la función phi de Euler. Se escoge ahora un número $e$ primo relativo con $\phi(n) = (p-1)(q-1)$, con esto construimos la llave pública $(e, n)$. El número $e$ debe cumplir que exista su inverso módulo $\phi(n)$, por lo que debe existir un número $d$ tal que: $de \equiv 1 \pmod{\phi(n)}$, es decir que $d$ es el inverso de $e$ mod $\phi(n)$. La pareja $(d, n)$ será la llave privada. El inverso puede determinarse de manera eficiente aplicando el algoritmo extendido de Euclides, pero si desconocemos los factores de $n$, el cálculo resulta muy díficil. En la práctica los sistemas de RSA poseen un tamaño de llave de al menos 1024 bits, lo que es equivalente al nivel de seguridad proporcionado por una llave de 80 bits de un sistema simétrico [8].

Un par de llaves de tipo RSA se resume empleando el algoritmo 1.

**Algoritmo 1** Generación de llaves RSA

**Entrada:** Párametro de seguridad $l \in \mathbb{Z}$.

**Salida:** Llave pública $(n, e)$ y llave privada $d$.

1: Seleccionar aleatoriamente dos números primos p, q de tamaño $\frac{l}{2}$ bits.
2: Calcular $n = pq$ y $\phi(n) = (p-1)(q-1)$.
3: Seleccionar un entero aleatorio $e$, tal que $1 < e < \phi(n)$ y $\gcd(e, \phi(n)) = 1$.
4: Calcular $d$ tal que $1 < d < \phi(n)$ y $ed \equiv 1 (\mathrm{mod}\ \phi(n))$.
5: Devolver $(n, e, d)$.

### 2.3.2. Esquema de firma digital y verificación

Las operaciones de firma digital y verificación se realizan con los algoritmos 2 y 3. La entidad que envía un mensaje $m$, primero debe calcular el hash $h = H(m)$ empleando una función $H$, donde $h$ funciona como marca única de $m$. Luego, quien envía el mensaje, ocupa su propia llave privada $d$ para calcular $s = h^d \bmod n$. Posteriormente se envía el mensaje $m$, así como la firma $s$ al receptor, este último deberá calcular el hash $h = H(m)$ y además debe calcular $h' = s^e \bmod n$ a partir de $s$; la firma $s$

de $m$ será válida si y solo si $h' = h$. La seguridad de este tipo de esquema recae en la intratabilidad matemática de calcular la llave privada $d$ a partir de la firma $s$ y la llave pública $(n, e)$.

**Algoritmo 2** Generación de firma digital RSA

**Entrada:** Llave pública $(n, e)$, llave privada $d$, mensaje $m$.
**Salida:** Firma $s$.
1: Calcular $h = H(m)$, siendo $H$ una función hash.
2: Calcular $s = h^d \bmod n$.
3: Devolver $s$.

**Algoritmo 3** Verificación de firma digital RSA

**Entrada:** Llave pública $(n, e)$, mensaje $m$, firma $s$.
**Salida:** Aceptar o rechazar la firma.
1: Calcular $h = H(m)$.
2: Calcular $h' = s^e \bmod n$.
3: Si $h = h'$ entonces aceptar la firma; si no rechazar la firma.

## 2.4. Algoritmo de encriptación basado en el Problema del Logaritmo Discreto

Estos algoritmos fueron ideados por los matemáticos Whitfield Diffie y Martin Hellman (DH) con el informático Ralph Merkle a mediados de los 70, estos algoritmos han demostrado su seguridad en comunicaciones inseguras como Internet. En la actualidad existen diversos algoritmos de este tipo pero han demostrado ser poco utilizables en la práctica ya sea por el tamaño de las llaves, el tamaño del texto cifrado generado o su velocidad que frecuentemente suele ser lenta.

Diffie-Hellman esta basado en las propiedades y en el tiempo necesario para calcular el valor del logaritmo de un número primo muy grande.

El protocolo Diffie-Hellman se resume de la siguiente manera:

- Las dos entidades o máquinas A y B interesadas en compartir una llave acuerdan un campo primo $\mathbb{F}_p$ y un generador $\alpha$ del grupo cíclico $(\mathbb{F}_p)^* = \mathbb{F}_p \setminus \{0\}$.
- La entidad A genera un valor aleatorio $a$, calcula $m = \alpha^a \bmod p$ y envía a B dicho valor
- La entidad B genera un valor aleatorio $b$, calcula $n = \alpha^b \bmod p$ y envía a A dicho valor
- La entidad A calcula $s = m^n \pmod p$ con $m^n = \alpha^{ba} \pmod p$

- La entidad B calcula $s = n^m \pmod{p}$ con $n^m = \alpha^{ba} \pmod{p}$
- El secreto compartido entre A y B es $s$. Si otra entidad quiere descubrir $s$ necesitara resolver el problema del logaritmo discreto.

# Capítulo 3

# Aspectos matemáticos

## 3.1. Definiciones y teoremas

Las siguientes definiciones fueron tomadas de [9].

**Definición** Un grupo $(G, *)$ consiste de un conjunto $G$ con una operación binaria sobre $G$, satisfaciendo los siguientes tres axiomas:

- Asociativa: $a * (b * c) = (a * b) * c$, para todo $a, b$ y $c \in G$.
- Identidad: Existe el elemento $1 \in G$, tal que $a * 1 = 1 * a = a$, para cada $a \in G$.
- Inverso: Para cada $a \in G$ existe el elemento $a^{-1}$ tal que $a * a^{-1} = a^{-1} * a = 1$.

  Un grupo se considera abeliano si además cumple la siguiente propiedad :

- Conmutatividad: $a * b = b * a$ para todo $a, b \in G$.

**Definición** Un subconjunto no vacío $H$ de un grupo $G$ se considera como un subgrupo de $G$ si $H$ es en sí mismo un grupo con respecto a la operación de $G$. Si $H$ es un subgrupo de $G$ y $H \neq G$, entonces $H$ es llamado un subgrupo propio de $G$.

**Definición** Un grupo $(G, *)$ es *cíclico* si existe un elemento $\alpha \in G$ tal que para cada $b \in G$ existe un $i$ con $b = \alpha^i = \alpha * \alpha \ldots * \alpha$ . Tal elemento $\alpha$ es llamado un generador de $G$.

**Teorema 1** *Si $G$ es un grupo y $a \in G$, entonces el conjunto de todas las potencias de $a$ forma un subgrupo cíclico de $G$, llamado el subgrupo generado por $a$, y denotado por $\langle a \rangle$.*

**Definición** Sea $G$ un grupo y $a \in G$. El orden de $a$ denotado por $o(a)$ es definido como el menor entero positivo $t$ tal que $a^t = 1$, si es que tal entero existe. Si $t$ no existe, entonces el orden de $a$ es definido como $\infty$.

**Definición** Un grupo $G$ es finito si la cardinalidad de $G$ denotada por $|G|$ es finita. El orden de G es el número de elementos de $G$, si éste es finito.

**Teorema 2** *Sea $G$ un grupo, y sea $a \in G$ un elemento de orden finito $t$. Entonces $|\langle a \rangle| = t$.*

**Teorema 3** *(Lagrange). Si $G$ es un grupo finito y $H$ es un subgrupo de $G$, entonces $|H|$ divide a $|G|$. Por lo tanto, si $a \in G$ el $o(a)$ divide a $|G|$.*

**Definición** Un anillo $(R, +\ , \times)$ consiste de un conjunto $R$ con dos operaciones binarias denotadas $+$ (adición) y $\times$ (multiplicación) sobre $R$ satisfaciendo los siguientes axiomas:

- $(R, +)$ es un grupo abeliano con identidad denotada por 0.
- La operación $\times$ es asociativa. Esto es, $a \times (b \times c) = (a \times b) \times c$, para todo $a, b, c \in R$.
- Existe una identidad multiplicativa denotada por 1, con $1 \neq 0$, tal que $1 \times a = a \times 1 = a$, para cada $a \in R$.
- La operación $\times$ es distributiva sobre $+$. Esto es, $a \times (b + c) = (a \times b) + (a \times c)$ y $(b + c) \times a = (b \times a) + (c \times a)$, para cada $a, b, c \in R$.

El anillo $R$ es un *anillo conmutativo* si $a \times b = b \times a \; \forall \; a, b \in R$.

**Definición** Un elemento $a$ de un anillo $R$ es llamado una unidad o un elemento invertible si existe un elemento $b \in R$ tal que $a \times b = 1$.

**Definición** Un campo es un anillo conmutativo en el cual todos los elementos no nulos tienen inversos multiplicativos.

**Definición** Un subconjunto $F$ de un campo $E$ es un subcampo de $E$ si $F$ es en si mismo un campo con respecto a las operaciones de $E$. Si esto se cumple, $E$ se dice que es una extensión de $F$.

**Definición** Un campo finito es un campo $F$ el cual contiene un número finito de elementos. El orden de $F$ es el número de elementos de $F$.

**Definición** Sea $R$ un anillo conmutativo. El anillo de polinomios con coeficientes en $R$ denotado por $R[x]$ es el anillo formado por el conjunto de todos los polinomios en la indeterminada $x$ teniendo coeficientes en $R$. Las dos operaciones son la suma y la multiplicación estándar de polinomios, con coeficientes que operan en el anillo $R$.

**Definición** Sea $f(x) \in F[x]$ un polinomio de grado al menos 1. Entonces $f(x)$ se dice que es un polinomio irreducible sobre $F$ si éste no puede ser escrito como el producto de dos polinomios $p(x), q(x) \in F[x]$, cada uno de grado positivo.

**Teorema 4** *Sea $f(x) \in \mathbb{Z}_p[x]$ un polinomio irreducible de grado $m$. Entonces $\mathbb{Z}_p[x]/(f(x))$ es un campo finito de orden $p^m$.*

**Teorema 5** *El polinomio irreducible $f(x) \in \mathbb{Z}_p[x]$ de grado $m$ es un polinomio primitivo si y solo si $f(x)$ divide $x^k - 1$ para $k = p^m - 1$ y para enteros positivos no menores a $k$*

**Teorema 6** *Para cada $m \geq 1$ existe un polinomio mónico primitivo de grado $m$ sobre $\mathbb{Z}_p$. De hecho, hay precisamente $\phi(p^m - 1)/m$ polinomios.*

Por lo tanto todo campo finito tiene una representación polinomial. Por lo que los elementos de un campo finito $\mathbb{F}_{p^m}$ pueden ser representados por polinomios en $\mathbb{Z}_p[x]$ de grado $< m$ y con coeficientes en $\mathbb{Z}_p$.

**Ejemplo**

Considere el campo finito $\mathbb{F}_{2^4}$ de orden 16 y sea $f(x) = x^4 + x + 1$ el polinomio irreducible sobre $\mathbb{Z}_2[x]$. Por lo tanto el campo finito $\mathbb{F}_{2^4}$ puede ser representado por el conjunto de todos los polinomios sobre $\mathbb{F}_2$ de grado a lo mas 4. Esto es,

$$\mathbb{F}_{2^4} = \{a_3x^3 + a_2x^2 + a_1x + a_0 | a_i \in \mathbb{F}_2\}.$$

Por conveniencia, el polinomio $a_3x^3+a_2x^2+a_1x+a_0$ es representado por un vector $(a_3a_2a_1a_0)$ de longitud 4, y

$$\mathbb{F}_{2^4} = \{(a_3a_2a_1a_0)|a_i \in 0, 1\}.$$

Los siguientes son algunos ejemplo de aritmética sobre campos.

- Los elementos simplemente son sumados componente a componente: por ejemplo, (1011) + (1001) = (0010).

- Para multiplicar los elementos del campo (1101) y (1001), multiplicarlos como polinomios y tomar el resultado de este producto y dividirlo por $f(x)$:
  $(x^3 + x^2 + 1)(x^3 + 1) = x^6 + x^5 + x^2 + 1 \equiv x^3 + x^2 + x + 1 \pmod{f(x)}$. Por lo tanto (1101) (1001) = (1111).

- La identidad multiplicativa de $\mathbb{F}_{2^4}$ es (0001).

- El inverso multiplicativo de (1011) is (0101). Esto se puede verificar, si se observa que $(x^3+x+1)(x^2+1) = x^5+x^2+x+1 \equiv 1 \ (\mathrm{mod}(f(x)))$, donde (1011) (0101) = (0001).
  $f(x)$ es un polinomio primitivo o equivalentemente el elemento $x = (0010)$ del campo es un generador de $\mathbb{F}^*_{2^4}$. Esto puede ser comprobado verificando que todos los elementos no nulos en $\mathbb{F}_{2^4}$ pueden ser obtenidos con potencias de $x$, Tabla 3.1.

**Definición** Un polinomio irreducible $f(x) \in \mathbb{Z}_p[x]$ de grado $m$ es llamado un *polinomio primitivo* si $x$ es un generador del grupo cíclico $\mathbb{F}^*_{p^m}$, el grupo multiplicativo de todos los elementos no nulos en $\mathbb{F}_{p^m}$.

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

**Ejemplo**

Sea $p = 97$, entonces $\mathbb{Z}^*_{97}$ es un grupo cíclico de orden $n = 96$. Un generador de $\mathbb{Z}^*_{97}$ es $\alpha = 5$. Ya que $5^{32} \equiv 35 (\mathrm{mod}\ 97)$, $log_5 35 = 32$ en $\mathbb{Z}^*_{97}$.

| i | $x^i$ mod $x^4+x+1$ | notación vector |
|---|---|---|
| 0 | $1$ | (0001) |
| 1 | $x$ | (0010) |
| 2 | $x^2$ | (0100) |
| 3 | $x^3$ | (1000) |
| 4 | $x+1$ | (0011) |
| 5 | $x^2+x$ | (0110) |
| 6 | $x^3+x^2$ | (1100) |
| 7 | $x^3+x+1$ | (1011) |
| 8 | $x^2+1$ | (0101) |
| 9 | $x^3+x$ | (1010) |
| 10 | $x^2+x+1$ | (0111) |
| 11 | $x^3+x^2+x$ | (1110) |
| 12 | $x^3+x^2+x+1$ | (1111) |
| 13 | $x^3+x^2+1$ | (1101) |
| 14 | $x^3+1$ | (1001) |

Tabla 3.1: Potencia de $x$ módulo $f(x) = x^4 + x + 1$

**Definición** El problema del logaritmo discreto (DLP) es el siguiente: Se toma un numero primo $p$, un generador $\alpha$ de $\mathbb{Z}_p^*$, y un elemento $\beta \in \mathbb{Z}_p^*$: El problema del logaritmo discreto, consiste buscar un entero $x$, $0 \leq x \leq p-2$, tal que $\alpha^x \equiv \beta \pmod{p}$.

**Definición** El problema del logaritmo discreto (GDLP) se define como sigue:
Dado 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 $\alpha^x = \beta$.

Para que PLDG tenga un uso practico en el área de criptografía se recomienda que el problema sea difícil de resolver en el grupo $G$.
Una vez definido un grupo cíclico $G$ y un generador $\alpha$, es posible establecer un esquema de llave pública [10].

# Capítulo 4

# Curvas Elípticas

## 4.1. Definición de curvas elípticas

Las curvas elípticas se definen mediante ecuaciones cúbicas. Estas han sido estudiadas durante varios años, actualmente son empleadas en criptografía. Al trabajar con curvas elípticas se puede definir una operación binaria sobre el conjunto de puntos de una manera geométrica lo que hace de dicho conjunto un grupo abeliano.

**Definición** Una curva elíptica es de la forma:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_5$$

Donde $a_1$,. . ., $a_5$ son elementos constantes que pertenecen a un campo.

**Definición** El orden de una curva elíptica $E$ definida sobre el campo $\mathbb{F}_q$ es el número de puntos que existe sobre la curva y es denotado por $\#E(\mathbb{F}_q)$

### 4.1.1. Curvas elípticas sobre el campo de números reales $\mathbb{R}$.

Una curva elíptica sobre $\mathbb{R}$, es definida como el conjunto de puntos $(x, y)$ con $x$, $y \in \mathbb{R}$ que satisfacen la ecuación de Weierstrass simplificada $E : y^2 = x^3 + ax + b$, donde $a$ , $b$ $\in \mathbb{R}$, y $\Delta = 4a^3 + 27b^2$ es el discriminante. Si $\Delta \neq 0$, la curva no tiene raíces repetidas. Para cada pareja de valores $(a, b)$ existe una curva elíptica diferente.

Por ejemplo si de la curva tomamos el valor de $a = -11$ y $b = 4$ la representación geométrica que se tiene es la mostrada en la Figura 4.1.

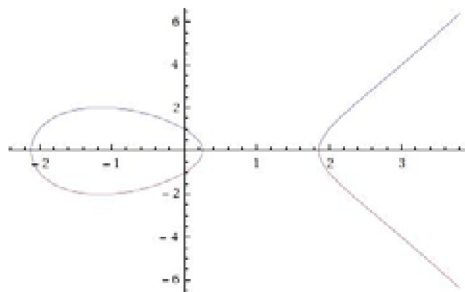

Figura 4.1: $y^2 = x^3 - 11x + 4$

Ahora si tomamos el valor de $a = -3$ y $b = 2$ el discriminante $\Delta = 0$ y su representación geométrica se puede observar en la Figura 4.2.

#### Suma de puntos en una curva elíptica $E$ sobre el campo $\mathbb{R}$

Sean dos puntos $P$ y $Q$ que estén sobre la curva $E$, se traza una línea recta que pase por esos dos puntos el resultado será una nueva intersección sobre la curva, ese nuevo punto será $-R$. Ahora si trazamos una línea vertical sobre el punto $-R$ el resultado es otro punto $R$, entonces para el caso general podemos definir la suma de dos puntos como $R = P + Q$. Figura 4.3.

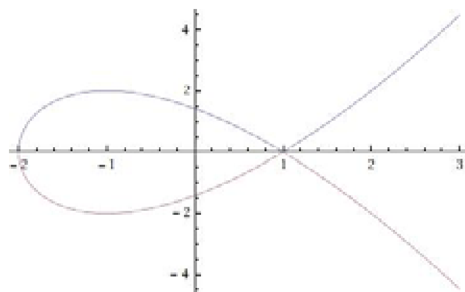

Figura 4.2: $y^2 = x^3 - 3x + 2$ con $\Delta = 0$

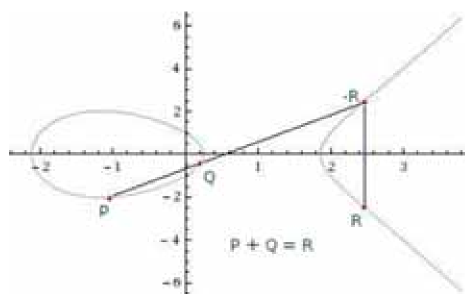


Figura 4.3: Suma de puntos sobre la curva $y^2 = x^3 - 11x + 4$

**Elemento inverso en una curva elíptica sobre el campo $\mathbb{R}$**

La Figura 4.3 muestra también el negativo del punto $R$. El punto $R = (x, y)$ es geométricamente la reflexión del punto $y$, es decir; $-R = (x, -y)$.

**Elemento identidad en una curva elíptica sobre $\mathbb{R}$**

Considere un punto $P$ sobre la curva y si queremos saber con que otro punto sumado me da como resultado el mismo punto, necesitamos un punto extra que se le llama $punto\ al\ infinito$ y lo denotaremos por $\vartheta$.
Se puede decir que el punto al infinito $\vartheta$ está infinitamente lejos sobre el eje vertical y es la *identidad* en una curva elíptica, y cumpliendo la propiedad $P + \vartheta = P$.
Así mismo tenemos que dado $P = (x, y)$ y su inverso $-P = (x, -y)$ la suma de los puntos da como resultado el punto al infinito $P + (-P) = \vartheta$. Figura 4.4.

**Producto de un entero $n$ con un punto de una curva elíptica sobre $\mathbb{R}$.**

La multiplicación entre un punto $P(x, y) \in E$ con $y \neq 0$ y un entero $n$ consiste en sumar el punto $P$, $n$ veces. Cuando $n = 1$ tenemos que: $1P = P$. Si $n = 2$ entonces se traza una tangente a la curva en el punto $P$. La tangente intersecta la curva en un segundo punto $-R$ y por lo tanto $2P = R$, Figura 4.5. Para $n = 3$ se calcula $3P = 2P + P$.
Si $y = 0$, la tangente es vertical y no intersecta a la curva en mas puntos. Por definición $2 * P = \vartheta$, Figura 4.6. En general $n * P = \vartheta$ si n es par y $n * P = P$ si n es impar

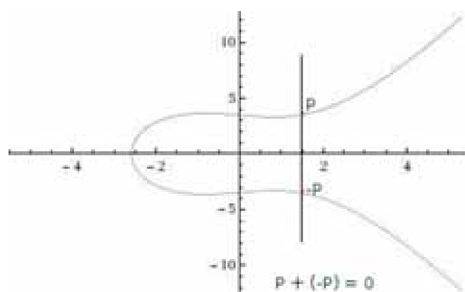


Figura 4.4: Suma del punto $P$ con su inverso $-P$ sobre la curva $y^2 = x^3 - 2x + 12$

cuando $y = 0$.

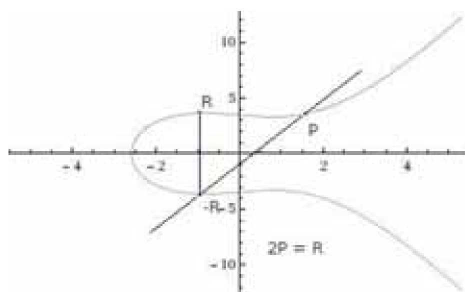


Figura 4.5: Suma del punto $P$ consigo mismo de la curva $y^2 = x^3 - 2x + 12$

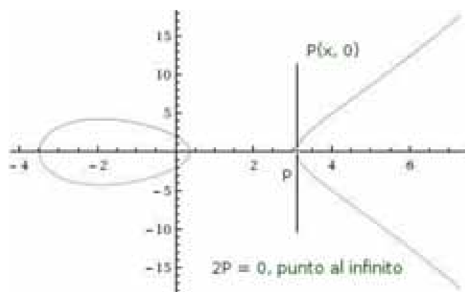


Figura 4.6: Suma del punto $P$ consigo mismo con la coordenada $y = 0$ de la curva $y^2 = x^3 + 4x + -7$

El campo $\mathbb{R}$ no se considera práctico para la critografía debido a los errores que pueden darse en el redondeo de los valores. En su lugar se utilizan los campos primos finitos $\mathbb{F}_p$ donde $p$ es un número primo y los campos finitos de la forma $\mathbb{F}_{2^m}$, $m \geq 1$, que son una alternativa muy confiable en criptografía moderna [13]. En este trabajo solo consideraremos los campos de la forma $\mathbb{F}_{2^m}$.

### 4.1.2. Definición de curvas elípticas sobre $\mathbb{F}_{2^m}$

Los campos finitos $\mathbb{F}_{2^m}$ con $m \geq 1$ resultan muy convenientes, ya que al ser de característica 2 los coeficientes de los polinomios pueden ser representados como una cadena de bits facilitando su implementación en hardware.

**Definición** Si $k$ es un número entero positivo y $P \in \mathbb{F}_{2^m}$, entonces $kP = \sum_{i=1}^{k} P$, denota el punto obtenido al sumar $k$ veces el punto $P$ a sí mismo. Este proceso de calcular $kP$ a partir de $P$ y $k$ es llamado multiplicación escalar.

**Definición** El orden de un punto es el menor número de veces que hay que sumar un punto $P$ consigo mismo para obtener el cero de la curva $E$. Básicamente es encontrar el número mas pequeño $k$ tal que $kP = \vartheta$. El orden de cualquier punto siempre divide al orden $\#E(\mathbb{F}_{2^m})$ de la curva, esto garantiza que si $k_1$ y $k_2$ son enteros, entonces $k_1 P = k_2 P$ si y solo si $k_1 \equiv k_2 \bmod q$.

### 4.1.3. Curvas elípticas sobre $\mathbb{F}_{2^m}$

La ecuación utilizada en estos campos, simplificada de la ecuación de Weierstrass, se ajusta a:

$$E : y^2 + xy = x^3 + ax^2 + b \tag{4.1}$$

Donde $a, b \in \mathbb{F}_{2^m}$ con $\Delta = b \neq 0$. Un punto $P(x, y)$ se encuentra sobre $E$ si satisface la ecuación.

#### Grupo de leyes para $E/\mathbb{F}_{2^m}$ de la forma $y^2 + xy = x^3 + ax^2 + b$

Al igual que en los campos reales en los campos finitos $\mathbb{F}_{2^m}$ se puede definir las operaciones que se usan de la siguiente manera [14]:

- Identidad: Para todo $P \in E(\mathbb{F}_{2^m})$ se tiene que $P + \vartheta = \vartheta + P = P$.
- Negativo: Si $P = (x, y) \in E(\mathbb{F}_{2^m})$, entonces $(x,\ y) + (x,\ x + y) = \vartheta$. El punto $(x,\ x + y)$ es denotado por $-P$ y es llamado el negativo de $P$.
- Suma de puntos: Sean $P(x, y)$ y $Q(x, y) \in E(\mathbb{F}_{2^m})$ y $P \neq {}^{\pm}Q$. Entonces $P + Q = (x_3,\ y_3)$, donde:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \tag{4.2}$$

$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \tag{4.3}$$

$$\lambda = \frac{y_1 + y_2}{x_1 + x_2} \tag{4.4}$$

- Doblado de un punto: Sea $P = (x_1,\ y_1) \in E(\mathbb{F}_{2^m})$ donde $P \neq -P$. Entonces $2P = (x_3,\ y_3)$, donde:

$$x_3 = \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2} \tag{4.5}$$

$$y_3 = x_1^2 + \lambda x_3 + x_3 \tag{4.6}$$

$$\lambda = x_1 + \frac{y_1}{x_1} \tag{4.7}$$

**Ejemplo.**

Para saber que puntos pertenecen a la curva se toma como ejemplo el campo finito $\mathbb{F}_{2^4}$ y se usará el polinomio irreducible $p(z) = z^4 + z + 1$. Un elemento $a_3z^3 + a_2z^3 + a_1z + a_0 \in \mathbb{F}_{2^4}, a_i \in \{0, 1\}$, puede representarse como cadenas de bits. Sea $a = a_3z^3 + a_0$ y $b = a_3z^3 + a_1z + a_0 \in \mathbb{F}_{2^4}, a_i \in \{0, 1\}$.

$$E : y^2 + xy = x^3 + (a_3z^3 + a_0)x^2 + (a_3z^3 + a_1z + a_0) \tag{4.8}$$

$$E : y^2 + xy + x^3 + (z^3 + 1)x^2 + (z^3 + z + 1) = 0 \tag{4.9}$$

Buscamos los valores de $x$ y $y \in \mathbb{F}_{2^4}$ tal que satisfaga la ecuación 4.1.
Si probamos por ejemplo con $x = z^2 + z \in \mathbb{F}_{2^m}$

$$E : y^2 + (z^2 + z)y + (z^2 + z)^3 + (z^3 + 1)(z^2 + 1)^2 + (z^3 + z + 1)$$

$$E : 1 + y^2 + z + yz + z^2 + yz^2 + 4z^3 + 4z^4 + 4z^5 + 3z^6 + z^7$$

Aplicamos residuo módulo 2 y reducimos módulo $p(z)$ con $p(z) = z^4 + z + 1$ que es nuestro polinomio irreducible.

$$1 + y^2 + z + yz + z^2 + yz^2 + 4z^3 + 4z^4 + 4z^5 + 3z^6 + z^7 mod\{2, p(z)\} \tag{4.10}$$

Obtenemos la ecuación:

$$E' : 2 + y^2 + 2z + yz + 2z^2 + yz^2 + 2z^3$$

Probamos ahora por ejemplo el valor de $y = z^2 + z \in \mathbb{F}_{2^m}$

$$E' : 2 + (z^2 + z)^2 + 2z + (z^2 + z)z + 2z^2 + (z^2 + z)z^2 + 2z^3$$

$$E' : 2z^4 + 6z^3 + 4z^2 + 2z + 2$$

Aplicamos residuo módulo 2 y reducimos módulo $p(z)$ con $p(z) = z^4 + z + 1$.

Se obtiene:

$$2z^4 + 6z^3 + 4z^2 + 2z + 2 \; mod \; \{2, p(z)\} = 0$$

$E : 0 = 0$
Satisface la ecuación 4.9.

Por lo tanto $P = (z^2 + z,\ z^2 + z)$ pertenece a la curva

$$E : y^2 + xy + x^3 + (z^3 + 1)x^2 + (z^3 + z + 1).$$

Como $P = (z^2 + z, z^2 + z)$ tiene una representación binaria, tendríamos a $P = (0110,\ 0110)$.

Si calculamos todos los puntos que pertenecen a la curva 4.9 obtenemos:

| $\vartheta$ | (0011, 0111) | (0110, 0000) | (1010, 1010) | (1110, 0001) |
|---|---|---|---|---|
| (0000, 1110) | (0100, 1001) | (0110, 0110) | (1011, 0001) | (1110, 1111) |
| (0001, 1100) | (0100, 1101) | (1000, 0010) | (1011, 1010) | |
| (0001, 1101) | (0101, 0000) | (1000, 1010) | (1100, 0001) | |
| (0011, 0100) | (0101, 0101) | (1010, 0000) | (1100, 1101) | |

Utilizando la aritmética de curvas elípticas sobre campos finitos $\mathbb{F}_{2^m}$ tenemos que: $(0011,\ 0111) + (1010,\ 1010) = (0100,\ 1001)$, Figura 4.7.

$2P = 2(1110,\ 1111) = (1010,\ 0000)$, también el punto $P(1000, 1\,010)$ tiene como negativo al punto $-P(1000,\ 0010)$ que es la reflexión de $P$.

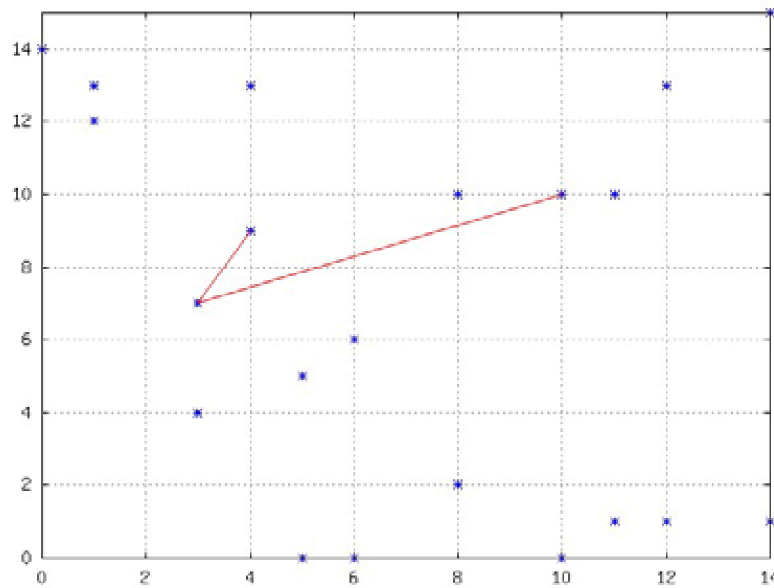


Figura 4.7: Gráfica del campo finito binario $\mathbb{F}_{2^4}$ con $a = z^3 + 1$ y $b = z^3 + z + 1$

## 4.2. Ley de $E(k)$

Una curva elíptica puede clasificarse de acuerdo a la característica de su campo base $K$ y su discriminante $\Delta$. Para curvas definidas sobre campos $\mathbb{F}_{2^m}$se clasifican en:

- No-Supersingulares
  - La curva es definida como: $y^2 + xy = x^3 + ax^2 + b$
  - $\Delta = b$
- Supersingulares
  - La curva es definida como: $y^2 + cy = x^3 + ax + b$
  - $\Delta = c^4$

Sea $E(K)$ el conjunto de puntos pertenecientes a una curva No-Supersingular. El número de puntos en $E(k)$ es llamado el *orden del grupo* y es denotado por $\#E(K)$.

## 4.3. Curvas elípticas de Koblitz

Sea $p(x)$ un polinomio irreducible de grado m sobre $\mathbb{F}_2[x]$, entonces $p(x)$ genera un campo finito $\mathbb{F}_{2^m} = \mathbb{F}_2[x]/p(x)$.

Las curvas elípticas de Koblitz, también llamadas curvas binarias anómalas son definidas sobre $\mathbb{F}_2$ por las ecuaciones:

$$E_0 : y^2 + xy = x^3 + 1 \tag{4.11}$$

$$E_1 : y^2 + xy = x^3 + x^2 + 1 \tag{4.12}$$

mas el punto al infinito $\vartheta$.
Este tipo de curvas son conocidas como $E_a$ de manera genérica donde $a = 0$ ó $a = 1$. El grupo de puntos de la curva elíptica son generados sobre una extensión $k$ de $\mathbb{F}_{2^m}$ y este grupo es denotado $E_a(\mathbb{F}_{2^m})$.

Sea $l$ un divisor de $m$. El grupo $E_a(\mathbb{F}_{2^l})$ es un subgrupo de $E_a(\mathbb{F}_{2^m})$, por lo tanto el orden $\#E_a(\mathbb{F}_{2^l})$ divide al orden $\#E_a(\mathbb{F}_{2^m})$. También el $\#E_0(2) = 4$ y $\#E_1(2) = 2$, por lo tanto el orden de cualquier punto de curvas $E_a$ generado sobre $\mathbb{F}_{2^m}$ es múltiplo de 4 en curvas $E_0$ y múltiplo de 2 en las curvas $E_1$.

Un número $m = nh$ se llama *casi-primo* si este puede ser factorizado como un número primo $n$ y un número relativamente pequeño $h$ con $h \in 2, 3, 4$ [24]. Considere el grupo $E_a(\mathbb{F}_{2^m})$ donde $m$ primo. Si el orden $\#E_a(\mathbb{F}_{2^m})$ es casi-primo, entonces este puede ser factorizádo como $\#E_a(\mathbb{F}_{2^m}) = hn$ donde $n$ es primo y $h$ es llamado el *cofactor* y es:

- $h = 4$ si $a = 0$.
- $h = 2$ si $a = 1$.

## 4.4. Problema del logaritmo discreto en curvas elípticas

Sea $E$ una curva elíptica definida sobre un campo finito y sea $G$ un punto sobre $E$ de orden n. El ECDLP (Por sus siglas en ingles: *Elliptic Curve Discrete Logarithm Problem*) es:
Dados $E$ y $G$ y un múltiplo escalar $Q$ de $G$, determinar un entero $k < n - 1$ tal que $Q = kG$.

## 4.5. Protocolo Diffie-Hellman Elíptico ECCDH

El protocolo Diffie-Hellman elíptico sobre un campo finito $\mathbb{F}_{2^m}$ para la curva E: $y^2+xy = x^3+ax^2+b$, donde $a$ y $b$ son elementos de $\mathbb{F}_{2^m}$. Los pasos para el intercambio de llaves es el siguiente:

- Se acuerda un campo finito binario para las entidades A y B, generando los valores de $m$, $a$, $b$ y un punto $P$ que se encuentran sobre $E$.

Entidad A

. Genera un valor aleatorio entero $v$ (llave privada).
. Calcula $Q = v * P$.
. Envía el punto $Q$ (llave pública).
. Recibe $R$.
. Calcula $S = v * R = v * w * P$.
. Cifra el mensaje utilizando $S$.
. Envía el mensaje cifrado.

Entidad B

. Genera un valor aleatorio entero $w$ (llave privada).
. Calcula $R = w * P$.
. Envía el punto $R$ (llave pública).
. Recibe $Q$.
. Calcula $S = w * Q = w * v * P$.
. Recibe el mensaje cifrado.
. Descifra el mensaje utilizando $S$.

Entidad C

*Recibe los valores de $m, a, b$ y el punto $P(x, y)$.
*Recibe $Q$ (llave pública).
*Recibe $R$ (llave pública).
*Recibe el mensaje cifrado.

Para que C pueda descifrar el mensaje, necesita calcular $v$ y $w$ enfrentándose al PLDCE, Figura 4.8.

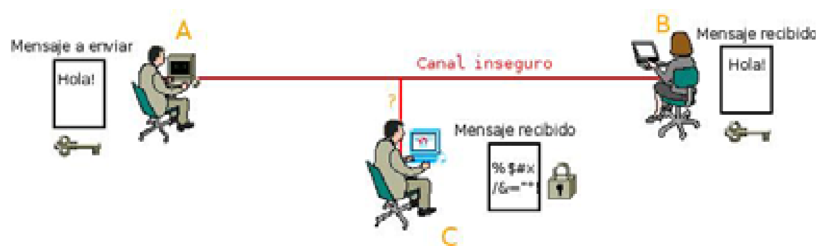


Figura 4.8: Protocolo Diffie-Hellman Eliptico ECCDH, para intercambio de claves.

## 4.6. Ataques a criptosistemas

### 4.6.1. Ataques al IFP

El método de IFP† del RSA utiliza $\mathbb{Z}_n^*$ como grupo, donde $n = pq$ con $p$ y $q$ números primos grandes. Existen algoritmos, de orden sub-exponencial, que resuelven el problema de factorización de $n$. Estos algoritmos explotan propiedades de los números enteros para lograr su objetivo como la propiedad llamada *smoothness*. Un número $n$ es $B$-smooth si todos sus factores primos son $\leq B$. Esta propiedad es aprovechada por algoritmos como Quadratic Sieve (QS) y Number Field Sieve (NFS) [15], los cuales explotan la propiedad de smoothness y utilizan una base de datos de números primos, llamada base de factores, lo cual permite que aumente la probabilidad de encontrar factores del número a factorizar, además de que el NFS utiliza dos bases de factores y es fácil de paralelizar. [1].

### 4.6.2. Ataques al DLP

En cuanto al problema del logaritmo discreto sobre $\mathbb{Z}_p^*$ y $\mathbb{F}_{p^m}^*$, los algoritmos más conocidos son el Pollard-$\rho$ (Pollard's rho) [16] que tiene un tiempo esperado de ejecución de $O(\sqrt{n})$ operaciones, donde $n$ es el orden del grupo cíclico, lo cual hace que el algoritmo no sea sub-exponencial. El algoritmo de Pohlig-Hellman es eficiente solo cuando el orden $n$ del grupo es smooth. El algoritmo más eficiente que no explota ninguna característica de los elementos del grupo es el index-calculus, este método utiliza una base de datos de primos pequeños y sus logaritmos correspondientes es calculada, esto permite calcular eficientemente logaritmos de elementos arbitrarios del grupo. El algoritmo index-calculus [17] y todas sus variantes, como el algoritmo de Coppersmith y el Number Field Sieve (adaptado para logaritmos) tiene un tiempo de ejecución sub-exponencial y son fácilmente paralelizables. El principio de funcionamiento del NFS es el mismo tanto para resolver el IFP como para DLP.

[1] †La seguridad del RSA es basada en el Problema de Factorización Entera, (IFP: *Integer Factorization Problem*)

### 4.6.3. Ataques al ECDLP

Hasta ahora no se conocen algoritmos de tiempo sub-exponencial, es decir que exploten propiedades sobre curvas elípticas para resolver el ECDLP ya que no se conoce el concepto de smoothness de un punto sobre una curva elíptica. Por está razón se cree que el ECDLP es mucho más difícil de resolver que el IFP o el DLP. Los mejores algoritmos conocidos están basados en el Pollard-$\rho$ y el Pollard-$\lambda$. El Pollard-$\rho$ toma alrededor de $\sqrt{\frac{\pi n}{2}}$ pasos, donde cada paso representa la suma de dos puntos en una curva elíptica, y el Pollard-$\lambda$ tomo alrededor de $2\sqrt{n}$ pasos, estos dos métodos pueden ser paralelizados. Recientemente se mostró que el método Pollard-$\rho$ puede ser acelerado por un factor de $\sqrt{2}$, entonces el tiempo esperado de ejecución del método Pollard-$\rho$ con esta mejora es de $\sqrt{\frac{\pi n}{4}}$. Aunque no se conocen algoritmos sub-exponenciales generales para resolver el ECDLP, dos clases de curvas son susceptibles a algoritmos de propósito especial. Primero las curvas elípticas $E\left(\mathbb{F}_q\right)$ con $n$ (el orden del punto base $G$) dividiendo a $q^B - 1$ para $B$ pequeño. Las llamadas curvas supersingulares que son susceptibles a ataques[18]. Estos ataques reducen eficientemente el ECDLP sobre este tipo de curvas al DLP tradicional en una extensión de grado pequeño de $\mathbb{F}_q$.

## 4.7. Eficiencia de los criptosistemas con curvas elípticas

El proceso de generación de los parámetros de un esquema ECC es determinar primero el nivel de seguridad que se desea obtener. Varios factores influyen en la determinación de dichos parámetros, como por ejemplo: el valor de la información, el tiempo que debe ser protegida, el tamaño de los párametros que serán usados, el nivel de seguridad provisto por el esquema, etc.

En la tabla 4.1 se muestra la comparación en cuanto a tamaño necesario de los parámetros de varios esquemas para alcanzar un nivel comparable de seguridad. En la primera columna muestra el tamaño de un parámetro en bits. La segunda columna lista los tamaños de claves en un esquema simétrico ideal. La tercera columna representa el tamaño de un esquema basado en ECC y la cuarta columna para los esquemas RSA y DSA (utilizan el IFP y DLP respectivamente). Cada fila representa un nivel de seguridad comparable entre los distintos métodos. Así podemos decir que los esquemas basados en ECC brindan la misma seguridad que los esquemas tradicionales, pero con un costo más reducido, debido al tamaño de parámetros.

La **Eficiencia** de un criptosistema de clave pública toma en cuenta tres factores:

- Sobrecarga de cálculos: Se refiere a cuantos cálculos computacionales se requiere para ejecutar las transformaciones de clave privada y clave pública.
- Tamaño de clave: Cuantos bits se requieren para almacenar el par de claves y algunos otros parámetros del sistema.

| Nivel de seguridad | Esq. Simétrico (clave) | Esq. ECC (n) | DSA/RSA (modulo) |
|---|---|---|---|
| 56 | 56 | 112 | 512 |
| 80 | 80 | 160 | 1024 |
| 112 | 112 | 224 | 2048 |
| 128 | 128 | 256 | 3072 |
| 192 | 192 | 384 | 7680 |

Tabla 4.1: Tamaños de clave comparables en (bits) [10]

- Ancho de banda: Cuantos bits deben ser comunicados para transferir un mensaje cifrado o una firma.

La implementación de ECC es eficiente particularmente en aplicaciones donde el ancho de banda, capacidad de procesamiento, disponibilidad de energía o almacenamiento están restringidos. Tales aplicaciones incluyen transacciones sobre dispositivos inalámbricos, computación en dispositivos handheld como PDAs o teléfonos celulares, smart cards, etc.

## 4.8. Criptografía con curvas elípticas

Los parámetros del dominio de las curvas elípticas son los valores básicos necesarios para definir el campo finito a usar, como por ejemplo los valores $a$ y $b$ que definen la curva, etc. Aunque estos pueden ser generados por cualquier entidad como la Certificate Authority (CA). Estos parámetros deben ser compartidos por las partes que quieren comunicarse, de manera que en general se trata de utilizar siempre los mismos parámetros recomendados por las organizaciones productoras de estándares.

### 4.8.1. Parámetros de curvas elípticas sobre $\mathbb{F}_{2^m}$

Los parámetros de dominio de una curva elíptica sobre $\mathbb{F}_{2^m}$ son una séptupla:

$$T = (m, f(x), a, b, G, n, h)$$

Consistiendo de un número entero $m$ que define el campo finito $\mathbb{F}_{2^m}$, un polinomio irreducible $f(x) \in \mathbb{F}_2[x]$ de grado $m$, dar explícitamente dos elementos $a$, $b \in \mathbb{F}_{2^m}$ especificando la forma de la curva elíptica $E(\mathbb{F}_{2^m})$ definida por la ecuación:

$$y^2 + xy = x^3 + ax^2 + b \tag{4.13}$$

un punto base $G = (x_G, y_G) \in E(\mathbb{F}_{2^m})$, un número primo $n$ que es el orden de $G$, y un entero $h$ el cual es el cofactor $h = \#E(\mathbb{F}_{2^m})/n$.

El proceso para generar una séptupla $T = (m, f(x), a, b, G, n, h)$ es el siguiente:

1. Seleccionar el nivel de seguridad (en bits) apropiado para los parámetros de curva elíptica, este debe ser un número entero $t \in \{56, 64, 80, 96, 112, 128, 192, 256\}$ de manera que calcular logaritmos sobre la curva elíptica asociada tome aproximadamente $2^t$ operaciones.

2. Sea $t'$ el número entero más pequeño que sea mayor que $t$ en el conjunto $\{64, 80, 96, 112,$
$128, 192, 256, 512\}$. Seleccionar $m \in \{113, 131, 163, 193, 233, 239, 283, 409, 571\}$ tal que $2t < m < 2t'$ para determinar el campo finito $\mathbb{F}_{2^m}$.

3. Seleccionar un polinomio binario irreducible $f(x)$ de grado $m$ de la Tabla 4.2 para determinar la representación de $\mathbb{F}_{2^m}$. Esta restricción esta diseñada para facilitar la interoperabilidad entre implementaciones.

4. Seleccionar elementos $a, b \in \mathbb{F}_{2^m}$ para determinar la curva elíptica $E(\mathbb{F}_{2^m})$ definida por la ecuación $E : y^2 + xy = x^3 + ax^2 + b \in \mathbb{F}_{2^m}$, un punto base $G = (x_G, y_G) \in E(\mathbb{F}_{2^m})$, un número primo $n$ el cual es el orden de $G$, y un número entero $h$, que es el cofactor $h = \#E(\mathbb{F}_2^n)/n$, sujetos a las siguientes restricciones:

    *a*) $b \neq 0$ en $\mathbb{F}_{2^m}$.

    *b*) $\#E(\mathbb{F}_{2^m}) \neq 2^m$.

    *c*) $2^{mB} \neq 1 \pmod{n}$ para todo $1 \leq B < 20$.

    *d*) $h \leq 4$

5. regresar $(m, f(x), a, b, G, n, h)$.

Las restricciones que deben cumplir los parámetros en:
*a*) Se pide que se cumplan la definición de curvas elípticas con respecto al discriminante $\Delta$ sobre $\mathbb{F}_{2^m}$.
*b*) Se pide evitar las curvas anómalas donde se pide estrictamente que $\#E(\mathbb{F}_{2^m}) \neq 2^m$.
*c*) Se pide evitar las curvas supersingulares.
*d*) Se pide que $h \leq 4$ para que $\#E(\mathbb{F}_{2^m})$ sea un número cerca de un primo grande.

La regla para elegir al polinomio irreducible $f(x)$ es elegir un trinomio de la forma $f(x) = x^m + x^k + 1$, con $m > k \geq 1$, y $k$ tan pequeño como sea posible. Si un trinomio irreducible no existe, se opta por un pentanomio de la forma $f(x) = x^m + x^{k_3} + x^{k_2} + x^{k_1} + 1$, $m > k_3 > k_2 > k_1 > 1$, con $k_3$ tan pequeño como sea posible, $k_2$ tan pequeño como sea posible dado $k_3$, y $k_1$ tan pequeño como sea posible dado $k_3$ y $k_2$.

### 4.8.2. Pares de llaves para curvas elípticas

Dado los parámetros de dominio de curvas elíptica $T = (m, f(x), a, b, G, n, h)$, un par de claves de una curva elíptica $(d, Q)$, asociado con $T$ consiste de una clave secreta de una curva elíptica $d$ la cual es un entero en el intervalo $[1, n-1]$, y una clave pública

| Campo | Polinomios irreducibles |
|---|---|
| $\mathbb{F}_{2^{163}}$ | $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$ |
| $\mathbb{F}_{2^{233}}$ | $f(x) = x^{233} + x^{74} + 1$ |
| $\mathbb{F}_{2^{283}}$ | $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$ |
| $\mathbb{F}_{2^{409}}$ | $f(x) = x^{409} + x^{87} + 1$ |
| $\mathbb{F}_{2^{571}}$ | $f(x) = x^{571} + x^{10} + x^5 + x^2 + 1$ |

Tabla 4.2: Polinomios recomendados por NIST en el FIPS 186-2 [11].

de curvas elíptica $Q = (x_Q, y_Q)$, la cual es el punto $Q = dG$. Para generar un par de claves, una entidad procede como lo muestra el algoritmo 4, dados los parámetros (validos) de dominio $T = (m, f(x), a, b, G, n, h)$.

**Algoritmo 4** Generación de llaves [12]

**Entrada:** $T = (m, f(x), a, b, G, n, h)$
**Salida:** $(d, Q)$

1: Calcular aleatoriamente o pseudo aleatoriamente un número entero $d \in [1, n-1]$.

2: Calcular $Q = dG$.
3: Retornar $(d, Q)$.

### 4.8.3. Validación de llaves públicas

Una clave pública de curva elíptica $Q$ se dice que es parcialmente válida si $Q$ es un punto sobre la curva elíptica asociada pero no necesariamente es cierto que $Q = dG$ para algún entero $d$. Se dice valida parcialmente ya que solo es necesario verificar (por una autoridad certificada) que $Q = (x_Q, y_Q) \neq \vartheta$, que $x_Q$ y $y_Q$ sean polinomios de grado a lo mas $m - 1$ para campos finitos $\mathbb{F}_{2^m}$ y que el punto obtenido $Q \in \mathbb{F}_{2^m}$ [10].

# Capítulo 5

# Implementación en software de la aritmética para $E(\mathbb{F}_{2^m})$

## 5.1. Preparación de bibliotecas

Para el desarrollo del proyecto se utilizaron las bibliotecas libres de GMP y LiDIA en donde la primera permite trabajar con números de cualquier longitud y la segunda incluyen operaciones sobre campos finitos que permitirán la creación de llaves públicas sobre el lenguaje de programación C++.

### 5.1.1. Instalación de la biblioteca GMP

Para la instalación de la bibliotecas GMP sera necesario descargarlo [19], esto nos permitirá realizar operaciones muy grandes y prácticamente no existe un limite salvo la disponibilidad de memoria. Las aplicaciones de GMP se utilizan principalmente en la criptografía, álgebra computacional, aplicaciones de seguridad en Internet, etc. Por lo que resulta una herramienta muy práctica para el estudio de curvas elípticas.

Una vez descargada la biblioteca procedemos a la compilación e instalación de esta:

```
#cd gmp-4.3.2
#./configure
#make
#make install
```

### 5.1.2. Instalación de la biblioteca LiDIA

La biblioteca LiDIA permite desarrollar diversas operaciones aritméticas sobre los siguientes anillo y campos $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, así como trabajar con campos primos $\mathbb{F}_p$ y los campos $\mathbb{F}_{2^m}$, factorización de números enteros, álgebra lineal, polinomios, curvas elípticas, etc. Por lo que se descargara [20], se compilará y se instalará de la siguiente forma:

```
#cd lidia-2.3.0/
#./configure
#make
#make install
```

Después de instalar verificamos si se encuentra el siguiente directorio:

- #cd /usr/local/include/LiDIA

La implementación de las operaciones para curvas elípticas están definidas en LiDIA, pero la finalidad del proyecto es implementar estás operaciones sobre hardware y mostrar una idea más a fondo, así que se definirán las operaciones de adición, doblado y multiplicación escalar de $E(\mathbb{F}_{2^m})$ aplicando las operaciones sobre campos finitos $\mathbb{F}_{2^m}$.

## 5.2. Implementación del primer programa

El primer programa realizará las operaciones principales en curvas elípticas como son: la suma de puntos, el doblado de un punto, el negativo de un punto, así como la multiplicación escalar sobre un punto en $\mathbb{F}_{2^m}$.

### 5.2.1. Aritmética GF2n

Los tipos de datos `gf2n` son un tipo de datos muy importante ya que estas nos permitirán realizar operaciones aritméticas sobre campos finitos $\mathbb{F}_{2^m}$ y así podremos definir la suma y doblado de puntos sobre las curvas elípticas. Para definir alguna variable de tipo `gf2n` basta únicamente con escribir: `gf2n` $x$. Algunas de las funciones elementales son descritas a continuación:

```
void x.assign_zero ()
  x ← 0

void x.assign_one ()
  x ← 1

void x.assign(const gf2n & a)
  x ← a

void add(gf2n & x, const gf2n & y, const gf2n & z)
  x ← y + z

void subtract(gf2n & x, const gf2n & y, const gf2n & z)
  x ← y − z

void multiply(gf2n & x, const gf2n & y, const gf2n & z)
  x ← y · z

void invert(gf2n & x, const gf2n & y)
  x ← y^-1, para y ≠ 0

void square(gf2n & x, const gf2n & y)
  x ← y^2

void power(gf2n & x, const gf2n & y, const bigint & i)
  x ← y^i, para y ≠ 0 y i > 0

bool x.is_zero () const
  regresa true si x = 0, false en otro caso
```

`gf2n randomize(const gf2n &` $a$`, unsigned int` $d = m$`)`
  `regresa un elemento` $a$ `del campo definido sobre` $\mathbb{F}_{2^d}$`, cuando` $m|d$

`bool solve_quadratic(gf2n &` $r$`, const gf2n &` $a_1$`, const gf2n &` $a_0$`)`
  `regresa true si existe una raíz` $r$ `del polinomio cuadrático`
  $Y^2 + a_1 Y + a_0$`, false en otro caso`

### 5.2.2. Menú principal

El código empleado en C++ y las funciones necesarias para generar el campo son mostradas en el Apéndice A.

Para realizar las operaciones necesarias sobre curvas elípticas se realizó un menú, en el cual se muestra las siguientes opciones:

1. `Definir m, a y b`
2. `Imprimir Puntos Existentes en la curva (m<=10)`
3. `Mostrar valores del campo`
4. `Sumar dos puntos sobre la curva`
5. `Duplicar punto sobre la curva`
6. `Negativo del punto P sobre la curva`
7. `Generación finita de puntos sobre la curva`
8. `Multiplicación escalar`
9. `Salir`

Por simplicidad de lectura del usuario los elementos que conforman el campo finito, son representados de manera decimal.

#### Definir m, a y b

La primera opción nos permite definir el campo finito $\mathbb{F}_{2^m}$, así como definir los valores de $a$ y $b$ de la ecuación 4.1.

Para lo cual se desarrollo el archivo 'galois_f2n.cpp', donde es definido el parámetro $m$ para generar el campo finito $\mathbb{F}_{2^m}$.

`void gf2n_init(unsigned int` $m$`)`

Esta función permite inicializar el campo finito $\mathbb{F}_{2^m}$, donde recibe como parámetro un tipo de dato entero sin signo. Además esta función selecciona un polinomio irreducible de grado $m$ sobre el campo finito $\mathbb{F}_{2^m}$.

También es necesario definir la característica del campo, la función se define de la siguiente forma:

`ct galois_field` $f$ `(const bigint &` *characteristic*`, lidia_size_t` $m$`)`

1. Donde $f$ generará el campo de tipo galois_field.
2. *characteristic* Define la caraterística de $\mathbb{F}_{c^m}$ $(c = 2)$.
3. $m$ Define el campo finito $\mathbb{F}_{2^m}$.

Para obtener el polinomio irreducible con el cual se está trabajando empleamos la llamada a la función:

`Fp_polynomial` $f$`.irred_polynomial () const`

También podemos obtener el número de elementos sobre $\mathbb{F}_2$:

`bigint` $f$`.number_of_elements () const`

Los valores de m, a y b para definir el campo $\mathbb{F}_{2^m}$ y la ecuación 4.1 por defecto se inicializaron con $m = 4$, $a = 8$ y $b = 9$

De acuerdo a la capacidad de memoria *RAM* podremos definir campos finitos muy grandes teniendo como limite un campo finito $\mathbb{F}_{2^{3000}}$.

Las siguientes implementaciones se encuentran en el archivo llamado 'AritmeticaCE_GF2n.cpp'.

**Impresión de puntos sobre la curva elíptica** $E : y^2 + xy = x^3 + ax^2 + b$

Una vez definido los parámetros sobre los cuales vamos a trabajar, podemos imprimir los puntos existentes sobre la curva elíptica, esto es: Encontrar parejas de valores $(x, y)$ tales que $y^2 + xy - [x^3 + ax^2 + b] = 0$. Cabe resaltar que la subtracción y la adición en campos finitos $\mathbb{F}_{2^m}$ son equivalentes. Para determinar estos valores en un campo relativamente pequeño, podemos usar el siguiente algoritmo 5:

Este algoritmo resulta ineficiente para campos finitos $\mathbb{F}_{2^m}$ grandes, es decir; en la actualidad resulta computacionalmente intratable imprimir todos los puntos, por dicha razón el programa solo imprime todos los puntos cuando el valor de $m \leq 10$.

**Algoritmo 5** Valores $x$ y $y$ que satisfacen la ecuación $y^2+xy+x^3+ax^2+b=0 \in \mathbb{F}_{2^m}$.
Archivo: `AritmeticaCE_GF2n.cpp`

**Entrada:** $a, b \in \mathbb{F}_{2^m}$
**Salida:** $(x, y)$
1: **para** $x = 0$ **hasta** $2^m$ **hacer**
2: sustituir $x$ en la ecuación $y^2 + xy + x^3 + ax^2 + b \in \mathbb{F}_{2^m}$
3: obtener una raíz $y$ de la ecuación cuadrática $y^2 + xy + x^3 + ax^2 + b \in \mathbb{F}_{2^m}$
4: **si** $y^2 + xy + x^3 + ax^2 + b = 0 \in \mathbb{F}_{2^m}$
5: imprimir par de valores $(x, y) \in \mathbb{F}_{2^m}$
6: **fin si**
7: **fin para**

Para visualizar la generación de puntos sobre una curva elíptica se desarrollo adicionalmente el archivo 'Graficar_GF2n.cpp' que permite generar los puntos existentes en la curva escribiéndolos sobre un archivo en forma de script llamado 'GraficaGF2n.m'. El software Octave [21] permitirá graficar dichos puntos siempre y cuando el parámetro $m$ del campo finito $\mathbb{F}_{2^m}$ sea $\leq 10$.

### Mostrar valores del campo

Está opción permite imprimir los valores de $a$ y $b$ de la ecuación $y^2+xy+x^3+ax^2+b$ sobre el campo finito $\mathbb{F}_{2^m}$, también se imprime en pantalla el polinomio irreducible que es necesario para generar el campo, así como el número de combinaciones que se pueden generar en un campo $2^m$

### Suma de puntos en una curva elíptica

Para realizar la suma de dos puntos sobre la curva se utilizará la definición descrita en el grupo de leyes para curvas elípticas definidas sobre el campo finito $\mathbb{F}_{2^m}$ :
$(P_x, P_y) + (Q_x, Q_y) = (R_x, R_y)$, donde:

$$R_x = \lambda^2 + \lambda + P_x + Q_x + a$$

$$R_y = \lambda(P_x + R_x) + R_x + P_y$$

$$\lambda = \frac{P_y + Q_y}{P_x + Q_x}$$

De está manera se implementara la suma de puntos utilizando las operaciones aritméticas sobre $\mathbb{F}_{2^m}$ Se puede notar también que cuando el valor de $P = Q$, es decir el valor de $\lambda$ es indefinido ya que el denominador $P_x + Q_x = 0$, esto nos dará como resultado el punto al infinito $\vartheta$, lo que se tiene que tener en cuenta al realizar la implementación.

Para verificar el valor de $\lambda$ se realizo una función, la cual nos devuelve un valor entero:

int LambdaAdicion(gf2n & $S$, Punto & $P$, Punto & $Q$)

Si $P_x + Q_x = 0$ sobre $\mathbb{F}_{2^m}$ la función regresa 0 en otro caso regresa 1 y $S \leftarrow \lambda$. El tipo de dato *Punto* es una estructura de tipo gf2n con dos valores asociados $x, y$ definida en el archivo 'AritmeticaCE_GF2n.h'.

Si el valor devuelto por `LambdaAdicion` es 0 se imprime en pantalla que la suma de $P + Q =$ *Punto al Infinito*, pero si el valor devuelto es 1 entonces llamamos a la siguiente función:

void AdicionCE_GF2n(Punto & $R$, Punto & $P$, Punto & $Q$, gf2n & $a$)
$R \leftarrow P + Q$ definida sobre $E(\mathbb{F}_{2^m})$

De la misma manera que se imprimieron puntos en la opción dos del menú principal, también podemos visualizar la suma de dos puntos trazando secantes haciendo notar en donde se encuentran localizados dichos puntos siempre que $m \leq 10$ mediante las siguientes funciónes:

void Grafica rPuntos(lidia_size_t & $m$, gf2n & $a$, gf2n & $b$)
void GraficaSuma(Punto & $R$, Punto & $P$, Punto & $Q$)

### Duplicar punto sobre la curva elíptica

Esta operación será realizada de acuerdo al grupo de leyes para curvas elípticas sobre el campo finito $\mathbb{F}_{2^m}$. Para calcular $R = 2P$ se usan las siguientes ecuaciones:

$$R_x = \lambda^2 + \lambda + a$$

$$R_y = {P_x}^2 + \lambda R_x + R_x$$

$$\lambda = P_x + \frac{P_y}{P_x}$$

Y se pide como restricción que la coordenada $P_x \neq 0$.

Como primera parte se obtiene el valor de $\lambda$ el cual nos servirá para determinar el valor de $2P$. En la implementación para calcular el valor de $\lambda$ se utiliza la siguiente función:

int LambdaDoblado(gf2n & $S$, Punto & $P$)

Si $P_x = 0$ sobre $\mathbb{F}_{2^m}$ la función regresa 0 en otro caso regresa 1 y $S \leftarrow \lambda$. Si el valor devuelto por `LambdaDoblado` es 0 se imprime en pantalla que el doblado de $P =$ *Punto al Infinito*, pero si el valor devuelto es 1 entonces llamamos a la siguiente función:

int DobladoCE_GF2n(Punto & $R$, Punto & $P$, gf2n & $S$, gf2n & $a$)

$R \leftarrow 2P$ `definida sobre` $E(\mathbb{F}_{2^m})$

Así mismo podemos graficar los puntos existentes sobre la curva elíptica en $\mathbb{F}_{2^m}$ y visualizar los puntos en juego mediante las siguientes funciones:

```
void GraficarPuntos(lidia_size_t & m, gf2n & a, gf2n & b)
void GraficaDoblado_Negativo(Punto & R, Punto & P)
```

**Negativo del punto P sobre la curva**

Para cada punto $(P_x, P_y)$ de la curva elíptica, existe un punto $(P_x, P_x + P_y)$ es decir la reflexión de $P$ que es conocido como el negativo y es denotado como $-P \ \forall \ P_x \neq 0$, así tenemos que $P + (-P) = \vartheta$. Para hacer esta función, primero se comprueba si la coordenada $P_x \neq 0$ usando la siguiente función:

```
int NegativoPuntoCE_GF2n(Punto & P, Punto & R)
```

Si el valor devuelto por `NegativoPuntoCE_GF2n` es 0 se imprime en pantalla: *El punto P no tiene Negativo* , pero si el valor devuelto es 1 entonces: $R \leftarrow -P$.

De esta manera se modifica el algoritmo 5 y calculamos la reflexión en cada iteración como lo muestra el algoritmo 6

---

**Algoritmo 6** Valores $x$ y $y$ que satisfacen la ecuación $y^2+xy+x^3+ax^2+b = 0 \in \mathbb{F}_{2^m}$.
Archivo: `AritmeticaCE_GF2n.cpp`

---

**Entrada:** $a, b \in \mathbb{F}_{2^m}$
**Salida:** $(x, y)$
1: **para** $x = 0$ **hasta** $2^m$ **hacer**
2: sustituir $x$ en la ecuación $y^2 + xy + x^3 + ax^2 + b \in \mathbb{F}_{2^m}$
3: obtener una raíz $y$ de la ecuación cuadrtíca $y^2 + xy + x^3 + ax^2 + b \in \mathbb{F}_{2^m}$
4: **si** $y^2 + xy + x^3 + ax^2 + b = 0 \in \mathbb{F}_{2^m}$
5: imprimir par de valores $(x, y) \in \mathbb{F}_{2^m}$
6: **si** $x \neq 0$
7: imprimir par de valores $(x, x + y) \in \mathbb{F}_{2^m}$
8: **fin si**
9: **fin si**
10: **fin para**

---

Para graficar la reflexión de un punto se utilizan las mismas funciones descritas para duplicar un punto.

### Generación finita de puntos sobre la curva

Esta opción permite generar elementos de manera aleatoria de acuerdo a los parámetros definidos en un inicio $(m, a, b)$ utilizando la función `randomize`, esta función nos devolverá un valor no mayor al campo finito $\mathbb{F}_{2^m}$, este valor a su vez es sustituido por $x$ en la ecuación $y^2 + xy + x^3 + ax^2 + b$ posteriormente calculamos una raíz de la ecuación cuadrática resultante con la función `solve_quadratic`, si los valores calculados de $x, y$ en $y^2 + xy + x^3 + ax^2 + b = 0$ entonces tenemos un punto $P$ que se encuentra sobre la curva y se obtenemos un valor entero 0. Ya que la función `solve_quadratic` siempre regresa la misma raíz (como es una ecuación cuadrática se tienen dos raíces), se utiliza un numero aleatorio *mod* 2 que sirve como bandera, si el valor devuelto es 1 regresamos el par de coordenadas $(x, y) \in \mathbb{F}_{2^m}$ en otro caso se calcula la reflexión $(x, x + y) \in \mathbb{F}_{2^m}$. La descripción de la función para generar puntos aleatorios es la siguiente:

```
int PuntosAleatoriosCE_GF2n(lidia_size_t &m gf2n &a, gf2n &b, Punto &P)
```

Si el valor devuelto por `PuntosAleatoriosCE_GF2n` es 0 entonces $P \in E(\mathbb{F}_{2^m})$, en otro caso el punto $P \notin E(\mathbb{F}_{2^m})$ y se busca otro punto llamando nuevamente la función.

### Multiplicación Escalar

Una vez definida las operaciones de suma y doblado para $E(\mathbb{F}_{2^m})$ se puede realizar la multiplicación escalar en curvas elípticas. El algoritmo 7 se utilizó para realizar la multiplicación en forma binaria.
Este algoritmo se realiza de una manera muy eficiente en la representación binaria, por lo cual hicimos algunas modificaciones para este programa.
Este algoritmo está basado en la idea de multiplicar $k$ veces el punto P, se toma la representación binaria del escalar $k$ y se realiza un barrido de izquierda a derecha, de tal forma que en cada paso se realiza el doblado del punto y si la posición del bit actual $k_i$ es 1 entonces se realiza la operación de suma del punto actual. Ejemplo:

$$15P = 2P + P \mapsto 6P + P \mapsto 14P + P \mapsto 15P$$

Esta opción permite realizar la multiplicación escalar de dos maneras:
La primera permite generar de manera aleatoria un punto $P \in E(\mathbb{F}_{2^m})$ con la llamada a la función `PuntosAleatoriosCE_GF2n` y un valor entero $k$ la cual será la llave privada y es calculada al azar con la función:

```
void LlavePrivadaCE_GF2n(lidia_size_t & m, gf2n & a, bigint & k)
  k ← llave privada
```

Una segunda opción permite leer un punto $P \in E(\mathbb{F}_{2^m})$ y leer un valor entero $k$ que sera la llave privada. Ya que necesitamos una cadena binaria del valor entero $k$

**Algoritmo 7** Suma y Doblado. Archivo: `AritmeticaCE_GF2n.cpp`

**Entrada:** $k = (k_{j-1} \dots k_1, k_0)$, $P \in \mathbb{F}_{2^m}$
**Salida:** $R = kP$
1: $R \leftarrow P$
2: **para** $i = j - 1$ **hasta** $0$ **hacer**
3: $R \leftarrow 2R$
4: **si** $k_j = 1$ **entonces**
5: $R = R + P$
6: **fin si**
7: **fin para**
8: **regresa** $R$

es necesario realizar su conversión con la función:

`void CadenaBinaria(bigint &` $k$`, char` *binario*`[], int &` *lim*`)`

Donde el parámetro $k$ es la llave privada en decimal, *lim* se usa como referencia para saber el ultimo bit(el menos significativo), y el parámetro *binario* es donde se almacenará el valor binario de $k$, es decir *binario* $\leftarrow k_2$.

De esta forma, se puede realizar la multiplicación escalar utilizando el algoritmo 7 con la función:

`void MultiplicacionEscalarCE_GF2n(Punto &` $R$`, Punto &` $P$`, char` *binario*`[], int & lim, gf2n &` $a$ `)`
$R \leftarrow kP$

**Salir**

Está opción permite finalizar el programa. También se puede visualizar el menú completo oprimiendo la tecla 0.

Por último se implemento el archivo 'menu.cpp' en donde se engloban las diferentes opciones del programa.

## 5.3. Ejecución del primer programa

Para la ejecución del programa primero realizamos la compilación con un simple $make posteriormente ejecutamos $./CE_GF2m, el cual nos desplegara las opciones mostradas en la Figura 5.1.
Donde los valores son por defecto $m = 4$, $a = 8$ y $b = 9$ en la ecuación de Weierstrass

```
|--------------------------------Menu--------------------------------|
|            Curvas Elipticas Sobre Campos Finitos Binarios            |
|                        ******E:/GF(2^m)******                        |
|                    E: y^2 + x*y - [x^3 + a*x^2 + b]                  |
|                                                                      |
|            1. Definir m,a y b                                        |
|            2. Imprimir Puntos Existentes en la curva (m<=10)         |
|            3. Mostrar valores del campo                              |
|            4. Sumar dos puntos sobre la curva                        |
|            5. Duplicar punto sobre la curva                          |
|            6. Negativo del punto P sobre la curva                    |
|            7. Generar Puntos al azar sobre la curva                  |
|            8. Multiplicación escalar                                 |
|            9. Salir                                                  |
|----------------------------------------------------------------------|
        Selecciona una opcion:
>>
```

Figura 5.1: Menú principal del programa.

simplificada $E : y^2 + xy - [x^3 + ax^2 + b]$.

### 5.3.1. Definir m, a y b

La opción 1 permite definir el campo sobre el cual vamos a trabajar de acuerdo a la ecuación $y^2 = x^3 + ax^2 + b$ sobre $\mathbb{F}_{2^m}$ como se muestra en la Figura 5.2.

```
            Selecciona una opcion:
 >>1

Dame el valor de m (F(2^m)):
m = 4

***El campo finito binario esta formado por el polinomio irreducible:
                              x^4 + x + 1 mod 2

***Numero de combinaciones posibles sobre el campo: 16

***Rango de los valores para a: [0,15]
***Rango de los valores para b: [1,15]
valor de
        a = 8
valor de
        b = 9
```

Figura 5.2: Definir m, a y b en la ecuación $y^2 = x^3 + ax^2 + b$ sobre $\mathbb{F}_{2^m}$.

### 5.3.2. Impresión de puntos existentes sobre la curva elíptica

La opción 2 genera los puntos que existan en la curva elíptca de acuerdo a los parámetros que se definieron en la opción 1 como lo muestra la figura 5.3, para visualizar los puntos de manera gráfica, el programa nos pregunta: ' Deseas graficar los puntos sobre la curva S/N? ' a lo cual podemos escribir 'S' o 's' para graficar dichos puntos como se muestra 5.4.

```
***Estos son los puntos que existen en la curva:***
1->(Dec:0, Dec:11); 2->(Dec:1, Dec:0); 2->(Dec:1, Dec:1); 3->(Dec:2, Dec:13); 3->(Dec:2, Dec:15); 4->(Dec:3, Dec:1
5); 4->(Dec:3, Dec:12); 6->(Dec:5, Dec:0); 6->(Dec:5, Dec:5); 8->(Dec:7, Dec:11); 8->(Dec:7, Dec:12); 9->(Dec:8, D
ec:1); 9->(Dec:8, Dec:9); 10->(Dec:9, Dec:6); 10->(Dec:9, Dec:15); 12->(Dec:11, Dec:2); 12->(Dec:11, Dec:9); 13->(
Dec:12, Dec:0); 13->(Dec:12, Dec:12); 16->(Dec:15, Dec:4); 16->(Dec:15, Dec:11);

***¿Deseas graficar los puntos sobre la curva S/N?: >s

Ahora abra octave y ejecuta:    octave:1>GraficaGF2n
```

Figura 5.3: Impresión de puntos sobre la curva $y^2 = x^3 + 8x^2 + 9$ para campos finitos $\mathbb{F}_{2^m}$.

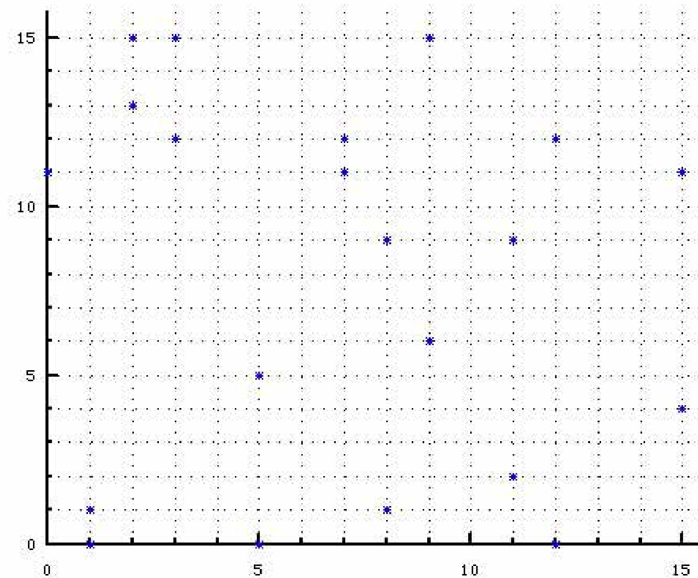


Figura 5.4: Gráfica de puntos sobre la curva $y^2 = x^3 + 8x^2 + 9$ para campos finitos $\mathbb{F}_{2^m}$.

### 5.3.3. Valores de la curva elíptica en $\mathbb{F}_{2^m}$

Para visualizar nuevamente los valores que se están utilizando sobre el campo, así como los valores de la ecuación de la curva elíptica y el polinomio que se está utilizando para generar dicho campo podemos seleccionar la opción 3, Figura 5.5.

```
*** GF(2^4)

***E: y^2 + x*y - [x^3 + Dec:8*x^2 + Dec:9]

***El campo finito binario esta formado por el polinomio irreducible:
                                x^4 + x + 1 mod 2

***Numero de combinaciones posibles sobre el campo: 16
```

Figura 5.5: Valores definidos sobre la curva elíptica.

### 5.3.4. Operaciones aritméticas sobre curvas elípticas en $\mathbb{F}_{2^m}$

Para realizar estas operaciones se toma como referencia el grupo de leyes para curvas elípticas en $\mathbb{F}_{2^m}$ descritas anteriormente. Se puede elegir primero la opción

7 que permite ver algunos puntos al azar que viven sobre la curva para realizar los cálculos. Figura 5.6.

```
Introduce la cantidad de numeros aleatorios que deseas: >>10
1. (Dec:5, Dec:0); 2. (Dec:0, Dec:11); 3. (Dec:8, Dec:1); 4. (Dec:3, Dec:15); 5. (Dec:12, Dec:0); 6. (Dec:15, Dec:
11); 7. (Dec:9, Dec:15); 8. (Dec:7, Dec:11); 9. (Dec:9, Dec:15); 10. (Dec:11, Dec:2);
```

Figura 5.6: Generación de puntos al azar.

### Sumar dos puntos sobre la curva elíptica

Para realizar la suma de dos puntos sobre la curva se utilizará (4.3), en donde se muestra como resultado un tercer punto que se encuentra sobre la curva, Figura 5.7.

```
*****Suma de Puntos Sobre La Curva Eliptica****
        R(x,y) = P(x,y) + Q(x,y)

Dame el valor de Px: 5
Dame el valor de Py: 5

Dame el valor de Qx: 2
Dame el valor de Qy: 13

        R(x,y) = (Dec:5, Dec:5) + (Dec:2, Dec:13) = (Dec:8, Dec:1)
```

Figura 5.7: Suma de puntos sobre la curva.

### Duplicar un punto sobre la curva elíptica

De igual manera para realizar esta operación se utiliza el grupo de leyes para calcular la operación de doblado. Figura 5.8.

```
*****Doblado de Puntos Sobre La Curva Eliptica****
        P(x,y) = 2P(x,y)

Dame el valor de Px: 7
Dame el valor de Py: 12
        2P(x,y) = (Dec:7, Dec:12) = (Dec:12, Dec:12)
```

Figura 5.8: Duplicar punto sobre la curva.

Estas dos operaciones resultan muy importantes al realizar la multiplicación escalar sobre un punto en la curva elíptica, ya que permiten calcular esta operación de una manera muy eficiente.

### Negativo de un punto sobre la curva elíptica

Para calcular el negativo de un punto se utiliza las leyes de grupo.

### 5.3.5. Multiplicación escalar

La ejecución de esta operación para calcular $R = kP$ se puede apreciar en la Figura 5.9

```
**Punto Base P: (Dec:12, Dec:0);
**Llave Privada K = 6;
2P = (Dec:11, Dec:9);
3P = (Dec:9, Dec:6);
6P = (Dec:15, Dec:4);
**Llave Publica Q = KP = (Dec:15, Dec:4);
```

Figura 5.9: Multiplicación escalar.

## 5.4. Resultados

En criptografía las únicas curvas elpticas sobre $\mathbb{F}_{2^m}$ de interés son las curvas No-Supersingulares como se definió en el capitulo 4.2.

### 5.4.1. Relación entre el orden de la curva elíptica y el orden del punto

Sabemos que el orden de la curva elíptica ($\#E_a(\mathbb{F}_{2^m})$) es el número total de puntos que existe sobre ella y el orden del punto $P$ es encontrar un numero $n$ tal que $nP = \vartheta$, $P \in E(\mathbb{F}_{2^m})$, donde $\vartheta$ es la identidad de la curva $E(\mathbb{F}_{2^m})$.
Se realizó la modificación del archivo 'AritmeticaCE_GF2n.cpp' del programa para saber el número total de puntos existentes en la curva elíptica definido sobre un campo $\mathbb{F}_{2^{23}}$ con los siguientes parámetros: $a = 0$ y $b = 1$. Entonces tenemos: $E_0 : y^2 + xy = x^3 + 1$ y $\#E_0(2) = 4$ que es el cofactor $h$. El programa nos devuelve que $\#E_0(\mathbb{F}_{2^{23}}) = 8383412$ y el punto P = (1030140, 5298230) $\in E(\mathbb{F}_{2^{23}})$ el cual tiene un orden de $n = 2095853$. Otra manera para calcular el orden de la curva podría ser de la siguiente forma:

$$\#E_0(\mathbb{F}_{2^m}) = hn = 4 * 2095853 = 8383412$$

Si ahora se define la siguiente curva $E_0 = y^2 + xy = x^3 + 1$ sobre el campo $\mathbb{F}_{2^{13}}$ y se toma como punto de referencia P = (8155, 2416) con orden de $n = 2003$ entonces podemos decir que el $\#E_0(\mathbb{F}_{2^{13}}) = 2003 * 4 = 8012$

También se puede verificar que si tomamos la curva $E_0 = y^2 + xy = x^3 + 1$ sobre el campo $\mathbb{F}_{2^{26}}$ el $\#E_0(\mathbb{F}_{2^{13}}) \mid \#E_0(\mathbb{F}_{2^{26}})$ ya que $13 \mid 26$. Esto lo podemos verificar si se toma el punto P = (44363973, 31988526) el cual tiene el orden $n = 16773122$ por lo tanto $\#E_0(\mathbb{F}_{2^{26}}) = 4 * 16773122 = 67092488$.

$$\#E_0(\mathbb{F}_{2^{13}}) \mid \#E_0(\mathbb{F}_{2^{26}}) = 8012 \mid 67092488$$

### 5.4.2. Pruebas para la multiplicación escalar sobre $E(\mathbb{F}_{2^m})$

El código correspondiente en Mathematica se muestra en el Apéndice D.

Mostramos con 3 ejemplos que el programa realiza de manera correcta la operación de la multiplicación escalar sobre $E(\mathbb{F}_{2^m})$ se realiza un pequeño programa en el software de Mathematica Versión 7, con los siguientes parámetros, para cada ejemplo.

**Primera prueba**

1. $\mathbb{F}_{2^{23}}$.
2. $E_0 = y^2 + xy = x^3 + 1$.
3. Llave privada 8388307, $k = 11111111111111011010011_2$.
4. Polinomio irreducible, $f(z) = z^{23} + z^5 + 1$.
5. Punto base, $G(7619787, 4097509)$.

El resultado nos da como polinomio las siguientes coordenadas:
$R = kG$
$Rx = z^{18} + z^{17} + z^{11} + z^7 + z^6 + z^3 + z^1 + 1$
$Ry = z^{22} + z^{21} + z^{16} + z^{14} + z^{12} + z^{10} + z^8 + z^6 + z^3 + z^2$

El programa en C++ muestra el mismo resultado al realizar la multiplicación escalar sobre $E(\mathbb{F}_{2^{23}})$ con los mismos parámetros, Figura 5.10.

```
                Selecciona una opcion:
 >>8

        0.Aleatoriamente
        1.Manualmente
   >1

**Punto Base P:
Dame el valor de Px: 7619787
Dame el valor de Py: 4097509
(Dec:7619787, Dec:4097509);
dame k: 8388307
**Llave Privada K = 8388307;
**Llave Publica R = KP = (Dec:395467, Dec:6378828);
```

Figura 5.10: Multiplicación escalar sobre $E(\mathbb{F}_{2^{23}})$.

**Segunda prueba**

1. $\mathbb{F}_{2^{89}}$.
2. $E_0 = y^2 + xy = x^3 + 1$.
3. Llave privada 6189700196426901374428954 46, $k = 1111111111111111111111111111-$

   $1111111111111111111111111111111111111 00110100100011001010110_2$.
4. Polinomio irreducible, $f(z) = z^{89} + z^{38} + 1$.
5. Punto base, $G(663576295626872044767 18272, 1600116813043305871354 26578)$.

El resultado nos da como polinomio las siguientes coordenadas:
$R = kG$
$Rx = z^{88} + z^{83} + z^{81} + z^{80} + z^{78} + z^{76} + z^{75} + z^{73} + z^{72} + z^{70} + z^{69} + z^{67} + z^{66} + z^{65} + z^{63} + z^{59} + z^{57} + z^{56} + z^{55} + z^{48} + z^{47} + z^{46} + z^{45} + z^{44} + z^{43} + z^{41} + z^{39} + z^{38} + z^{36} + z^{35} + z^{34} + z^{33} + z^{32} + z^{31} + z^{29} + z^{28} + z^{27} + z^{26} + z^{25} + z^{21} + z^{20} + z^{19} + z^{18} + z^{15} + z^{14} + z^{13} + z^{12} + z^{10} + z$

$Ry = z^{86} + z^{85} + z^{83} + z^{82} + z^{78} + z^{74} + z^{72} + z^{68} + z^{67} + z^{66} + z^{65} + z^{59} + z^{58} + z^{54} + z^{53} + z^{52} + z^{51} + z^{50} + z^{49} + z^{48} + z^{47} + z^{44} + z^{42} + z^{39} + z^{38} + z^{35} + z^{27} + z^{25} + z^{24} + z^{22} + z^{21} + z^{18} + z^{17} + z^{15} + z^{11} + z^{8} + z^{7} + z^{6} + z^{3} + z^{2} + 1$

El programa en C++ muestra el mismo resultado al realizar la multiplicación escalar sobre $E(\mathbb{F}_{2^{89}})$ con los mismos parámetros, Figura 5.11.

Figura 5.11: Multiplicación escalar sobre $E(\mathbb{F}_{2^{89}})$.

**Tercera prueba**

1. $\mathbb{F}_{2^{233}}$.
2. $E_0 = y^2 + xy = x^3 + 1$.
3. Llave privada 9954926935811275748695117245540509049022179443407731103250475934 89999, $k = 1001001110110011000100101100000000111011000111000011100100101000011101101100-$

$100111011111110100100111111101100011000001001101010100110010010011011010011100-$

$1010011011111111111111111111111111111111111111001101000101110101011001001111_2$

4. Polinomio irreducible, $f(z) = z^{233} + z^{74} + 1$.
5. Punto base, $G = (9980522611481012342443087688797002679043489582926858424680 3305-$

$54073382, 12814767389816757102953168016268660157166792010263439198493421287958179)$

El resultado nos da como polinomio las siguientes coordenadas:
$R = kG$
$Rx = 1 + z + z^2 + z^3 + z^4 + z^6 + z^9 + z^{10} + z^{13} + z^{15} + z^{16} + z^{21} + z^{24} + z^{25} + z^{27} + z^{31} + z^{33} + z^{35} + z^{36} + z^{40} + z^{44} + z^{47} + z^{49} + z^{50} + z^{54} + z^{56} + z^{58} + z^{62} + z^{63} + z^{64} + z^{65} + z^{69} + z^{70} + z^{71} + z^{74} + z^{75} + z^{77} + z^{79} + z^{81} + z^{83} + z^{84} + z^{85} + z^{87} + z^{88} + z^{92} + z^{93} + z^{94} + z^{95} + z^{96} + z^{98} + z^{99} + z^{101} + z^{104} + z^{106} + z^{107} + z^{112} + z^{113} + z^{114} + z^{116} + z^{117} + z^{121} + z^{123} + z^{125} + z^{128} + z^{130} + z^{131} + z^{134} + z^{135} + z^{138} + z^{139} + z^{144} + z^{147} + z^{149} + z^{151} + z^{152} + z^{153} + z^{156} + z^{158} + z^{159} + z^{162} + z^{163} + z^{165} + z^{167} + z^{169} + z^{170} + z^{171} + z^{172} + z^{174} + z^{175} + z^{176} + z^{177} + z^{180} + z^{181} + z^{182} + z^{184} + z^{185} + z^{188} + z^{190} + z^{191} + z^{192} + z^{193} + z^{194} + z^{195} + z^{199} + z^{200} + z^{202} + z^{203} + z^{204} + z^{205} + z^{206} + z^{208} + z^{209} + z^{210} + z^{213} + z^{215} + z^{217} + z^{218} + z^{219} + z^{220} + z^{222} + z^{223} + z^{225} + z^{227} + z^{228} + z^{231} + z^{232}$

$Ry = 1 + z + z^3 + z^5 + z^8 + z^9 + z^{10} + z^{14} + z^{16} + z^{17} + z^{18} + z^{21} + z^{24} + z^{27} + z^{29} + z^{31} + z^{33} + z^{35} + z^{36} + z^{37} + z^{38} + z^{39} + z^{41} + z^{42} + z^{44} + z^{47} + z^{50} + z^{53} + z^{57} + z^{59} + z^{60} + z^{61} + z^{63} + z^{65} + z^{66} + z^{68} + z^{69} + z^{71} + z^{75} + z^{76} + z^{77} + z^{82} + z^{85} + z^{87} + z^{91} + z^{92} + z^{93} + z^{94} + z^{95} + z^{97} + z^{99} + z^{100} + z^{101} + z^{102} + z^{105} + z^{106} + z^{108} + z^{110} + z^{113} + z^{115} + z^{116} + z^{117} + z^{118} + z^{120} + z^{123} + z^{124} + z^{125} + z^{130} + z^{132} + z^{133} + z^{134} + z^{136} + z^{139} + z^{141} + z^{142} + z^{143} + z^{145} + z^{147} + z^{149} + z^{150} + z^{153} + z^{156} + z^{157} + z^{158} + z^{159} + z^{162} + z^{163} + z^{165} + z^{167} + z^{168} + z^{170} + z^{172} + z^{173} + z^{174} + z^{175} + z^{176} + z^{177} + z^{180} + z^{182} + z^{184} + z^{185} + z^{187} + z^{190} + z^{192} + z^{193} + z^{194} + z^{197} + z^{198} + z^{199} + z^{200} + z^{202} + z^{203} + z^{206} + z^{208} + z^{209} + z^{213} + z^{214} + z^{215} + z^{220} + z^{222} + z^{224} + z^{226} + z^{228} + z^{229} + z^{231} + z^{232}$

El programa en C++ muestra el mismo resultado al realizar la multiplicación escalar sobre $E(\mathbb{F}_{2^{233}})$ con los mismos parámetros, Figura 5.12.

## 5.5. Implementación del segundo programa

Para la implementación del segundo programa se realizaron diferentes archivos (ver Apéndice B) el cual muestra la manera en que el par de llaves: públicas y privadas son generadas, a partir de los parámetros definidos sobre algún campo en especifico. El NIST (National Institute of Standards and Technology) recomienda algunos

```
           Selecciona una opcion:
>>8

      0.Aleatoriamente
      1.Manualmente
  >1

**Punto Base P:
Dame el valor de Px: 66357629562687204476718272
Dame el valor de Py: 160011681304330587135426578
(Dec:66357629562687204476718272, Dec:160011681304330587135426578);
dame k: 618970019642690137442895446
**Llave Privada K = 618970019642690137442895446;
**Llave Publica R = KP = (Dec:63144087525963461471859016, Dec:99013501230833757990584084);
```

Figura 5.12: Multiplicación escalar sobre $E(\mathbb{F}_{2^{233}})$.

parámetros para trabajar con curvas elípticas y realizar operaciones aritméticas sobre ellas [22].
La implementación del siguiente programa se utilizara el protocolo de Diffie-Hellman elíptico, descrito en el capitulo 4.5.

### 5.5.1. Implementación del servidor

Para poder comunicar diferentes sistemas de computo se emplea en este caso la tecnología llamada RPC †[1]. En el archivo 'servidorCCE.x' se describen las funciones necesarias para poder realizar la comunicación entre dos clientes que intercambiaran las llaves. Los parámetros necesarios para poder realizar el intercambio de llaves son los siguientes:

1. El número $m$: define el campo $\mathbb{F}_{2^m}$
2. $f(x)$: Polinomio irreducible de grado $m$
3. $a$, $b$: Valores de la ecuación $y^2 + xy = x^3 + ax^2 + b$
4. $G$: Un punto sobre la curva elíptica
5. $n$: El orden de la punto $G$
6. $h$: Cofactor de la curva

Estos parámetros son enviados a los clientes para trabajar sobre la misma curva elíptica y facilitar la interoperabilidad. Además de estos parámetros el servidor recibe los puntos $Q = k_1G$ y $R = k_2G$ de los clientes definidos con el nombre de 'host1' y 'host2' quienes estableceran la comunicación. Una vez que los clientes envían dichos puntos el servidor reenvía el punto $Q$ hacia 'host1' y el punto $R$ hacia el 'host2'. Los clientes realizaran las operaciones necesarias para obtener la llave en comun $S$. Una vez realizada la transacción el servidor permanece a la escucha para las siguientes peticiones realizando el mismo procedimiento para el intercambio de llaves.

[1]†RPC Remote Procedure Call, protocolo que permite ejecutar un programa desde una maquina remota sin preocuparse por la forma de comunicación

Una vez creado el archivo 'servidorCCE.x' con las funciones requeridas es necesario crear el archivo de cabecera 'servidorCCE.h'. Este se creara automáticamente al escribir el comando `$rpcgen servidorCCE.x` sobre la consola. La implementación de cada función de cabecera están definidas sobre el archivo servidorCCE.c, las cuales se describen a continuación:

`int * leer_archivo_1_svc(Parametros *` *operandos*`, struct svc_req *` *conexion*`)`

Esta función devuelve un valor entero 1 si los valores del archivo 'parametros' pueden ser leídos correctamente para la construcción del campo $\mathbb{F}_{2^m}$ y los valores necesarios para alguna curva elíptica en especifico. La función devuelve 0 en otro caso.

Las siguientes funciones devuelven al cliente los parámetros necesarios para generar las llaves:

`char ** campo_m_1_svc(Parametros *` *operandos*`, struct svc_req *` *conexion*`)`
  Regresa $m$ para definir el campo finito $\mathbb{F}_{2^m}$

`char ** constante_a_1_svc(Parametros *` *operandos*`, struct svc_req *` *conexion*`)`
  Regresa $a$ de la ecuación $y^2 + xy = x^3 + ax^2 + b$

`char ** constante_b_1_svc(Parametros *` *operandos*`, struct svc_req *` *conexion*`)`
  Regresa $b$ de la ecuación $y^2 + xy = x^3 + ax^2 + b$

`char ** punto_gx_1_svc(Parametros *` *operandos*`, struct svc_req *` *conexion*`)`
`char ** punto_gy_1_svc(Parametros *` *operandos*`, struct svc_req *` *conexion*`)`
  Regresan las coordenadas del punto $G \in E(\mathbb{F}_{2^m})$

`char ** cofactor_h_1_svc(Parametros *` *operandos*`, struct svc_req *` *conexion*`)`
  Regresa el cofactor $h \in \mathbb{F}_2$

`char ** primo_n_1_svc(Parametros *` *operandos*`, struct svc_req *` *conexion*`)`
  Regresan el numero $n$ que es el orden del punto $G$

### 5.5.2. Implementación de clientes

Para definir los clientes que realizaran la conexión a través de RPC utilizando el protocolo TCP se realiza el archivo 'conexion.cpp' en el cual está definida la siguiente función:

`CLIENT * conexion(char **`*server*`)`

La función recibe como parámetro la dirección IP del servidor, está a su vez llama la función:

```
clnt_create(*server, CCE_PROG, SERVIDORCCEVERS, tcp)
```

Esta llamada nos devuelve una estructura con toda la información necesaria para realizar la conexión mediante RPC. Los parámetros que admite son: La dirección IP del servidor, el nombre del programa y la versión que se definieron en el archivo 'servidor.x' y el tipo de protocolo para realizar la petición en este caso TCP.

Si la conexión es satisfactoria procedemos a llamar las funciones que posee el servidor, la primera función que se llama permite definir el campo finito $\mathbb{F}_{2^m}$ así como el resto de parámetros necesarios, esto asegura que los clientes estarán trabajando con los mismos valores.

```
int defineCampo(char *n_tupla[], CLIENT **conexion)
```

Como parámetros recibe *n_tupla* en el cual se almacenaran los valores de los parámetros necesarios para la transacción de las llaves, y el parámetro de conexión.

Una vez con los valores necesarios, cada cliente $c_1$ y $c_2$ genera como primera instancia la llave privada donde la implementación se encuentra en el archivo 'ecc_DH.cpp' mediante la llamada a la función:

```
bigint llavePrivadaK(char *n_tupla[])
```

Está función regresa de manera aleatoria un numero $k \in \mathbb{F}_2$, $k < n$, donde $n = o(G)$.

Como segunda instancia los clientes $c_1$ y $c_2$ generan una llave pública $Q_{c_1} = k_{c_1}G$ y $Q_{c_2} = k_{c_2}G \in \mathbb{F}_{2^m}$

```
void llavePublicaA(bigint k, char *n_tupla[], char Qx[], char Qy[])
```

La función realiza las operaciones aritméticas necesarias como la multiplicación escalar sobre curvas elípticas utilizando la adición y doblado como se definió en el primer programa y devolviendo la llave pública. El punto $Q_{c_x}$ es enviado por ambos clientes al servidor este a su vez reenvía el punto $Q_{c_1}$ al cliente $c_2$ y el punto $Q_{c_2}$ al cliente $c_1$ para que estos puedan generar una llave en común, donde cada cliente realiza nuevamente la multiplicación escalar con su llave privada, así los clientes estarán listos para intercambiar información utilizando algún algoritmo de cifrado.

## 5.6. Ejecución del segundo programa

Las operaciones sobre $\mathbb{F}_{2^m}$ se realizan por el lado del cliente lo cual implica el tener instaladas las bibliotecas necesarias para realizar dichas operaciones aritméticas.

Para configurar las direcciones IP de los sistemas de cómputo, podemos tener un servidor DHCP el cual nos asigna una dirección automáticamente en cada computadora y bastara con saber que dirección tenemos asignada con el comando `#ifconfig` sobre la consola. Si no es así, realizaremos una conexión Ethernet manualmente:

1. El servidor tendrá la dirección `#ifconfig ethx 192.168.1.2`

2. El host1 tendrá la dirección `#ifconfig ethx 192.168.1.3`

3. Finalmente al host 2 podemos asignarle `#ifconfig ethx 192.168.1.4`

Como lo muestra la figura 5.13.

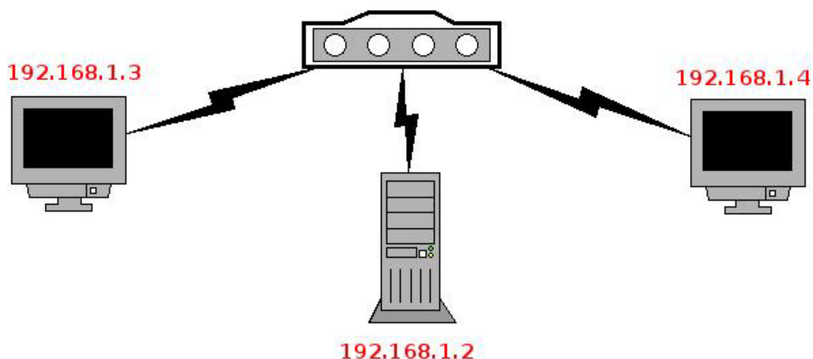


Figura 5.13: Conexión física de los sistemas de computo.

Para revisar que las conexiones estén correctas a cada sistema de cómputo podemos lanzar el comando `#ping IP` hacia las otras dos computadoras.
Para la ejecución del segundo programa realizamos la compilación con el comando $make, este creara tres archivos ejecutables el primero que es el servidor llamado 'servidorCCE' y los archivos 'host1' y 'host2' que funcionaran como los clientes.

Para poner en escucha al servidor simplemente basta ejecutar la siguiente instrucción sobre la consola `$./servidorCCE`, ahora el servidor estará listo para recibir la petición de los clientes, una vez que reciba alguna petición este enviara los parámetros necesarios, posteriormente los clientes envían los puntos $P$ y $R$ el cual el servidor los imprime en pantalla. De está manera cualquiera que sea ajeno a la comunicación

entre los clientes puede observar los mismos puntos que recibe el servidor como lo muestra la Figura 5.14.

Figura 5.14: Servidor.

Para que el cliente pueda realizar una conexión segura con un segundo cliente deberá realizar la petición hacia el servidor con el siguiente comando en consola `$./host1 192.168.1.2` de esta manera el cliente recibirá los valores necesarios para realizar la multiplicación escalar sobre curvas elípticas en campos finitos $\mathbb{F}_{2^m}$ como lo muestra la Figura 5.15, posteriormente este manda la llave pública que es interceptada por el servidor o por cualquier otra entidad ajena.

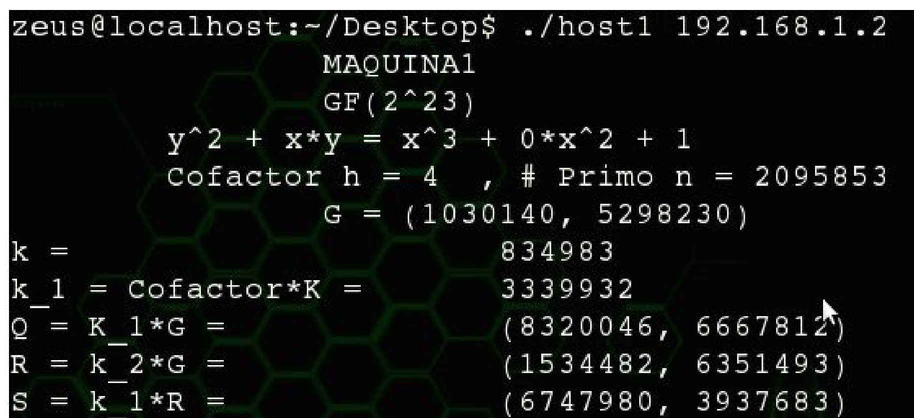


Figura 5.15: Host 1.

El segundo cliente realiza el mismo procedimiento sobre consola `$./host2 192.168.1.2`. Figura 5.16.

Al final los clientes tendrán una clave en común con el cual pueden realizar el cifrado de datos. El procedimiento general se resume de la siguiente manera:

El servidor envía a host1 y a host2 los valores de $m, a$, $b$, $n$, $h$ y $G \in E(\mathbb{F}_{2^m})$
host1 genera una llave privada al azar $k_1 \in \{1, n-1\}$.
host2 genera una llave privada al azar $k_2 \in \{1, n-1\}$.

```
rafa@localhost:~$ ./host2 192.168.1.2
               MAQUINA2
               GF(2^23)
        y^2 + x*y = x^3 + 0*x^2 + 1
        Cofactor h = 4  , # Primo n = 2095853
               G = (1030140, 5298230)
K =                     68327
k_2 = Cofactor*K =      273308
R = K_2*G =             (1534482, 6351493)
Q = K_1*G =             (8320046, 6667812)
S = k_2*Q =             (6747980, 3937683)
```

Figura 5.16: Host 2.

host1 calcula $Q = h * k_1 * G$.
host2 calcula $R = h * k_2 * G$.
host1 y host2 envían tanto $Q$ como $R$ respectivamente al servidor.
host1 recibe $R$ del servidor y calcula $S = h * k_1 * R$.
host2 recibe $Q$ del servidor y calcula $S = h * k_2 * Q$.

# Capítulo 6

# Aritmética en Hardware sobre campos finitos $\mathbb{F}_{2^m}$

### 6.0.1. Descripción de hardware

Para lograr un mejor rendimiento en la implementación de operaciones sobre curvas elípticas y para poder realizar la integración a nivel de hardware es necesario desarrollar las operaciones sobre campos finitos $\mathbb{F}_{2^m}$. Por lo que particularmente se implementaran dos operaciones: La multiplicación y la operación de elevar al cuadrado en lenguaje VHDL.

Para la codificación VHDL y la simulación se utiliza como referencia el dispositivo FPGA de Xilinx Virtex2P XC2VP30.

El FPGA es un circuito reprogramable que permite implementar diversos circuitos digitales en él. Está dividido en un gran número de bloques lógicos programables o también llamados `CLB o Slice` y se encuentran distribuidos a través de todo el chip con interconexiones programables, también cuenta con bloques de entrada y salida conocidas como `IOBs` en el cual se conectan las configuraciones internas con pines.

Los slices son los elementos esenciales del FPGA ya que en ellos se pueden implementar circuitos combinacionales como secuenciales y están agrupados de dos en dos o de cuatro en cuatro formando así los bloques lógicos configurables (CLBs). Dentro de los CLBs se encuentran los módulos LUTs registros y multiplexores programables.

Los elementos más importantes son los generadores reprogramables de función lógica realizadas por las LUTs (Look-Up-Table) o tablas de búsqueda que son celdas de SRAM y multiplexores para seleccionar la salida.

### 6.0.2. Representación base polinomial

Un campo finito de la forma $\mathbb{F}_{2^m}$ es tal que $\#(\mathbb{F}^*_{2^m}) = 2^m - 1$, donde $\mathbb{F}^*_{2^m} = \mathbb{F}_{2^m} \setminus \{0\}$ [14]. Un elemento arbitrario $A \in \mathbb{F}_{2^m}$ es denotado por un polinomio de grado menor a $m$, tal que:

$$A = a(z) = a_{m-1}z^{m-1} + \ldots + a_1 z + a_0 = \sum_{i=0}^{m-1} a_i z^i, a_i \in \{0,1\} \tag{6.1}$$

Donde $a(z)$ se conoce como la representación en base polinomial de $A \in \mathbb{F}_{2^m}$.

Debido a la representación binaria de elementos $x \in \mathbb{F}_{2^m}$, nos lleva a la motivación de poder realizar operaciones aritméticas en hardware, trabajando particularmente sobre un campo finito $\mathbb{F}_{2^{233}}$ ya que se considera suficiente para crear llaves públicas seguras[11].

La multiplicación resulta ser una de las operaciones más costosas sobre campos finitos (la otra operación es la inversión modular), por lo cual se utilizará el algoritmo de Karatsuba-Ofman para reducir el costo de ejecución de dicha operación.

### 6.0.3. Multiplicación sobre campos finitos $\mathbb{F}_{2^m}$

Sean $A = a(z) = (a_{m-1}, \ldots, a_1, a_0)$ y $B = b(z) = (b_{m-1}, \ldots, b_1, b_0) \in \mathbb{F}_{2^m}$ y sea $C = c(z)$ la multiplicación de los dos polinomios, entonces:

$$A \cdot B = C = c(z) = (c_{m-1}, \ldots,\ c_1, c\ _0)$$

donde $c(z) \in \mathbb{F}_{2^m}$ que es el resultado de multiplicar $a(z) \cdot b(z)\ mod\ p(z)$, donde $p(z)$ es el polinomio irreducible de grado $m$. Ya que $\mathbb{F}_{2^m} \cong \frac{\mathbb{F}_2[z]}{p(z)}$, con $p(z)$ polinomio irreducible en $\mathbb{F}_2[Z]$ de grado $m$.

Para poder obtener $c(z)$, podemos obtener primero el producto polinomial $c'(z)$

$$c'(z) = a(z) \cdot b(z) = \left[\sum_{i=0}^{m-1} a_i z^i\right] \cdot \left[\sum_{i=0}^{m-1} b_i z^i\right] \tag{6.2}$$

que tiene grado a lo más $2m - 2$.

Posteriormente se realiza la operación de reducción para así obtener $c(z)$ que es de grado a lo más $m - 1$ mediante el polinomio irreducible $p(z)$, obteniendo:

$$c(z) = c'(z)\ mod\ p(z) \tag{6.3}$$

Para poder realizar la multiplicación, primero se calculará $c'(z)$ con el algoritmo de Karatsuba-Ofman y posteriormente se calculará la reducción $mod\ p(z)$ donde $p(z)$ sera el polinomio irreducible $z^{233} + z^{74} + 1 \in \mathbb{F}_{2^{233}}$, a través de corrimientos y la operación lógica XOR se obtendrá finalmente $c(z)$.

### 6.0.4. Algoritmo de Karatsuba-Ofman

El algoritmo de Karatsuba-Ofman (AKO)[23] se basa en la observación siguiente: Considere el campo finito $\mathbb{F}_{2^m} \cong \frac{\mathbb{F}_2[z]}{p(z)}$, donde $p(z)$ es el polinomio irreducible de grado $m = rn$ en $\mathbb{F}_2[z]$, donde $r = 2^k$, $k \in \mathbb{Z}^+$ y sean $A$ y $B$ dos elementos $\in \mathbb{F}_{2^m}$. Los elementos $A, B$ se puede representar en base polinomial de la siguiente manera:

$$A = \sum_{i=0}^{m-1} a_i z^i = \sum_{i=\frac{m}{2}}^{m-1} a_i z^i + \sum_{i=0}^{\frac{m}{2}-1} a_i z^i = z^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} a_{i+\frac{m}{2}} z^i + \sum_{i=0}^{\frac{m}{2}-1} a_i z^i = z^{\frac{m}{2}} A_H + A_L \tag{6.4}$$

y

$$B = \sum_{i=0}^{m-1} b_i z^i = \sum_{i=\frac{m}{2}}^{m-1} b_i z^i + \sum_{i=0}^{\frac{m}{2}-1} b_i z^i = z^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} b_{i+\frac{m}{2}} z^i + \sum_{i=0}^{\frac{m}{2}-1} b_i z^i = z^{\frac{m}{2}} B_H + B_L \quad (6.5)$$

Usando las ecuaciones 6.4 y 6.5, el producto polinomial es:

$$C = z^m A_H B_H + (A_H B_L + A_L B_H) z^{\frac{m}{2}} + A_L B_L \quad (6.6)$$

Siguiendo el algoritmo de Karatsuba-Ofman la ecuación 6.6 se puede escribir de la siguiente forma:

$$C = z^m A_H B_H + (A_H B_H + A_L B_L + (A_L - A_H)(B_H - B_L)) z^{\frac{m}{2}} + A_L B_L \quad (6.7)$$

Como estamos trabajando sobre $\mathbb{F}_{2^m}$, la sustración es equivalente a la adición así que podemos sobreescribir la ecuación 6.7 como:

$$C = z^m A_H B_H + (A_H B_H + A_L B_L + (A_L + A_H)(B_H + B_L)) z^{\frac{m}{2}} + A_L B_L = z^m C_H + C_L \quad (6.8)$$

En la ecuación 6.8 se utilizan cuatro sumas y tres multiplicaciones, mientras que en la ecuación 6.6 se utilizan cuatro multiplicaciones y tres sumas. Para realizar las operaciones de suma en campos finitos $\mathbb{F}_{2^m}$, solo se necesita realizar operaciones lógicas XOR. Se sabe que la multiplicación es muy costosa por lo que se utilizará la ecuación 6.8 para realizar dicha operación. Los algoritmos clásicos suelen ser de $O(m^2)$ por tal motivo se decidió trabajar con el algoritmo de Karatsuba-Ofman ya que es de orden $O(m^{\log_2 3})$.

Para cada elemento $A \in \mathbb{F}_{2^{233}}$, en la implementación de la multiplicación será dividido en varios módulos hasta obtener multiplicadores de 4 bits, como se muestra en la figura 6.1.

Gracias a la independencia entre las multiplicaciones es posible realizar el proceso de paralelización en el multiplicador de Karatsuba-Ofman. Una vez realizadas las tres multiplicaciones internas es posible terminar la multiplicación de m bits a través de concatenaciones y sumas presentadas en la Figura 6.2.

Este método es muy eficiente en multiplicaciones de $2^n$ bits sin embargo se esta desperdiciando 23 bits. Para realizar un multiplicador de 233 bits exactos se deja a discusión, con lo cual se despreciaran los bits mas significativos de la multiplicación.

### 6.0.5. Reducción modular

El resultado de la multiplicación entre dos polinomios de grado $m-1$ es a lo mas un polinomio de grado $2m-2$, por lo que es necesario realizar una reducción

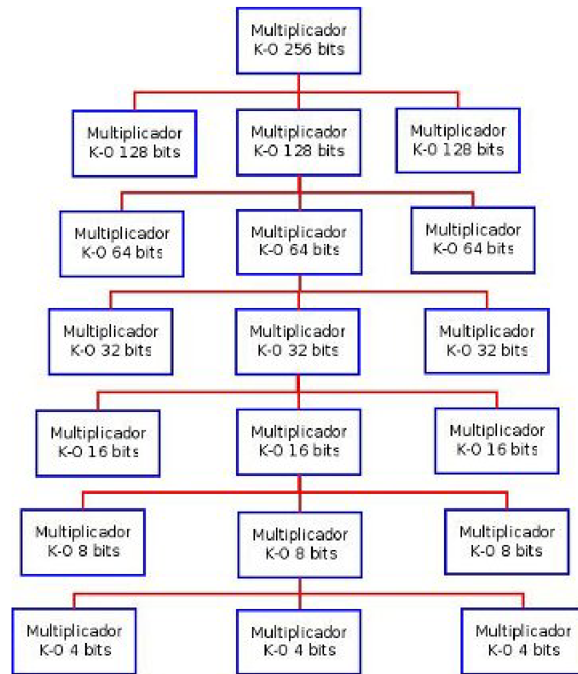


Figura 6.1: Diagrama de bloques para la multiplicación Karatsuba-Ofman.

modular para satisfacer la aritmética de campos finitos y así obtener un polinomio de grado $m-1$. Para el caso particular del campo finito $\mathbb{F}_{2^{233}}$ se utilizará el polinomio irreducible $p(z) = z^{233} + z^{74} + 1$.

El algoritmo de reducción modular en campos finitos está basado en la observación:

$$C(z) = c_{2m-2}z^{2m-2} + \ldots + c_m z^m + c_{m-1}z^{m-1} + \ldots + c_1 z + c_0 \tag{6.9}$$

$$\equiv (c_{2m-2}z^{2m-2} + \ldots + c_m)r(z) + c_m z^m + c_{m-1}z^{m-1} + \ldots + c_1 z + c_0 (mod\ p(z)). \tag{6.10}$$

La ecuación 6.10 sugiere un algoritmo eficiente para realizar la reducción modular de un número $C \in \mathbb{F}_{2^m}$ si este tiene a lo mas $2m-2$ bits. Para implementar el algoritmo se realizaran con operaciones simples de superposiciones de bits y mediante la operación lógica XOR (adición sobre campos finitos). Debido a que el polinomio irreducible en el campo finito $\mathbb{F}_{2^{233}}$ es un trinomio, el proceso de reducción resulta ser eficiente. La figura 6.3 muestra el proceso de reducción de un polinomio conformado por $2m-2$ bits, para un $m = 233$ y con el polinomio irreducible $p(z) = z^{233} + z^{74} + 1$.

El diseño completo para realizar la multiplicación se muestra mediante la figura 6.4, donde se observa la distribución de los tres multiplicadores de Karatsuba-Ofman como primera parte y posteriormente se realiza la reducción modular obteniendo un elemento de $\mathbb{F}_{2^m}$.

Para desarrollar el multiplicador Karatsuba-Ofman se utilizará el algoritmo 8.

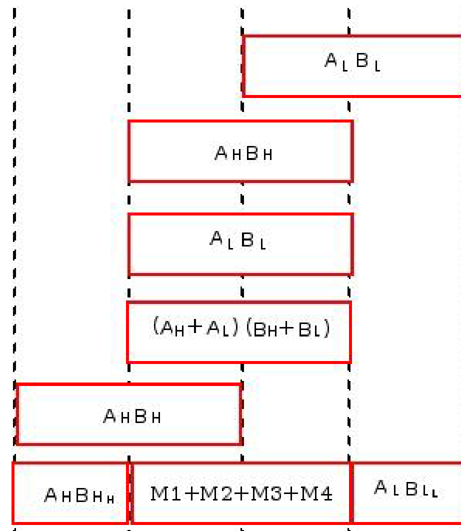


Figura 6.2: Proceso de paralelización para la multiplicación Karatsuba-Ofman.

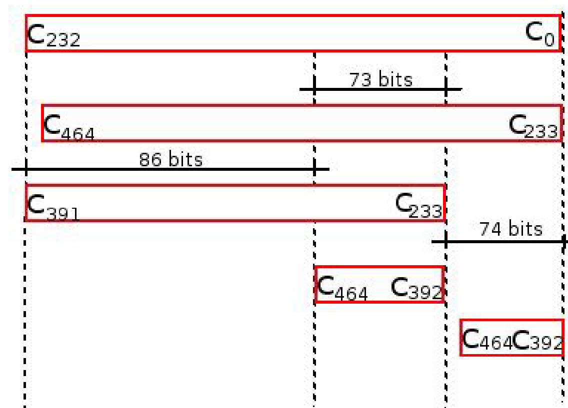


Figura 6.3: Reducción modular: $z^{233} + z^{74} + 1$.

### 6.0.6. Elevar al cuadrado

Elevar al cuadrado un elemento $A \in \mathbb{F}_{2^m}$, resulta una operación sencilla ya que podemos realizar la operación de la siguiente manera:

Sea $A = a(z) = \sum_{i=0}^{m-1} a_i z^i \in \mathbb{F}_{2^m}$ entonces tenemos que: $A^2 = a(z)^2 = \sum_{i=0}^{m-1} a_i z^{2i}$

La forma más directa de implementar la operación aritmética de elevar al cuadrado, es empleando la definición de multiplicación polinomial que consiste como primer paso calcular el producto $c'(z) = a(z) \cdot a(z)$ y como segundo paso aplicar la reducción modular $c(z) = c'(z) \bmod p(z)$ mediante el polinomio reducible. Aunque para calcular $A^2$ solo basta insertar ceros del lado derecho de cada bit del elemento como se muestra en la Figura: 6.5, posteriormente se realizaría la reducción modular.

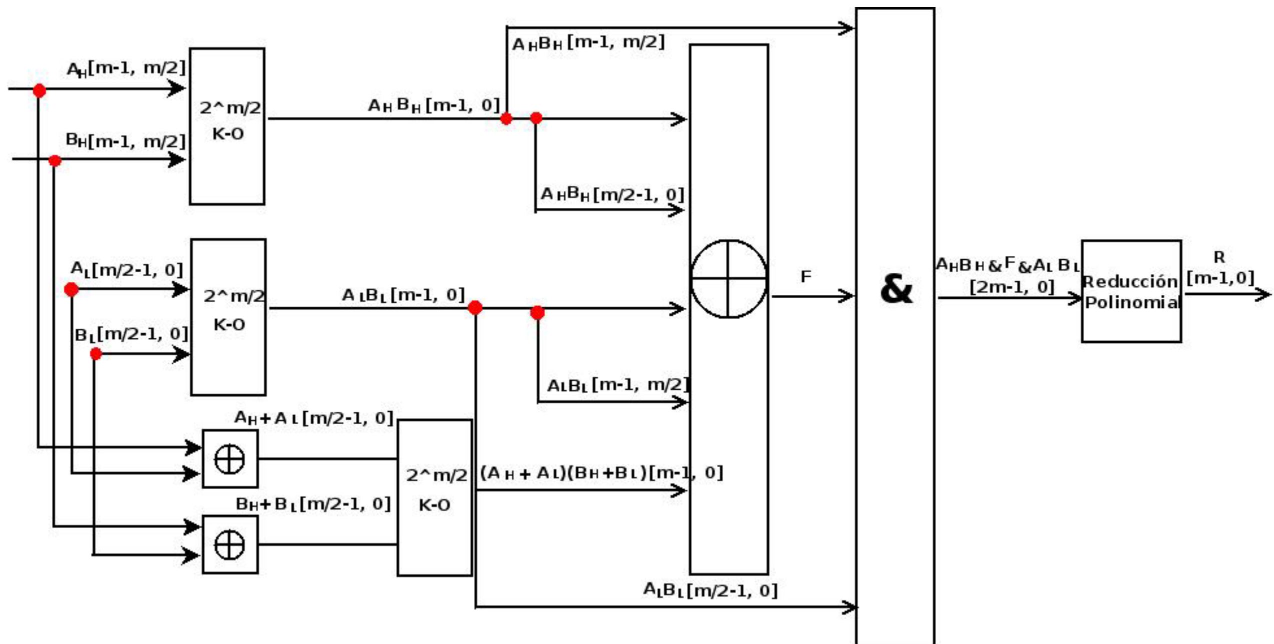


Figura 6.4: Arquitectura completa para la multiplicación sobre $\mathbb{F}_{2^m}$.

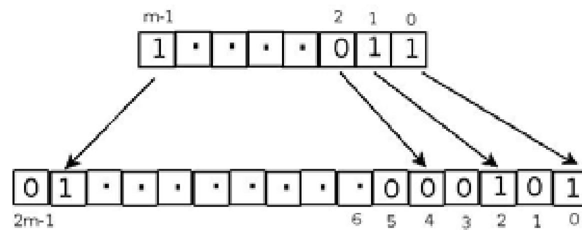


Figura 6.5: Elevar al cuadrado un elemento $a \in \mathbb{F}_{2^m}$.

Debido a que la mitad de la representación binaria de $A^2$ esta compuesta de ceros, esta propiedad puede ser aprovechada en el proceso de reducción modular como lo muestra el algoritmo 9.

## 6.0.7. Implementación de multiplicación en campos finitos $\mathbb{F}_{2^m}$

El código correspondiente puede consultarse en el Apéndice C.

### Multiplicador de 4 bits

Como primera parte se desarrollo un multiplicador de 4 bits en el lenguaje VHDL que funcionará como base para realizar el multiplicador final. La implementación se

**Algoritmo 8** Karatsuba-Ofman

**Entrada:** A, B y C
**Salida:** C = AB
1: **si** m=4 **entonces**
2: C = $mul_m$(A, B)
3: **regresa**
4: **fin si**
5: **para** $i = 0$ **hasta** $\frac{m}{2} - 1$ **hacer**
6: $M_A i = A_{L_i} + A_{H_i}$
7: $M_B i = B_{L_i} + B_{H_i}$
8: **fin para**
9: $mul2^k(C_L,A_L,B_L)$
10: $mul2^k(M,M_A,M_B)$
11: $mul2^k(C_H,A_H,B_H)$
12: **para** $i = 0$ **hasta** $m - 1$ **hacer**
13: $M_i = M_i + C_{L_i} + C_{H_i}$
14: **fin para**
15: **para** $i = 0$ **hasta** $m - 1$ **hacer**
16: $M_{\frac{m}{2}+i} = C_{\frac{m}{2}+i} + M_i$
17: **fin para**
18: **regresa** $C(z)$

**Algoritmo 9** $a(z)^2 \bmod z^{233} + z^{74} + 1$ [24]

**Entrada:** $a(z) = \sum_{i=0}^{m-1} a_i z^i$
**Salida:** $a(z)^2 \bmod z^{233} + z^{74} + 1$
**para** los bits $i = 0$ **hasta** 73 **hacer**
2: **si** $i$ es par **entonces** $c_i = a_{\frac{i}{2}}$ XOR $a_{\frac{i+392}{2}}$
**otro** $c_i = a_{\frac{i+233}{2}}$
4: **fin si**
**fin para**
6: **para** los bits $i = 74$ **hasta** 146 **hacer**
**si** $i$ es par **entonces** $c_i = a_{\frac{i}{2}}$ XOR $a_{\frac{i+318}{2}}$
8: **otro** $c_i = a_{\frac{i+233}{2}}$ XOR $a_{\frac{i+159}{2}}$
**fin si**
10: **fin para**
**para** los bits $i = 147$ **hasta** 232 **hacer**
12: **si** $i$ es par **entonces** $c_i = a_{\frac{i}{2}}$
**otro** $c_i = a_{\frac{i+233}{2}}$ XOR $a_{\frac{i+159}{2}}$
14: **fin si**
**fin para**
16: **regresa** $C(z) = \sum_{i=0}^{m-1} c_i z^i$

encuentra sobre el archivo 'GF2n.vhdl', en donde se encuentra la función:

function M4b ($a$, $b$: biti_vector(3 downto 0)) return bit_vector
$c = a * b \in \mathbb{F}_2$ de 4 bits

Esta función fue desarrollada aplicando la idea de Karatsuba-Ofman como se definió anteriormente y recibe como parámetros de entrada $a$ y $b$ que son de 4 bits y devuelve la multiplicación en $c$ el cual utiliza a lo mas 7 bits para realizar la multiplicación.

La multiplicación de 4 bits puede observarse en la Figura 6.6.

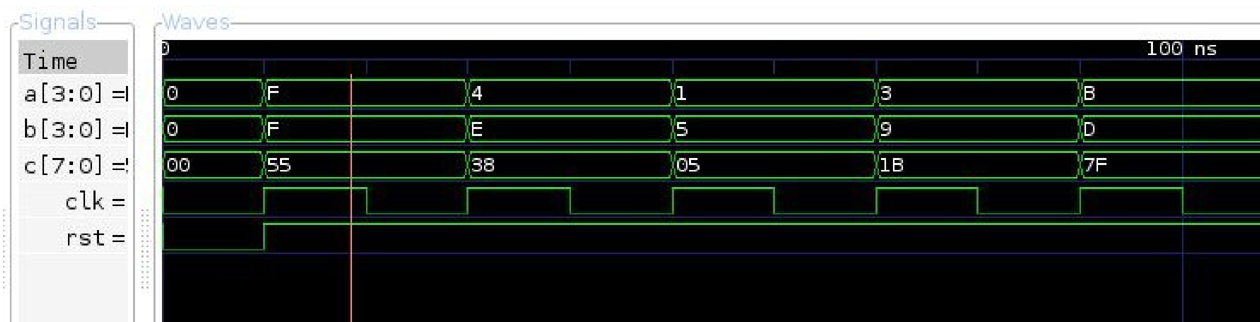


Figura 6.6: Multiplicador K-O de 4 bits.

Las primeras cinco entradas y las salidas del multiplicador de 4 bits corresponden a los siguientes polinomios:

$$a = F_{16} = 1111_2 = z^3 + z^2 + z + 1$$
$$b = F_{16} = 1111_2 = z^3 + z^2 + z + 1$$
$$c = 55_{16} = 1010101_2 = z^6 + z^4 + z^2 + 1$$

$$a = 4_{16} = 100_2 = z^2$$
$$b = E_{16} = 1110_2 = z^3 + z^2 + z$$
$$c = 38_{16} = 111000_2 = z^5 + z^4 + z^3$$

$$a = 1_{16} = 1_2 = 1$$
$$b = 5_{16} = 101_2 = z^2 + 1$$
$$c = 05_{16} = 101_2 = z^2 + 1$$

$$a = 3_{16} = 11_2 = z + 1$$
$$b = 9_{16} = 1001_2 = z^3 + 1$$
$$c = 1B_{16} = 11011_2 = z^4 + z^3 + z + 1$$

$$a = B_{16} = 1011_2 = z^3 + z + 1$$
$$b = D_{16} = 1101_2 = z^3 + z^2 + 1$$
$$c = 7F_{16} = 1111111_2 = z^6 + z^5 + z^4 + z^3 + z^2 + z + 1$$

### Multiplicador K-O de 8 bits

Para realizar el multiplicador de 8 bits se utiliza recursividad, es decir; el multiplicador de 4 bits será utilizado para desarrollar este multiplicador.
La función que se utiliza se define de la siguiente manera:

function M8b ($a$, $b$: biti_vector(7 downto 0)) return bit_vector
$c = a * b \in \mathbb{F}_2$ de 8 bits

Esta función recibe dos elementos $a$ y $b \in \mathbb{F}_2$ de 8 bits, dentro de la función se llama el multiplicador de 4 bits como sigue:

CH := M4b($a$(7 downto 4), $b$(7 downto 4))

La cual representa la multiplicación de la parte alta $A_H * B_H$.

CL := M4b($a$(3 downto 0), $b$(3 downto 0))

Representa la multiplicación de la parte baja $A_L * B_L$

CM:=M4b(($a$(7 downto 4)XOR $a$(3 downto 0)),($b$(7 downto 4)XOR $b$(3 downto 0)))

La multiplicación de la parte media $(A_H + A_L)(B_H + B_L)$.

Para obtener finalmente la multiplicación mod 2 se realiza el barrido con operaciones XOR como se mostró en la Figura 6.2.

Los barridos de 8 bits se muestran a continuación:

```
BHAL := CH(15 downto 0) & CL(31 downto 16)
```

Concatenación de la parte alta de CL con la parte baja de CH

```
M := ((CH xor CL) xor CM) xor BHAL
```

Sumatoria de M1 + M2 + M3 + M4

```
karatsuba := CH(31 downto 16) & M & CL(15 downto 0)
```

Finalmente se realizan las concatenaciones de los barridos y se devuelve el Multiplicador de Karatsuba-Ofman. La simulación del multiplicador de 8 bits se puede observar en la Figura 6.7

Las entradas mostradas en la Figura 6.7 corresponden a los siguientes polinomios:

$a = F6_{16} = 11110110_2 = z^7 + z^6 + z^5 + z^4 + z^2 + z$
$b = F1_{16} = 11110001_2 = z^7 + z^6 + z^5 + z^4 + 1$
$c = 57D6_{16} = 101011111010110_2 = z^{14} + z^{12} + z^{10} + z^9 + z^8 + z^7 + z^6 + z^4 + z^2 + z$

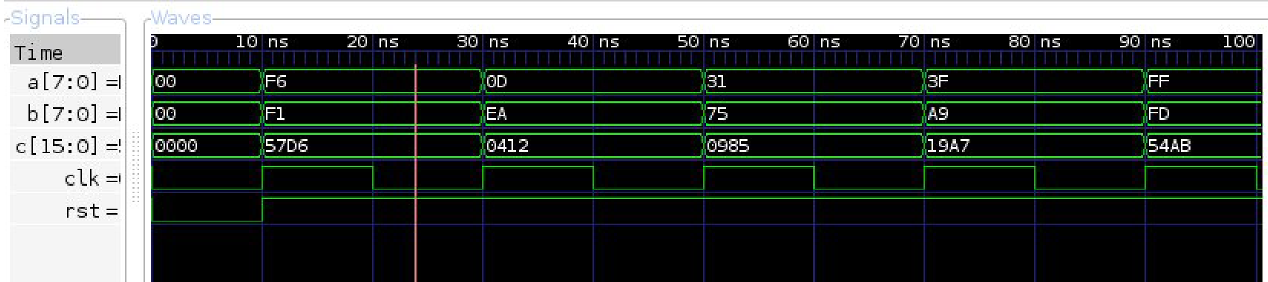


Figura 6.7: Multiplicador K-O de 8 bits.

$a = 0D_{16} = 1101_2 = z^3 + z^2 + 1$
$b = EA_{16} = 11101010_2 = z^7 + z^6 + z^5 + z^3 + z$
$c = 0412_{16} = 10000010010_2 = z^{10} + z^4 + z$

$a = 31_{16} = 110001_2 = z^5 + z^4 + 1$
$b = 75_{16} = 1110101_2 = z^6 + z^5 + z^4 + z^2 + 1$
$c = 0985_{16} = 100110000101_2 = z^{11} + z^8 + z^7 + z^2 + 1$

$a = 3F_{16} = 111111_2 = z^5 + z^4 + z^3 + z^2 + z + 1$
$b = A9_{16} = 10101001_2 = z^7 + z^5 + z^3 + 1$
$c = 19A7_{16} = 1100110100111_2 = z^{12} + z^{11} + z^8 + z^7 + z^5 + z^2 + z + 1$

$a = FF_{16} = 11111111_2 = z^7 + z^6 + z^5 + z^4 + z^3 + z^2 + z + 1$
$b = FD_{16} = 11111101_2 = z^7 + z^6 + z^5 + z^4 + z^3 + z^2 + 1$
$c = 54AB_{16} = 101010010101011_2 = z^{14} + z^{12} + z^{10} + z^7 + z^5 + z^3 + z + 1$

De la misma manera se desarrollan los demás multiplicadores hasta llegar con el multiplicador de 128 bits.

La tabla 6.1 muestra los diferentes resultados obtenidos a partir de los multiplicadores de Karatsuba-Ofman: En la tabla se pueden apreciar los recursos del FPGA en slices, el número de bits de cada multiplicador y el tiempo necesario para generar la salida.

### Multiplicador K-O de 233 bits

Para la implementación del multiplicador de 233 bits se utiliza el multiplicador de 256 despreciando los bits mas significativos, de esta manera solo se toma los 465 bits

| Multiplicador No. Bits | No. Slices | No. LUTs de 4 entradas | Tiempo en ns |
|---|---|---|---|
| 4 | 6 | 11 | 2.653 |
| 8 | 28 | 49 | 4.214 |
| 16 | 100 | 175 | 5.909 |
| 32 | 339 | 592 | 7.608 |
| 64 | 1078 | 1882 | 9.052 |
| 128 | 3411 | 5954 | 10.742 |

Tabla 6.1: Costo de multiplicadores de n bits

como resultado final al aplicar el algoritmo Karatsuba-Ofman. Algunos resultados pueden observarse en las siguientes Figuras:

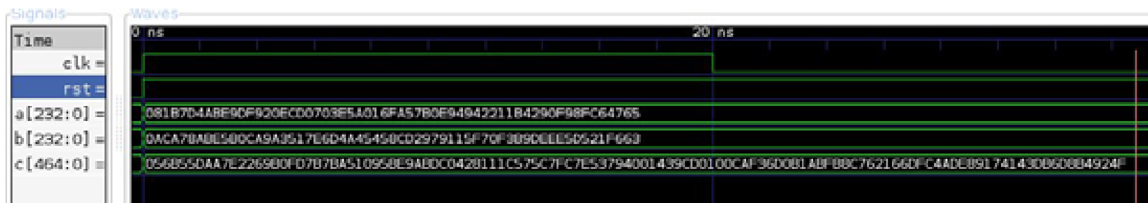


Figura 6.8: Multiplicador K-O de 233 bits.

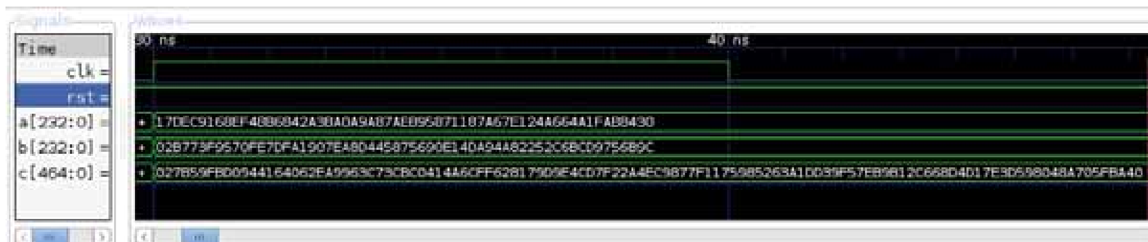


Figura 6.9: Multiplicador K-O de 233 bits.

### Residuo modular de 233 bits

Para realizar el residuo modular se basa en la estrategia descrita en: 6.3.

**Primera prueba:**

Entradas:

Figura 6.10: Multiplicador K-O de 233 bits.

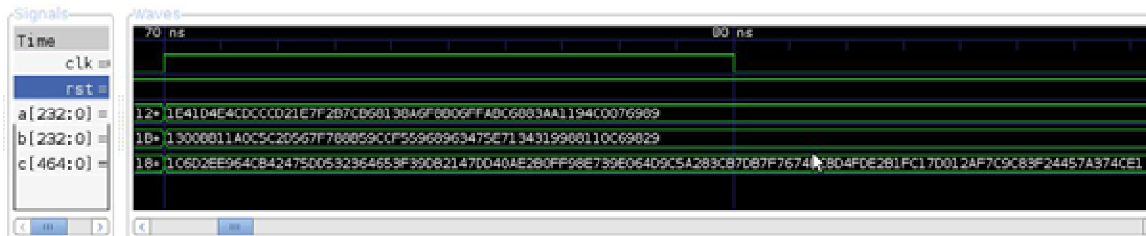


Figura 6.11: Multiplicador K-O de 233 bits.

a $= 081B7D4ABE9DF920ECD0703E5A016FA57B0E94942211B4290F98FC64765_{16}$
b $= 0ACA78ABE5B0CA9A3517E6D4A45458CD2979115F70F3B9DEEE5D521F663_{16}$

Estas entradas $a, b$ equivalen a los siguientes polinomios:

a $= z^{231} + z^{224} + z^{223} + z^{221} + z^{220} + z^{218} + z^{217} + z^{216} + z^{215} + z^{214} + z^{212} + z^{210} + z^{207} + z^{205} + z^{203} + z^{201} + z^{200} + z^{199} + z^{198} + z^{197} + z^{195} + z^{192} + z^{191} + z^{190} + z^{188} + z^{187} + z^{186} + z^{185} + z^{184} + z^{183} + z^{180} + z^{177} + z^{171} + z^{170} + z^{169} + z^{167} + z^{166} + z^{163} + z^{162} + z^{160} + z^{154} + z^{153} + z^{152} + z^{145} + z^{144} + z^{143} + z^{142} + z^{141} + z^{138} + z^{136} + z^{135} + z^{133} + z^{124} + z^{122} + z^{121} + z^{119} + z^{118} + z^{117} + z^{116} + z^{115} + z^{113} + z^{110} + z^{108} + z^{106} + z^{105} + z^{104} + z^{103} + z^{101} + z^{100} + z^{95} + z^{94} + z^{93} + z^{91} + z^{88} + z^{86} + z^{83} + z^{80} + z^{78} + z^{73} + z^{69} + z^{64} + z^{60} + z^{59} + z^{57} + z^{56} + z^{54} + z^{49} + z^{47} + z^{44} + z^{39} + z^{38} + z^{37} + z^{36} + z^{35} + z^{32} + z^{31} + z^{27} + z^{26} + z^{25} + z^{24} + z^{23} + z^{22} + z^{18} + z^{17} + z^{14} + z^{10} + z^{9} + z^{8} + z^{6} + z^{5} + z^{2} + 1$

b $= z^{231} + z^{229} + z^{227} + z^{226} + z^{223} + z^{221} + z^{218} + z^{217} + z^{216} + z^{215} + z^{211} + z^{209} + z^{207} + z^{205} + z^{204} + z^{203} + z^{202} + z^{201} + z^{198} + z^{196} + z^{195} + z^{193} + z^{192} + z^{187} + z^{186} + z^{183} + z^{181} + z^{179} + z^{176} + z^{175} + z^{173} + z^{169} + z^{168} + z^{166} + z^{164} + z^{160} + z^{158} + z^{157} + z^{156} + z^{155} + z^{154} + z^{153} + z^{150} + z^{149} + z^{147} + z^{146} + z^{144} + z^{142} + z^{139} + z^{137} + z^{134} + z^{130} + z^{128} + z^{126} + z^{122} + z^{120} + z^{119} + z^{115} + z^{114} + z^{111} + z^{110} + z^{108} + z^{105} + z^{103} + z^{100} + z^{98} + z^{97} + z^{96} + z^{95} + z^{92} + z^{88} + z^{84} + z^{82} + z^{80} + z^{79} + z^{78} + z^{77} + z^{76} + z^{74} + z^{73} + z^{72} + z^{67} + z^{66} + z^{65} + z^{64} + z^{61} + z^{60} + z^{59} + z^{57} + z^{56} + z^{55} + z^{52} + z^{51} + z^{50} + z^{48} + z^{47} + z^{46} + z^{45} + z^{43} + z^{42} + z^{41} + z^{39} + z^{38} + z^{37} + z^{34} + z^{32} + z^{31} + z^{30} + z^{28} + z^{26} + z^{24} + z^{21} + z^{16} + z^{15} + z^{14} + z^{13} + z^{12} + z^{10} + z^{9} + z^{6} + z^{5} + z + 1$

Resultado:

c = $0DCEE78240460AE952E8915505BA1D82BF424BDDC4F23F37436C5F49E12_{16}$

$c$ es equivalente al siguiente polinomio:
c = $z^{231}+z^{230}+z^{228}+z^{227}+z^{226}+z^{223}+z^{222}+z^{221}+z^{219}+z^{218}+z^{217}+z^{214}+z^{213}+z^{212}+z^{211}+z^{205}+z^{202}+z^{194}+z^{190}+z^{189}+z^{183}+z^{181}+z^{179}+z^{178}+z^{177}+z^{175}+z^{172}+z^{170}+z^{168}+z^{165}+z^{163}+z^{162}+z^{161}+z^{159}+z^{155}+z^{152}+z^{148}+z^{146}+z^{144}+z^{142}+z^{140}+z^{134}+z^{132}+z^{131}+z^{129}+z^{128}+z^{127}+z^{125}+z^{120}+z^{119}+z^{118}+z^{116}+z^{115}+z^{109}+z^{107}+z^{105}+z^{104}+z^{103}+z^{102}+z^{101}+z^{100}+z^{98}+z^{93}+z^{90}+z^{87}+z^{85}+z^{84}+z^{83}+z^{82}+z^{80}+z^{79}+z^{78}+z^{76}+z^{75}+z^{74}+z^{70}+z^{67}+z^{66}+z^{65}+z^{64}+z^{61}+z^{57}+z^{56}+z^{55}+z^{54}+z^{53}+z^{52}+z^{49}+z^{48}+z^{46}+z^{45}+z^{44}+z^{42}+z^{37}+z^{36}+z^{34}+z^{33}+z^{31}+z^{30}+z^{26}+z^{24}+z^{23}+z^{22}+z^{21}+z^{20}+z^{18}+z^{15}+z^{12}+z^{11}+z^{10}+z^{9}+z^{4}+z$

Figura 6.12

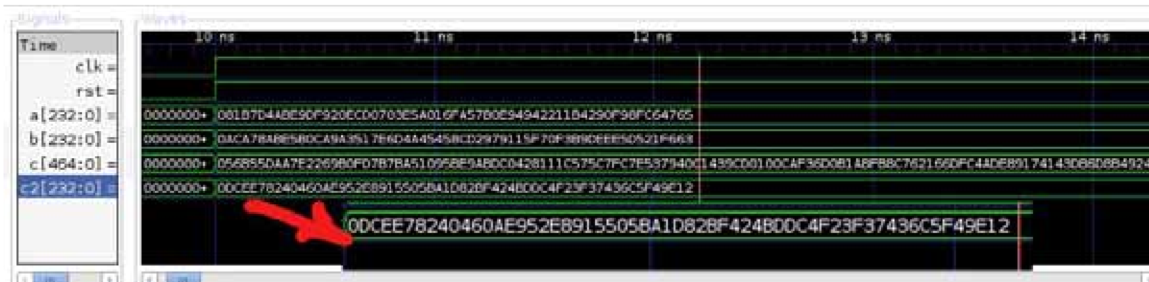


Figura 6.12: Multiplicador de 233 bits.

**Segunda prueba:**

Entradas:

a = $17DEC9168EF4BB6842A3BA0A9A87AEB95871187A67E124A664A1FAB8430_{16}$
b = $02B773F9570FE7DFA1907EA8D445875690E14DA94A82252C6BCD9756B9C_{16}$

Estas entradas $a, b$ equivalen a los siguientes polinomios:

a = $z^{232}+z^{230}+z^{229}+z^{228}+z^{227}+z^{226}+z^{224}+z^{223}+z^{222}+z^{221}+z^{219}+z^{218}+z^{215}+z^{212}+z^{208}+z^{206}+z^{205}+z^{203}+z^{199}+z^{198}+z^{197}+z^{195}+z^{194}+z^{193}+z^{192}+z^{190}+z^{187}+z^{185}+z^{184}+z^{183}+z^{181}+z^{180}+z^{178}+z^{177}+z^{175}+z^{170}+z^{165}+z^{163}+z^{161}+z^{157}+z^{156}+z^{155}+z^{153}+z^{152}+z^{151}+z^{149}+z^{143}+z^{141}+z^{139}+z^{136}+z^{135}+z^{133}+z^{131}+z^{126}+z^{125}+z^{124}+z^{123}+z^{121}+z^{119}+z^{118}+z^{117}+z^{115}+z^{113}+z^{112}+z^{111}+z^{108}+z^{106}+z^{104}+z^{103}+z^{98}+z^{97}+z^{96}+z^{92}+z^{88}+z^{87}+z^{82}+z^{81}+z^{80}+z^{79}+z^{77}+z^{74}+z^{73}+z^{70}+z^{69}+z^{68}+z^{67}+z^{66}+z^{65}+z^{60}+z^{57}+z^{54}+z^{51}+z^{49}+z^{46}+z^{45}+z^{42}+z^{41}+z^{38}+z^{35}+z^{33}+z^{28}+z^{27}+z^{26}+z^{25}+z^{24}+z^{23}+z^{21}+z^{19}+z^{17}+z^{16}+z^{15}+z^{10}+z^{5}+z^{4}$

b $= z^{229}+z^{227}+z^{225}+z^{224}+z^{222}+z^{221}+z^{220}+z^{218}+z^{217}+z^{216}+z^{213}+z^{212}+z^{211}+z^{210}+z^{209}+z^{208}+z^{207}+z^{204}+z^{202}+z^{200}+z^{198}+z^{197}+z^{196}+z^{191}+z^{190}+z^{189}+z^{188}+z^{187}+z^{186}+z^{185}+z^{182}+z^{181}+z^{180}+z^{179}+z^{178}+z^{176}+z^{175}+z^{174}+z^{173}+z^{172}+z^{171}+z^{169}+z^{164}+z^{163}+z^{160}+z^{154}+z^{153}+z^{152}+z^{151}+z^{150}+z^{149}+z^{147}+z^{145}+z^{143}+z^{139}+z^{138}+z^{136}+z^{134}+z^{130}+z^{126}+z^{124}+z^{123}+z^{118}+z^{117}+z^{116}+z^{114}+z^{112}+z^{110}+z^{109}+z^{107}+z^{104}+z^{99}+z^{98}+z^{97}+z^{92}+z^{90}+z^{87}+z^{86}+z^{84}+z^{83}+z^{81}+z^{79}+z^{76}+z^{74}+z^{71}+z^{69}+z^{67}+z^{61}+z^{57}+z^{54}+z^{52}+z^{49}+z^{47}+z^{46}+z^{42}+z^{41}+z^{39}+z^{37}+z^{36}+z^{35}+z^{34}+z^{31}+z^{30}+z^{28}+z^{27}+z^{24}+z^{22}+z^{21}+z^{20}+z^{18}+z^{16}+z^{14}+z^{13}+z^{11}+z^{9}+z^{8}+z^{7}+z^{4}+z^{3}+z^{2}$

Resultado:
c $= 10F6F77418842516ADD37998A64622792D3637FF8D06BC976A55FB0EF05_{16}$
$c$ es equivalente al siguiente polinomio:

c $= z^{232}+z^{227}+z^{226}+z^{225}+z^{224}+z^{222}+z^{221}+z^{219}+z^{218}+z^{217}+z^{216}+z^{214}+z^{213}+z^{212}+z^{210}+z^{209}+z^{208}+z^{206}+z^{200}+z^{199}+z^{195}+z^{190}+z^{185}+z^{182}+z^{180}+z^{176}+z^{174}+z^{173}+z^{171}+z^{169}+z^{167}+z^{166}+z^{164}+z^{163}+z^{162}+z^{160}+z^{157}+z^{156}+z^{154}+z^{153}+z^{152}+z^{151}+z^{148}+z^{147}+z^{144}+z^{143}+z^{139}+z^{137}+z^{134}+z^{133}+z^{130}+z^{126}+z^{125}+z^{121}+z^{117}+z^{114}+z^{113}+z^{112}+z^{111}+z^{108}+z^{105}+z^{103}+z^{102}+z^{100}+z^{97}+z^{96}+z^{94}+z^{93}+z^{89}+z^{88}+z^{86}+z^{85}+z^{84}+z^{83}+z^{82}+z^{81}+z^{80}+z^{79}+z^{78}+z^{77}+z^{76}+z^{75}+z^{71}+z^{70}+z^{68}+z^{62}+z^{61}+z^{59}+z^{57}+z^{56}+z^{55}+z^{54}+z^{51}+z^{48}+z^{46}+z^{45}+z^{44}+z^{42}+z^{41}+z^{39}+z^{37}+z^{34}+z^{32}+z^{30}+z^{28}+z^{27}+z^{26}+z^{25}+z^{24}+z^{23}+z^{21}+z^{20}+z^{15}+z^{14}+z^{13}+z^{11}+z^{10}+z^{9}+z^{8}+z^{2}+1$
Figura 6.13

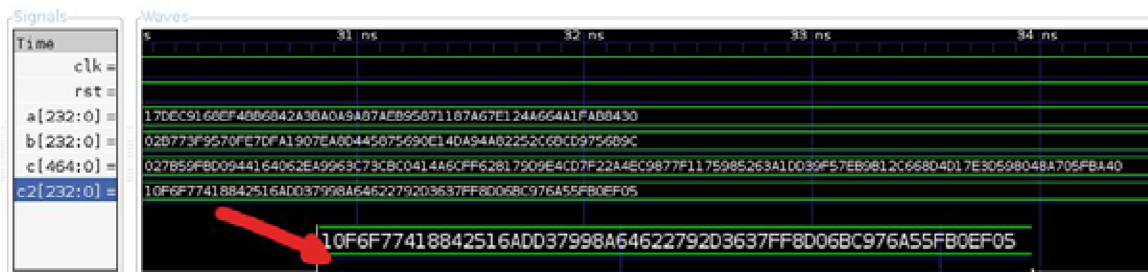


Figura 6.13: Multiplicador de 233 bits.

**Tercera prueba:**

Entradas:

a $= 03E13F982B4917F96CED2DC090806E4A12B515D402D19DB099DB79C0941_{16}$
b $= 125948599A6DA8BB286210FC79872C82FEB360CFA2D4B80D06F16EDD765_{16}$

Estas entradas $a, b$ equivalen a los siguientes polinomios:

a $= z^{229}+z^{228}+z^{227}+z^{226}+z^{225}+z^{220}+z^{217}+z^{216}+z^{215}+z^{214}+z^{213}+z^{212}+z^{211}+z^{208}+z^{207}+z^{201}+z^{199}+z^{197}+z^{196}+z^{194}+z^{191}+z^{188}+z^{184}+z^{182}+z^{181}+z^{180}+z^{179}+z^{178}+z^{177}+z^{176}+z^{175}+z^{172}+z^{170}+z^{169}+z^{167}+z^{166}+z^{163}+z^{162}+z^{161}+z^{159}+z^{158}+z^{156}+z^{153}+z^{151}+z^{150}+z^{148}+z^{147}+z^{146}+z^{139}+z^{136}+z^{131}+z^{122}+z^{121}+z^{119}+z^{118}+z^{117}+z^{114}+z^{111}+z^{109}+z^{104}+z^{101}+z^{99}+z^{97}+z^{96}+z^{94}+z^{92}+z^{88}+z^{86}+z^{84}+z^{83}+z^{82}+z^{80}+z^{78}+z^{69}+z^{67}+z^{66}+z^{64}+z^{60}+z^{59}+z^{56}+z^{55}+z^{54}+z^{52}+z^{51}+z^{49}+z^{48}+z^{43}+z^{40}+z^{39}+z^{36}+z^{35}+z^{34}+z^{32}+z^{31}+z^{29}+z^{28}+z^{26}+z^{25}+z^{24}+z^{23}+z^{20}+z^{19}+z^{18}+z^{11}+z^{8}+z^{6}+1$

b $= z^{232}+z^{229}+z^{226}+z^{224}+z^{223}+z^{220}+z^{218}+z^{215}+z^{210}+z^{208}+z^{207}+z^{204}+z^{203}+z^{200}+z^{199}+z^{197}+z^{194}+z^{193}+z^{191}+z^{190}+z^{188}+z^{187}+z^{185}+z^{183}+z^{179}+z^{177}+z^{176}+z^{175}+z^{173}+z^{172}+z^{169}+z^{167}+z^{162}+z^{161}+z^{157}+z^{152}+z^{147}+z^{146}+z^{145}+z^{144}+z^{143}+z^{142}+z^{138}+z^{137}+z^{136}+z^{135}+z^{132}+z^{131}+z^{126}+z^{125}+z^{124}+z^{121}+z^{119}+z^{118}+z^{115}+z^{109}+z^{107}+z^{106}+z^{105}+z^{104}+z^{103}+z^{102}+z^{101}+z^{99}+z^{97}+z^{96}+z^{93}+z^{92}+z^{90}+z^{89}+z^{83}+z^{82}+z^{79}+z^{78}+z^{77}+z^{76}+z^{75}+z^{73}+z^{69}+z^{67}+z^{66}+z^{64}+z^{62}+z^{59}+z^{57}+z^{56}+z^{55}+z^{47}+z^{46}+z^{44}+z^{38}+z^{37}+z^{35}+z^{34}+z^{33}+z^{32}+z^{28}+z^{26}+z^{25}+z^{23}+z^{22}+z^{21}+z^{19}+z^{18}+z^{16}+z^{15}+z^{14}+z^{12}+z^{10}+z^{9}+z^{8}+z^{6}+z^{5}+z^{2}+1$

Resultado:

c $= 09E235B1694EC600B0FD783C1861A70EB34ED0F00EBDF935927247ABD3B_{16}$
$c$ es equivalente al siguiente polinomio:

c $= z^{231}+z^{228}+z^{227}+z^{226}+z^{225}+z^{221}+z^{217}+z^{216}+z^{214}+z^{212}+z^{211}+z^{209}+z^{208}+z^{204}+z^{202}+z^{201}+z^{199}+z^{196}+z^{194}+z^{191}+z^{190}+z^{189}+z^{187}+z^{186}+z^{182}+z^{181}+z^{171}+z^{169}+z^{168}+z^{163}+z^{162}+z^{161}+z^{160}+z^{159}+z^{158}+z^{156}+z^{154}+z^{153}+z^{152}+z^{151}+z^{145}+z^{144}+z^{143}+z^{142}+z^{136}+z^{135}+z^{130}+z^{129}+z^{124}+z^{123}+z^{121}+z^{118}+z^{117}+z^{116}+z^{111}+z^{110}+z^{109}+z^{107}+z^{105}+z^{104}+z^{101}+z^{100}+z^{98}+z^{95}+z^{94}+z^{93}+z^{91}+z^{90}+z^{88}+z^{83}+z^{82}+z^{81}+z^{80}+z^{71}+z^{70}+z^{69}+z^{67}+z^{65}+z^{64}+z^{63}+z^{62}+z^{60}+z^{59}+z^{58}+z^{57}+z^{56}+z^{55}+z^{52}+z^{49}+z^{48}+z^{46}+z^{44}+z^{43}+z^{40}+z^{37}+z^{34}+z^{33}+z^{32}+z^{29}+z^{26}+z^{22}+z^{21}+z^{20}+z^{19}+z^{17}+z^{15}+z^{13}+z^{12}+z^{11}+z^{10}+z^{8}+z^{5}+z^{4}+z^{3}+z+1$
Figura 6.14

El costo total del multiplicador completo de 233 bits puede observarse en la tabla: 6.2.

| **Multiplicador No. Bits** | **No. Slices** | **No. LUTs de 4 entradas** | **Tiempo en ns** |
|---|---|---|---|
| 233 | 10003 | 17535 | 13.003 |

Tabla 6.2: Costo del multiplicador completo de 233 bits.

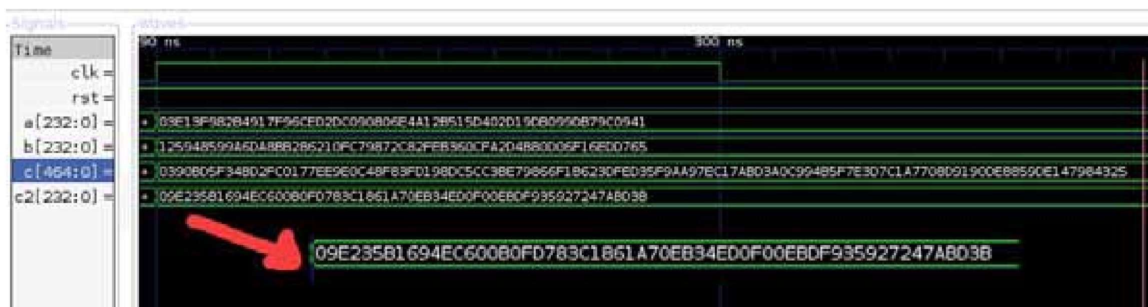


Figura 6.14: Multiplicador de 233 bits.

### 6.0.8. Implementación de cuadrados en campo finito $\mathbb{F}_{2^{233}}$

El código correspondiente puede consultarse en el Apéndice C.

#### Cuadrado de 233 bits

Para realizar esta operación se utilizó el algoritmo 9.

La función que se utiliza se definió como biblioteca en el archivo gf2n.vhdl donde se encuentra la siguiente función:

`function cuadrado (`$a$`: bit_vector(232 downto 0)) return bit_vector`
 $c = a^2 \in \mathbb{F}_2$ `de 233 bits`

Esta función recibe un elemento $a \in \mathbb{F}_2$ de 233 bits, y nos devuelve $c = a^2 \in \mathbb{F}_2$.

**Primera prueba:**

Entrada:

a $= 081B7D4ABE9DF920ECD0703E5A016FA57B0E94942211B4290F98FC64765_{16}$

Esta entrada $a$ equivale al siguiente polinomio:

a $= z^{231} + z^{224} + z^{223} + z^{221} + z^{220} + z^{218} + z^{217} + z^{216} + z^{215} + z^{214} + z^{212} + z^{210} + z^{207} + z^{205} + z^{203} + z^{201} + z^{200} + z^{199} + z^{198} + z^{197} + z^{195} + z^{192} + z^{191} + z^{190} + z^{188} + z^{187} + z^{186} + z^{185} + z^{184} + z^{183} + z^{180} + z^{177} + z^{171} + z^{170} + z^{169} + z^{167} + z^{166} + z^{163} + z^{162} + z^{160} + z^{154} + z^{153} + z^{152} + z^{145} + z^{144} + z^{143} + z^{142} + z^{141} + z^{138} + z^{136} + z^{135} + z^{133} + z^{124} + z^{122} + z^{121} + z^{119} + z^{118} + z^{117} + z^{116} + z^{115} + z^{113} + z^{110} + z^{108} + z^{106} + z^{105} + z^{104} + z^{103} + z^{101} + z^{100} + z^{95} + z^{94} + z^{93} + z^{91} + z^{88} + z^{86} + z^{83} + z^{80} + z^{78} + z^{73} + z^{69} + z^{64} + z^{60} + z^{59} + z^{57} + z^{56} + z^{54} + z^{49} + z^{47} + z^{44} + z^{39} + z^{38} + z^{37} + z^{36} + z^{35} + z^{32} + z^{31} + z^{27} + z^{26} + z^{25} + z^{24} + z^{23} + z^{22} + z^{18} + z^{17} + z^{14} + z^{10} + z^{9} + z^{8} + z^{6} + z^{5} + z^{2} + 1$

Resultado:

c = $1E6B31D4DA0DCCB92493087AEAE3F32F66D7ABDA78143BA4DE70051DB6F_{16}$
$c$ es equivalente al siguiente polinomio:

c = $z^{232}+z^{231}+z^{230}+z^{229}+z^{226}+z^{225}+z^{223}+z^{221}+z^{220}+z^{217}+z^{216}+z^{212}+z^{211}+z^{210}+z^{208}+z^{206}+z^{203}+z^{202}+z^{200}+z^{199}+z^{197}+z^{191}+z^{190}+z^{188}+z^{187}+z^{186}+z^{183}+z^{182}+z^{179}+z^{177}+z^{176}+z^{175}+z^{172}+z^{169}+z^{166}+z^{163}+z^{160}+z^{157}+z^{156}+z^{151}+z^{146}+z^{145}+z^{144}+z^{143}+z^{141}+z^{139}+z^{138}+z^{137}+z^{135}+z^{133}+z^{131}+z^{130}+z^{129}+z^{125}+z^{124}+z^{123}+z^{122}+z^{121}+z^{120}+z^{117}+z^{116}+z^{113}+z^{111}+z^{110}+z^{109}+z^{108}+z^{106}+z^{105}+z^{102}+z^{101}+z^{99}+z^{98}+z^{96}+z^{94}+z^{93}+z^{92}+z^{91}+z^{89}+z^{87}+z^{85}+z^{84}+z^{83}+z^{82}+z^{80}+z^{79}+z^{77}+z^{74}+z^{73}+z^{72}+z^{71}+z^{64}+z^{62}+z^{57}+z^{56}+z^{55}+z^{53}+z^{52}+z^{51}+z^{49}+z^{46}+z^{43}+z^{42}+z^{40}+z^{39}+z^{38}+z^{37}+z^{34}+z^{33}+z^{32}+z^{22}+z^{20}+z^{16}+z^{15}+z^{14}+z^{12}+z^{11}+z^{9}+z^{8}+z^{6}+z^{5}+z^{3}+z^{2}+z+1$
Figura 6.15

Figura 6.15: Cuadrado de 233 bits.

**Segunda prueba:**

Entrada:

a = $17DEC9168EF4BB6842A3BA0A9A87AEB95871187A67E124A664A1FAB8430_{16}$
Esta entrada $a$ equivale al siguiente polinomio:

a = $z^{232}+z^{230}+z^{229}+z^{228}+z^{227}+z^{226}+z^{224}+z^{223}+z^{222}+z^{221}+z^{219}+z^{218}+z^{215}+z^{212}+z^{208}+z^{206}+z^{205}+z^{203}+z^{199}+z^{198}+z^{197}+z^{195}+z^{194}+z^{193}+z^{192}+z^{190}+z^{187}+z^{185}+z^{184}+z^{183}+z^{181}+z^{180}+z^{178}+z^{177}+z^{175}+z^{170}+z^{165}+z^{163}+z^{161}+z^{157}+z^{156}+z^{155}+z^{153}+z^{152}+z^{151}+z^{149}+z^{143}+z^{141}+z^{139}+z^{136}+z^{135}+z^{133}+z^{131}+z^{126}+z^{125}+z^{124}+z^{123}+z^{121}+z^{119}+z^{118}+z^{117}+z^{115}+z^{113}+z^{112}+z^{111}+z^{108}+z^{106}+z^{104}+z^{103}+z^{98}+z^{97}+z^{96}+z^{92}+z^{88}+z^{87}+z^{82}+z^{81}+z^{80}+z^{79}+z^{77}+z^{74}+z^{73}+$

$z^{70}+z^{69}+z^{68}+z^{67}+z^{66}+z^{65}+z^{60}+z^{57}+z^{54}+z^{51}+z^{49}+z^{46}+z^{45}+z^{42}+z^{41}+z^{38}+z^{35}+z^{33}+z^{28}+z^{27}+z^{26}+z^{25}+z^{24}+z^{23}+z^{21}+z^{19}+z^{17}+z^{16}+z^{15}+z^{10}+z^{5}+z^{4}$

Resultado:

c $= 065C931E21D01AB68B76412EAB352E794C24D9F4BF3502334A6610EE77E_{16}$
$c$ es equivalente al siguiente polinomio:

c $= z^{230}+z^{229}+z^{226}+z^{224}+z^{223}+z^{222}+z^{219}+z^{216}+z^{213}+z^{212}+z^{208}+z^{207}+z^{206}+z^{205}+z^{201}+z^{196}+z^{195}+z^{194}+z^{192}+z^{184}+z^{183}+z^{181}+z^{179}+z^{177}+z^{176}+z^{174}+z^{173}+z^{171}+z^{167}+z^{165}+z^{164}+z^{162}+z^{161}+z^{160}+z^{158}+z^{157}+z^{154}+z^{148}+z^{145}+z^{143}+z^{142}+z^{141}+z^{139}+z^{137}+z^{135}+z^{133}+z^{132}+z^{129}+z^{128}+z^{126}+z^{124}+z^{121}+z^{119}+z^{118}+z^{117}+z^{114}+z^{113}+z^{112}+z^{111}+z^{108}+z^{106}+z^{103}+z^{102}+z^{97}+z^{94}+z^{91}+z^{90}+z^{88}+z^{87}+z^{84}+z^{83}+z^{82}+z^{81}+z^{80}+z^{78}+z^{75}+z^{73}+z^{72}+z^{71}+z^{70}+z^{69}+z^{68}+z^{65}+z^{64}+z^{62}+z^{60}+z^{53}+z^{49}+z^{48}+z^{45}+z^{44}+z^{42}+z^{39}+z^{37}+z^{34}+z^{33}+z^{30}+z^{29}+z^{24}+z^{19}+z^{18}+z^{17}+z^{15}+z^{14}+z^{13}+z^{10}+z^{9}+z^{8}+z^{6}+z^{5}+z^{4}+z^{3}+z^{2}+z$

Figura 6.16

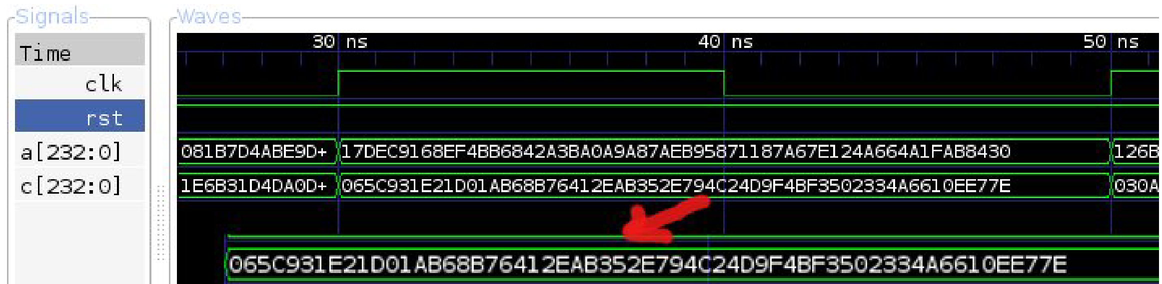


Figura 6.16: Cuadrado de 233 bits.

**Tercera prueba:**

Entrada:

a $= 126BC400F9DCAD6FEBDA08512342DA42CAD9EE9E76A13012BECB61D841A_{16}$

Esta entrada $a$ equivale al siguiente polinomio:

a $= z^{232}+z^{229}+z^{226}+z^{225}+z^{223}+z^{221}+z^{220}+z^{219}+z^{218}+z^{214}+z^{203}+z^{202}+z^{201}+z^{200}+z^{199}+z^{196}+z^{195}+z^{194}+z^{192}+z^{191}+z^{190}+z^{187}+z^{185}+z^{183}+z^{182}+z^{180}+z^{178}+z^{177}+z^{175}+z^{174}+z^{173}+z^{172}+z^{171}+z^{170}+z^{169}+z^{167}+z^{165}+z^{164}+z^{163}+z^{162}+z^{160}+z^{159}+z^{157}+z^{151}+z^{146}+z^{144}+z^{140}+z^{137}+z^{133}+z^{132}+z^{130}+z^{125}+z^{123}+z^{122}+z^{120}+z^{119}+z^{117}+z^{114}+z^{109}+z^{107}+z^{106}+z^{103}+z^{101}+z^{99}+z^{98}+z^{96}+z^{95}+z^{92}+z^{91}+z^{90}+z^{89}+z^{87}+z^{86}+z^{85}+z^{83}+z^{80}+z^{79}+z^{78}+z^{77}+z^{74}+z^{73}+$

$z^{72}+z^{70}+z^{69}+z^{67}+z^{65}+z^{60}+z^{57}+z^{56}+z^{48}+z^{45}+z^{43}+z^{41}+z^{40}+z^{39}+z^{38}+z^{37}+z^{35}+z^{34}+z^{31}+z^{29}+z^{28}+z^{26}+z^{25}+z^{20}+z^{19}+z^{18}+z^{16}+z^{15}+z^{10}+z^{4}+z^{3}+z$

Resultado:

c = $030AEFA4E71EBFCF44374B9A61F9C67E003A510D97459D1D343C8127CA7_{16}$
$c$ es equivalente al siguiente polinomio:

c $=z^{229}+z^{228}+z^{223}+z^{221}+z^{219}+z^{218}+z^{217}+z^{215}+z^{214}+z^{213}+z^{212}+z^{211}+z^{209}+z^{206}+z^{203}+z^{202}+z^{201}+z^{198}+z^{197}+z^{196}+z^{192}+z^{191}+z^{190}+z^{189}+z^{187}+z^{185}+z^{184}+z^{183}+z^{182}+z^{181}+z^{180}+z^{179}+z^{178}+z^{175}+z^{174}+z^{173}+z^{172}+z^{170}+z^{166}+z^{161}+z^{160}+z^{158}+z^{157}+z^{156}+z^{154}+z^{151}+z^{149}+z^{148}+z^{147}+z^{144}+z^{143}+z^{141}+z^{138}+z^{137}+z^{132}+z^{131}+z^{130}+z^{129}+z^{128}+z^{127}+z^{124}+z^{123}+z^{122}+z^{118}+z^{117}+z^{114}+z^{113}+z^{112}+z^{111}+z^{110}+z^{109}+z^{97}+z^{96}+z^{95}+z^{93}+z^{90}+z^{88}+z^{84}+z^{79}+z^{78}+z^{76}+z^{75}+z^{72}+z^{70}+z^{69}+z^{68}+z^{66}+z^{62}+z^{60}+z^{59}+z^{56}+z^{55}+z^{54}+z^{52}+z^{48}+z^{47}+z^{46}+z^{44}+z^{41}+z^{40}+z^{38}+z^{33}+z^{32}+z^{31}+z^{30}+z^{27}+z^{20}+z^{17}+z^{14}+z^{13}+z^{12}+z^{11}+z^{10}+z^{7}+z^{5}+z^{2}+z+1$

Figura 6.17

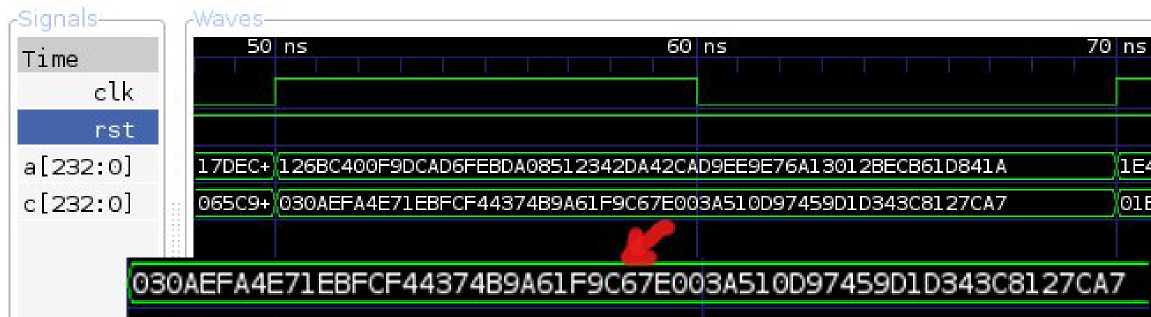


Figura 6.17: Cuadrado de 233 bits.

El costo total de la operación elevar al cuadrado de 233 bits puede observarse en la tabla: 6.3.

| **Cuadrado No. Bits** | **No. Slices** | **No. LUTs de 4 entradas** | **Tiempo en ns** |
|---|---|---|---|
| 233 | 88 | 153 | 2.098 |

Tabla 6.3: Costo de la operación cuadrado de 233 bits.

# Capítulo 7

# Conclusiones

Las curvas elípticas son una buena opción para la criptografía seleccionando los parámetros correctos ya que hasta ahora han demostrado su eficiencia en tiempo de cálculos, espacio reducido en Hardware para dispositivos pequeños y con recursos limitados. Las curvas elípticas comparado con otros sistemas de cifrado como el RSA que utiliza 1024 bits para garantizar su seguridad, muestran el mismo nivel de resguardo utilizando tan solo 160 bits.

A continuación se muestran comparaciones de las operaciones desarrolladas en el Proyecto actual en cuanto a los tiempos y los recursos en Hardware:

### Comparación del multiplicador sobre campos finitos $\mathbb{F}_{2^m}$

| Referencia | Plataforma | Multiplicador | No. Slices | No. LUTs | Tiempo en ns |
|---|---|---|---|---|---|
| Proyecto actual | XC2VP30 | $\mathbb{F}_{2^{233}}$ | 10003 | 17535 | 13.003 |
| [24] | XC2V4000 | $\mathbb{F}_{2^{233}}$ | 9588 | 16674 | >14.497 |
| [25] | XCV3200E | $\mathbb{F}_{2^{163}}$ | 6730 | - | 21.623 |

Tabla 7.1: Comparación de multiplicadores de n bits

Como se puede observar en la tabla 7.1 en el presente proyecto se utiliza un mayor número de slices en el multiplicador sobre campos finitos $\mathbb{F}_{2^{233}}$ esto es por la implementación del multiplicador de Karatsuba-Ofman de 4 bits y comparado con [24], está ocupa un 4 % menos en cuanto a slices, pero la diferencia entre tiempo podemos decir que es mucho mas eficiente el presente proyecto ya que está realiza la operación aproximadamente un 10 % mas rápido esto es lógico ya que la operación ocupa más recursos en Hardware. Queda a discusión sobre cual multiplicador es mejor ya que dependiendo cual sea el objetivo que se esté buscando así sea el tiempo o espacio requerido para calcular dicha operación.

### Comparación del cuadrado sobre campos finitos $\mathbb{F}_{2^m}$

| Referencia | Plataforma | Cuadrado | No. Slices | No. LUTs | Tiempo en ns |
|---|---|---|---|---|---|
| Proyecto actual | XC2VP30 | $\mathbb{F}_{2^{233}}$ | 88 | 153 | 2.098 |
| [24] | XC2V4000 | $\mathbb{F}_{2^{233}}$ | 88 | 153 | - |
| [25] | XCV3200E | $\mathbb{F}_{2^{163}}$ | 95 | - | 41.765 |

Tabla 7.2: Comparación del cuadrado de n bits

La tabla 7.2 muestra el mismo recurso en Hardware ya que se utiliza el mismo algoritmo entre el Proyecto actual y [24] para desarrollar la operación de elevar al cuadrado.

### 7.0.9. Trabajos a futuro

El objetivo general que se persigue en el proyecto es generar un par de llaves: llave pública y llave privada a nivel de hardware para obtener un mejor rendimiento en el calculo empleando la aritmética de curvas elípticas para lo cual se necesita desarrollar las operaciones sobre campos finitos $\mathbb{F}_{2^m}$ como lo son la multiplicación, la adición, el cuadrado, el inverso, la solución de la ecuación cuadrática de Weierstrass. Donde las primeras tres operaciones fueron desarrolladas en el proyecto (La adición sobre $\mathbb{F}_{2^m}$ se desarrolla mediante la operaciones lógicas XOR).

Para el desarrollo del inverso implica calcular varias multiplicaciones con lo que resulta una operación sumamente costosa hablando en términos computacionales aplicando el algoritmo extendido de Euclides.

Las *coordenadas proyectivas* permiten realizar la adición de puntos a través de la representación de cada punto como una clase de relación de equivalencia. La representación tradicional de coordenadas se le conoce como *coordenadas afines*, cada punto afín puede ser relacionado uno a uno donde la idea fundamental es dividir el espacio $K^3$ en clases definidas por equivalencias. La suma y doblado son redefinidos sobre la nueva representación, ahora estas operaciones son realizadas sin necesidad de realizar inversiones de campo, dichas inversiones solo aparecen al convertir una representación proyectiva en un afín [14].

La función traza (trace function) se define como:
Sea $c = \sum_{i=0}^{m-1} c_i z^i \in \mathbb{F}_{2^m}$, con $c_i \in 0,\ 1$

$$Tr(c) = Tr(\sum_{i=0}^{m-1} c_i z^i) = \sum_{i=0}^{m-1} c_i Tr(z^i)$$

La cual tiene las siguientes propiedades: Sea $c \in \mathbb{F}_{2^m}$

1. $Tr(c) = Tr(c^2) = Tr(c)^2$, y $Tr(c) \in 0,\ 1$.
2. $Tr(c + d) = Tr(c) + Tr(d)$, es lineal.
3. Sea $u \in \mathbb{F}_{2^m}$, entonces $Tr(u) = Tr(a)$, donde $a$ es el coeficiente de la ecuación $y^2 + xy = x^3 + ax^2 + b$.

Para resolver la ecuación cuadrática reducida de Weierstrass $E : y^2 = x^3 + ax + b$ conociendo los valores de $x$, $a$ y $b$, se puede aplicar la definición de la media traza (half-trace)[14]: Sea $m$ un entero par, la función media traza se define como:

$$H(c) = \sum_{i=0}^{\frac{(m-1)}{2}} c^{2^{2i}}$$

Está tiene las siguientes propiedades: Sea $c,\ d \in \mathbb{F}_{2^m}$

1. $H(c+d) = H(c) + H(d)$, para todo $c,\ d\ \in \mathbb{F}_{2^m}$
2. $H(c) = H(c^2) + c + Tr(c)$, para todo $c \in \mathbb{F}_{2^m}$.
3. $H(c)$ es una solución a la ecuación cuadrática $x^2 + x = c + Tr(c)$.

Una vez desarrollando las operaciones anteriores sobre campos finitos $\mathbb{F}_{2^m}$ podemos dar paso a la implementación de las operaciones aritméticas sobre curvas elípticas en campos finitos $\mathbb{F}_{2^m}$ como lo son la suma, doblado de puntos y finalizando con la multiplicación escalar generando así el par de llaves para uso criptográfico.

# Apéndice A

# Códigos fuentes

---

Archivo: `galois_f2n.cpp` (*Métodos necesarios para definir el campo* $\mathbb{F}_{2^m}$)

```
1 # ###########################################################################
2 #void galois_f2n::c_Galois_F2n(int &caracteristica, lidia_size_t &m){
3 #        gf2n_init(m);
4 #        galois_field campo(caracteristica, m);
5 #        galois_f2n field;
6 #        field.polinomio_Irreducible(campo);
7 #        field.combinaciones_Galois_F2n(campo);
8 #        return;
9 #}
10 #
11 #void galois_f2n::polinomio_Irreducible(galois_field &campo){
12 #        cout<<"\n***El campo finito binario esta formado por el polinomio irreducible: "<<endl;
13 #        cout<<"\t\t\t\t";
14 #        campo.irred_polynomial().pretty_print();
15 #        cout<<endl;
16 #        return;
17 #}
18 #
19 #void galois_f2n::combinaciones_Galois_F2n(galois_field & campo){
20 #        cout <<"\n***Numero de combinaciones posibles sobre el campo: "<<campo.number_of_elements()\
21          <<endl;
22 #        return;
23 #}
24 # ###########################################################################
```

---

Archivo: `AritmeticaCE_GF2n.cpp` (*Aritmética sobre el campo finito* $\mathbb{F}_{2^m}$)

```
 1# ##########################################################################################
 2#void  AritmeticaCE_GF2n::leerPunto(gf2n &s){
 3#        cin>>s;
 4#        return;
 5#}
 6#
 7#int  AritmeticaCE_GF2n::LambdaAdicion(gf2n &S,  Punto &P,  Punto &Q){
 8#        gf2n Ry;
 9#        add(S, P.x, Q.x);
10#        if(S.is_zero()){
11#                return(0);
12#        }
13#        add(Ry, P.y, Q.y);
14#        invert(S, S);
15#        multiply(S, S, Ry);
16#        return(1);
17#}
18#
19#void  AritmeticaCE_GF2n::AdicionCE_GF2n(Punto &R,  Punto &P,  Punto &Q,  gf2n &S,  gf2n &a){
20#        if(Q.x.is_zero()  ||  P.x.is_zero()){
21#               R.x.assign(P.x);
22#               R.y.assign(P.y);
23#               return;
24#        }
25#        gf2n aux,  s2;
26#
27#        add(aux,Q.x,a);
28#        add(aux,aux,P.x);
29#        add(aux,aux,S);
30#        square(s2,S);
31#        add(aux,aux,s2);
32#        R.x = aux;
33#
34#        add(s2,aux,P.y);
35#        add(aux,P.x,aux);
36#        multiply(aux,S,aux);
37#        add(aux,aux,s2);
38#        R.y = aux;
39#        return;
40#}
41#
42#int  AritmeticaCE_GF2n::LambdaDoblado(gf2n &S,  Punto &P){
43#        if(P.x.is_zero()){
44#                return(0);
45#        }
46#        else{
47#                invert(S, P.x);
48#                multiply(S, S, P.y);
49#                add(S, S, P.x);
50#                return(1);
51#        }
52#}
53#
54#void  AritmeticaCE_GF2n::DobladoCE_GF2n(Punto &R,  Punto &P,  gf2n &S,  gf2n &a){
55#        gf2n aux,  s2;
56#
57#        square(s2, S);
58#        add(aux,S,a);
59#        add(aux,s2,aux);
```

```
60#        R.x = aux;
61#
62#        multiply(s2,S,aux);
63#        add(s2,s2,aux);
64#        square(aux,P.x);
65#        add(aux,aux,s2);
66#        R.y = aux;
67#        return;
68#}
69#
70#void AritmeticaCE_GF2n::GraficarPuntos(lidia_size_t &m, gf2n &a, gf2n &b){
71#        gf2n x, y;
72#        gf2n aux1, aux2, aux3;
73#        bigint cont, rango;
74#        fstream ptr("puntosDec", ios::out);
75#
76#        power(rango,2, m);
77#
78#        for(cont = 0;cont<rango;cont++){
79#                add(x,cont,0);
80#                square(aux1, x);
81#                multiply(aux1,aux1,a);
82#                add(aux1,aux1,b);
83#                power(aux2,x,3);
84#                add(aux2,aux1,aux2);
85#
86#                solve_quadratic(y,x,aux2);
87#
88#                square(aux1,y);
89#                multiply(aux3,x,y);
90#                add(aux1,aux1,aux3);
91#                add(aux1,aux1,aux2);
92#
93#                if(aux1.is_zero()){
94#                        ptr<<x<<endl<<y<<endl;
95#                        if(!x.is_zero()){
96#                                add(y,x,y);
97#                                ptr<<x<<endl<<y<<endl;
98#                        }
99#                }
100#        }
101#        ptr.close();
102#
103#        char cadena[300];
104#        int cuenta = 1;
105#        fstream ptr2("puntosDec", ios::in);
106#        fstream ptr3("Puntos", ios::out);
107#        ptr2>>cadena;
108#        while(!ptr2.eof()){
109#                for(int i=4; i<strlen(cadena); i++){
110#                        ptr3<<cadena[i];
111#                }
112#
113#                if(cuenta%2 == 1){
114#                        ptr3<<"\t";
115#                }
116#                else{
117#                        ptr3<<endl;
118#                }
119#                cuenta++;
120#                ptr2>>cadena;
121#        }
122#        ptr2.close();
123#        remove("puntosDec");
124#        ptr3.close();
125#}
```

```
126#
127#int  AritmeticaCE_GF2n::NegativoPuntoCE_GF2n(Punto &P, Punto &R){
128#        if(P.x.is_zero()){
129#                return(0);
130#        }
131#        else{
132#                R.x = P.x;
133#                add(R.y,P.x,P.y);
134#                return(1);
135#        }
136#}
137#
138#int  AritmeticaCE_GF2n::PuntosAleatoriosCE_GF2n(lidia_size_t &m, gf2n &a, gf2n &b, Punto &puntoG){
139#        puntoG.x = NULL;
140#        puntoG.y = NULL;
141#        Punto R;
142#        R.x = NULL;
143#        R.y = NULL;
144#        gf2n aux1, aux2, aux3;
145#        int x;
146#
147#        R.x = randomize(a,m);
148#        square(aux1, R.x);
149#        multiply(aux1,aux1,a);
150#        add(aux1,aux1,b);
151#        power(aux2,R.x,3);
152#        add(aux2,aux1,aux2);
153#
154#        solve_quadratic(R.y,R.x,aux2);
155#
156#        square(aux1,R.y);
157#        multiply(aux3,R.x,R.y);
158#        add(aux1,aux1,aux3);
159#        add(aux1,aux1,aux2);
160#
161#        if(aux1.is_zero()){
162#                x = rand()%2;
163#                if(x==0){
164#                        add(R.y,R.x,R.y);
165#                }
166#                puntoG.x = R.x;
167#                puntoG.y = R.y;
168#                return(0);
169#        }
170#        else{
171#                return(1);
172#        }
173#}
174#
175#int  AritmeticaCE_GF2n::ComprobacionEcuacion(Punto &P, gf2n &a, gf2n &b){
176#        gf2n aux1, aux2,aux3;
177#
178#        square(aux1, P.x);
179#        multiply(aux1,aux1,a);
180#        add(aux1,aux1,b);
181#        power(aux2,P.x,3);
182#        add(aux2,aux1,aux2);
183#
184#        square(aux1,P.y);
185#        multiply(aux3,P.x,P.y);
186#        add(aux1,aux1,aux3);
187#        add(aux1,aux1,aux2);
188#        if(aux1.is_zero()){
189#                return(1);
190#        }
191#        return(0);
```

```
192#}
193#
194#void  AritmeticaCE_GF2n::CadenaBinaria(bigint &k, char binario[235], int &lim){
195#        bigint r;
196#        char aux[235];
197#        int i=0, j;
198#
199#        while(k!=0){
200#                r = k%2;
201#                if(r==1){
202#                        aux[i]='1';
203#                }
204#                else{
205#                        aux[i]='0';
206#                }
207#                i++;
208#                k /=2;
209#        }
210#        lim = i;
211#        for(j=0; j<lim; j++){
212#                binario[j] = aux[i-1];
213#                i--;
214#        }
215#        binario[j] = '\0';
216#        ~r;
217#}
218#
219#void  AritmeticaCE_GF2n::MultiplicacionEscalarCE_GF2n(Punto &Q, Punto &P, char binario[235],
220#      int &lim, gf2n &a){
221#        int i=0, j;
222#        gf2n S;
223#        Punto R, R2;
224#        bigint cont, T;
225#        cont = 1;
226#        T = 1;
227#
228#        AritmeticaCE_GF2n w;
229#        Q.x.assign_zero();
230#        Q.y.assign_zero();
231#
232#        while(binario[i]!='1'){
233#                i++;
234#        }
235#        if(lim==1){
236#                Q.x.assign(P.x);
237#                Q.y.assign(P.y);
238#                return;
239#        }
240#        Q.x.assign(P.x);
241#        Q.y.assign(P.y);
242#
243#        w.LambdaDoblado(S, Q);
244#        w.DobladoCE_GF2n(R2, Q, S, a);
245#
246#        for(j=i+1; j<lim; j++){
247#                T = 2*T;
248#                if(w.LambdaDoblado(S, Q)==0){
249#                                //cout<<"Orden Punto: "<<T<<endl;
250#                                R.x.assign_zero();
251#                                R.y.assign_zero();
252#                                cout<<T<<"P = ("<<R.x<<", "<<R.y<<"); \n";
253#                }
254#                else{
255#                        w.DobladoCE_GF2n(R, Q, S, a);
256#                                cout<<T<<"P = ("<<R.x<<", "<<R.y<<"); \n";
257#                }
```

```
258#
259#                  if(binario[j]=='1'){
260#                          T = T + cont;
261#                          if(R.x == P.x && R.y == P.y){
262#                                  R.x.assign(R2.x);
263#                                  R.y.assign(R2.y);
264#                                  cout<<T<<"P = ("<<R.x<<", "<<R.y<<"); \n";
265#                          }
266#                          else if(R.x == P.x && R.x != R.y){
267#                                          //cout<<"\t--->Orden Punto: "<<T<<endl;
268#                                          R.x.assign_zero();
269#                                          R.y.assign_zero();
270#                                          cout<<T<<"P = ("<<R.x<<", "<<R.y<<"); \n";
271#                          }
272#                          else{
273#                                  w.LambdaAdicion(S, P, R);
274#                                  w.AdicionCE_GF2n(R, P ,R, S, a);
275#                                  cout<<T<<"P = ("<<R.x<<", "<<R.y<<"); \n";
276#                          }
277#                  }
278#                  Q.x.assign(R.x);
279#                  Q.y.assign(R.y);
280#          }
281#}
282#
283#void AritmeticaCE_GF2n::LlavePrivadaCE_GF2n(lidia_size_t &m, gf2n &a, bigint &k){
284#          char cadena[300];
285#          gf2n llavek;
286#          do{
287#                  llavek = randomize(a,m);
288#          }while(llavek.is_zero());
289#
290#          fstream ptr("llave", ios::out);
291#          ptr<<llavek;
292#          ptr.close();
293#
294#          fstream ptr2("llave", ios::in);
295#          ptr2>>cadena;
296#          ptr2.close();
297#
298#          fstream ptr3("llave", ios::out);
299#          for(int i=4; i<strlen(cadena); i++){
300#                  ptr3<<cadena[i];
301#          }
302#          ptr3.close();
303#
304#          fstream ptr4("llave", ios::in);
305#          ptr4>>k;
306#          ptr4.close();
307#
308#          remove("llave");
309#          return;
310#}
311#
312# ##################################################################################
```

Archivo: `Graficar_GF2n.cpp` (*Graficar puntos de una curva elíptica*)

```
1# ##############################################################################
2#void Graficar_GF2n::GraficaPuntosOctave(){
3#        fstream file("GraficaGF2n.m",ios::out);
4#        file<<"d = load('Puntos');"<<endl;
5#        file<<"x = d(:,1);"<<endl;
6#        file<<"y = d(:,2);"<<endl;
7#        file<<"plot(x,y,'*');"<<endl;
8#        file<<"grid(\"minor\");"<<endl;
9#        file.close();
10#       return;
11#}
12#
13#void Graficar_GF2n::GraficaSuma(struct Punto &r,struct Punto &p, struct Punto &q){
14#       GraficaPuntosOctave();
15#
16#       fstream ptr("puntosDec2", ios::out);
17#       ptr<<r.x<<endl<<p.x<<endl<<q.x<<endl<<r.y<<endl<<p.y<<endl<<q.y;
18#       ptr.close();
19#
20#       char cadena[100], aux[100];
21#       string h[6];
22#       int cuenta = 1, y = 0 , l, i;
23#       fstream ptr2("puntosDec2", ios::in);
24#       ptr2>>cadena;
25#       while(ptr2){
26#               for(i=4, l=0; i<strlen(cadena); i++, l++){
27#                       aux[l]=cadena[i];
28#               }
29#               aux[l] = '\0';
30#               h[y] = aux;
31#               ptr2>>cadena;
32#               y++;
33#       }
34#       ptr2.close();
35#       remove("puntosDec2");
36#
37#       fstream file("GraficaGF2n.m",ios::app | ios::out);
38#
39#               file<<"hold on"<<endl;
40#
41#               file<<"p = (["<<h[0]<<","<<h[1]<<","<<h[2]<<"]);"<<endl;
42#               file<<"q = (["<<h[3]<<","<<h[4]<<","<<h[5]<<"]);"<<endl;
43#
44#               file<<"plot(p,q,'r');"<<endl;
45#               file<<"hold off;"<<endl;
46#
47#       file.close();
48#       return;
49#}
50#
51#void Graficar_GF2n::GraficaDoblado_Negativo(struct Punto &r,struct Punto &p){
52#       GraficaPuntosOctave();
53#
54#       fstream ptr("puntosDec2", ios::out);
55#       ptr<<r.x<<endl<<r.x<<endl<<p.x<<endl;
56#       ptr<<r.y<<endl<<r.y<<endl<<p.y<<endl;
57#       ptr.close();
58#
59#       char cadena[100], aux[100];
60#       string h[6];
61#       int cuenta = 1, y = 0 , l, i;
62#       fstream ptr2("puntosDec2", ios::in);
63#       ptr2>>cadena;
64#       while(ptr2){
65#               for(i=4, l=0; i<strlen(cadena); i++, l++){
66#                       aux[l]=cadena[i];
```

```
67#                 }
68#                 aux[l] = '\0';
69#                 h[y] = aux;
70#                 ptr2>>cadena;
71#                 y++;
72#         }
73#         ptr2.close();
74#         remove("puntosDec2");
75#
76#         fstream file("GraficaGF2n.m",ios::app | ios::out);
77#                 file<<"hold on"<<endl;
78#                 file<<"p = (["<<h[0]<<","<<h[1]<<","<<h[2]<<"]);"<<endl;
79#                 file<<"q = (["<<h[3]<<","<<h[4]<<","<<h[5]<<"]);"<<endl;
80#                 file<<"plot(p,q,'r');"<<endl;
81#                 file<<"hold off;"<<endl;
82#         file.close();
83#         return;
84#}
85#
86# ##########################################################################################
```

Archivo: `menu.cpp` (*Menu principal para operaciones aritméticas en* $E(\mathbb{F}_{2^m})$)

```
 1# ##########################################################################################
 2#void menu::muestra_Menu(int op){
 3#         menu M;
 4#
 5#         AritmeticaCE_GF2n f;
 6#         Punto p, q, r, puntoG;
 7#
 8#         bigint iter, k;
 9#
10#         Graficar_GF2n Grafica;
11#
12#         galois_f2n g;
13#         int caracteristica=2, lim;
14#         lidia_size_t m;
15#
16#         char cadBin[235], u;
17#
18#         gf2n a, b, S;
19#
20#         g.m_Galois_F2n(m);
21#         g.c_Galois_F2n(caracteristica, m);
22#         M.iniciarValores(a, b);
23#
24#         M.mostrar_Menu();
25#         do{
26#                 cout<<"\t\tSelecciona una opcion:\n >>";
27#                 cin>>op;
28#                 switch (op){
29#                         case 0:
30#                                 cout<<"Menu cero"<<endl;
31#                                 M.mostrar_Menu();
32#                         break;
33#                         case 1:
34#                                 g.leer_m_GF2n(m);
```

```
35#                     g.c_Galois_F2n(caracteristica, m);
36#                     g.rango_GF2n(m);
37#                     cout<<"valor de";
38#                     g.leer_x_GF2n(a, 0, m);
39#                     cout<<"valor de";
40#                     g.leer_x_GF2n(b, 1, m);
41#                break;
42#                case 2:
43#                     if(m<=10){
44#                          cout<<"\n***Estos son los puntos que existen
45#                               en la curva:***\n";
46#                          f.CalculaPuntos(m,a,b);
47#
48#                          cout<<"\n***¿Deseas graficar los puntos
49#                               sobre la curva S/N?: >"; cin>>u;
50#                          if(u == 'S' || u == 's'){
51#                               f.GraficarPuntos(m,a,b);
52#                               Grafica.GraficaPuntosOctave();
53#                               cout<<"\nAhora abra octave y
54#                                    ejecuta:\t"<<"octave:1>GraficaGF2n\n";
55#                          }
56#                     }
57#                break;
58#                case 3:
59#                     cout<<"*** GF(2^"<<m<<")"<<endl;
60#                     cout<<"\n***E: y^2 + x*y - [x^3 + "<<a<<"*x^2 + "<<b<<"]"<<endl;
61#                     g.c_Galois_F2n(caracteristica, m);
62#
63#                break;
64#                case 4:
65#                     cout<<"\n*****Suma de Puntos Sobre La Curva Eliptica****"<<endl;
66#                     cout<<"\tR(x,y) = P(x,y) + Q(x,y)"<<endl;
67#
68#                     do{
69#                          cout<<"\nDame el valor de Px: ";
70#                          f.leerPunto(p.x);
71#                          cout<<"Dame el valor de Py: ";
72#                          f.leerPunto(p.y);
73#                          if(f.ComprobacionEcuacion(p, a, b)==0){
74#                               cout<<"\n*El punto P no vive sobre la curva"<<endl;
75#                          }
76#                     }while(f.ComprobacionEcuacion(p, a, b)!=1);
77#
78#                     do{
79#                          cout<<"\nDame el valor de Qx: ";
80#                          f.leerPunto(q.x);
81#                          cout<<"Dame el valor de Qy: ";
82#                          f.leerPunto(q.y);
83#                          if(f.ComprobacionEcuacion(q, a, b)==0){
84#                               cout<<"\n*El punto Q no vive sobre la curva"<<endl;
85#                          }
86#                     }while(f.ComprobacionEcuacion(q, a, b)!=1);
87#
88#                     if(f.LambdaAdicion(S, p, q)==0){
89#                          cout<<"\tR(x,y) = ("<<p.x<<", "<<p.y<<") +
90#                               "<<"("<<q.x<<", "<<q.y<<") = ";
91#                          cout<<"Infinito";
92#                     }
93#                     else{
94#                          f.AdicionCE_GF2n(r, p, q, S, a);
95#                          cout<<"\n\tR(x,y) = ("<<p.x<<", "<<p.y<<") +
96#                               "<<"("<<q.x<<", "<<q.y<<") = ";
97#                          cout<<"("<<r.x<<", "<<r.y<<")"<<endl;
98#                          if(m<=10){
99#                               cout<<"\n***¿Deseas graficar la suma
100#                                   de puntos sobre la curva S/N?: >";
```

```
101#                        cin>>u;
102#                        if(u == 'S' || u == 's'){
103#                                f.GraficarPuntos(m,a,b);
104#                                Grafica.GraficaSuma(r,p,q);
105#                                cout<<"\nAhora abra octave y ejecuta:\t"
106#                                    <<"octave:1>GraficaGF2n\n";
107#                        }
108#                }
109#        }
110#        break;
111#        case 5:
112#                cout<<"\n**Doblado de Puntos Sobre La Curva Eliptica**"<<endl;
113#                cout<<"\tP(x,y) = 2P(x,y)"<<endl;
114#                do{
115#                        cout<<"\nDame el valor de Px: ";
116#                        f.leerPunto(p.x);
117#                        cout<<"Dame el valor de Py: ";
118#                        f.leerPunto(p.y);
119#                        if(f.ComprobacionEcuacion(p, a, b)==0){
120#                            cout<<"\n*El punto P no vive sobre la curva"<<endl;
121#                        }
122#                }while(f.ComprobacionEcuacion(p, a, b)!=1);
123#
124#                if(f.LambdaDoblado(S, p)==0){
125#                        cout<<"\t2P(x,y) = ("<<p.x<<", "<<p.y<<") = ";
126#                        cout<<"Infinito";
127#                }
128#                else{
129#                        f.DobladoCE_GF2n(r, p, S, a);
130#                        cout<<"\t2P(x,y) = ("<<p.x<<", "<<p.y<<") = ";
131#                        cout<<"("<<r.x<<", "<<r.y<<")"<<endl;
132#                        if(m<=10){
133#                                cout<<"\n***¿Deseas graficar el doblado
134#                                        del punto sobre la curva S/N?: >";
135#                                cin>>u;
136#                                if(u == 'S' || u == 's'){
137#                                        f.GraficarPuntos(m,a,b);
138#                                        Grafica.GraficaDoblado_Negativo(r, p);
139#                                        cout<<"\nAhora abra octave y ejecuta:
140#                                                \t"<<"octave:1>GraficaGF2n\n";
141#                                }
142#                        }
143#                }
144#        break;
145#        case 6:
146#                cout<<"\n*Negativo de un Punto Sobre La Curva Eliptica*"<<endl;
147#                do{
148#                        cout<<"\nDame el valor de Px: ";
149#                        f.leerPunto(p.x);
150#                        cout<<"Dame el valor de Py: ";
151#                        f.leerPunto(p.y);
152#                        if(f.ComprobacionEcuacion(p, a, b)==0){
153#                             cout<<"\n*El punto P no vive sobre la curva"<<endl;
154#                        }
155#                }while(f.ComprobacionEcuacion(p, a, b)!=1);
156#
157#                if(f.NegativoPuntoCE_GF2n(p, r)==0){
158#                        cout<<"\n*El punto P no tiene Negativo"<<endl;
159#                }
160#                else{
161#                        cout<<"\t-P(x,y) = -("<<p.x<<", "<<p.y<<") = ";
162#                        cout<<"("<<r.x<<", "<<r.y<<")"<<endl;
163#                        if(m<=10){
164#                                cout<<"\n***¿Deseas graficar el negativo
165#                                        del punto sobre la curva S/N?: >";
166#                                cin>>u;
```

```
167#                                        if(u == 'S' || u == 's'){
168#                                                f.GraficarPuntos(m,a,b);
169#                                                Grafica.GraficaDoblado_Negativo(r, p);
170#                                                cout<<"\nAhora abra octave y ejecuta:
171#                                                        \t"<<"octave:1>GraficaGF2n\n";
172#                                        }
173#                                }
174#                        }
175#                break;
176#                case 7:
177#                        cout<<"Introduce la cantidad de numeros aleatorios que deseas:
178#                            >>"; cin>>iter;
179#                        for(bigint i=0; i<iter; i++){
180#                                if((f.PuntosAleatoriosCE_GF2n(m, a, b, puntoG))==0){
181#                                        cout<<i+1<<"->";
182#                                        cout<<"("<<puntoG.x<<", "<<puntoG.y<<"); ";
183#                                }
184#                                else{i--;}
185#                        }
186#                        cout<<endl;
187#                break;
188#                case 8:
189#                        cout<<"\n\t0.Aleatoriamente\n\t1.Manualmente"<<endl<<" >";
190#                        cin>>iter;
191#                        if(iter == 0){
192#                                cout<<"\n**Punto Base P: ";
193#                                do{
194#                                        f.PuntosAleatoriosCE_GF2n(m, a, b, puntoG);
195#                                }while(puntoG.x.is_zero());
196#                                cout<<"("<<puntoG.x<<", "<<puntoG.y<<"); "<<endl;
197#                                f.LlavePrivadaCE_GF2n(m, a, k);
198#                                cout<<"**Llave Privada K = "<<k<<"; "<<endl;
199#                        }
200#                        else{
201#                                cout<<"\n**Punto Base P: ";
202#                                do{
203#                                        cout<<"\nDame el valor de Px: ";
204#                                        f.leerPunto(puntoG.x);
205#                                        cout<<"Dame el valor de Py: ";
206#                                        f.leerPunto(puntoG.y);
207#                                        if(f.ComprobacionEcuacion(puntoG, a, b)==0){
208#                                                cout<<"\n*El punto P no vive sobre
209#                                                        la curva"<<endl;
210#                                        }
211#                                }while(f.ComprobacionEcuacion(puntoG, a, b)!=1);
212#                                cout<<"("<<puntoG.x<<", "<<puntoG.y<<"); "<<endl;
213#                                cout<<"dame k: ";
214#                                cin>>k;
215#                                cout<<"**Llave Privada K = "<<k<<"; "<<endl;
216#
217#                        }
218#                        if(k.is_zero()){
219#                                q.x.assign_zero();
220#                                q.y.assign_zero();
221#                        }
222#                        else{
223#                              f.CadenaBinaria(k,cadBin,lim);
224#                              f.MultiplicacionEscalarCE_GF2n(q, puntoG, cadBin, lim, a);
225#                        }
226#                        cout<<"**Llave Publica Q = KP = ("<<q.x<<", "<<q.y<<"); ";
227#                        cout<<endl;
228#                        cout<<endl;
229#                break;
230#                case 9:
231#                        return;
232#                break;
```

```
233#                    default:
234#                            cout<<"Opcion invalida..."<<endl;
235#                    break;
236#            }
237#            cout<<"\n—>Cero para ver menu"<<endl;
238#    }while(1);
239#}
240#
241#void menu::mostrar_Menu(void){
242#            cout<<"\n\n\t|————————————————Menu————————————————|"<<endl;
243#            cout<<"\t|\t\tCurvas Elípticas Sobre Campos Finitos Binarios\t\t|"<<endl;
244#            cout<<"\t|\t\t          ******E:/GF(2^m)******       \t\t\t|"<<endl;
245#            cout<<"\t|\t\t     E: y^2 + x*y - [x^3 + a*x^2 + b]\t\t\t|"<<endl;
246#            cout<<"\t|\t\t\t\t\t\t\t\t\t|"<<endl;
247#            cout<<"\t|\t\t1. Definir m,a y b\t\t\t\t\t|"<<endl;
248#            cout<<"\t|\t\t2. Imprimir Puntos Existentes en la curva (m<=10)\t|"<<endl;
249#            cout<<"\t|\t\t3. Mostrar valores del campo\t\t\t\t|"<<endl;
250#            cout<<"\t|\t\t4. Sumar dos puntos sobre la curva\t\t\t|"<<endl;
251#            cout<<"\t|\t\t5. Duplicar punto sobre la curva\t\t\t|"<<endl;
252#            cout<<"\t|\t\t6. Negativo del punto P sobre la curva\t\t\t|"<<endl;
253#            cout<<"\t|\t\t7. Generacion finita de puntos sobre la curva\t\t|"<<endl;
254#            cout<<"\t|\t\t8. Multiplicación escalar\t\t\t\t|"<<endl;
255#            cout<<"\t|\t\t9. Salir\t\t\t\t\t\t|"<<endl;
256#            cout<<"\t|——————————————————————————————————————|"<<endl;
257#    return;
258#}
259#
260#void menu::iniciarValores(gf2n &a, gf2n &b){
261#    a = 8;
262#    b = 9;
263#    return;
264#}
265# ##################################################################################
```

---

Archivo: `Makefile` (*Archivo para la compilación del programa*)

```
 1# ##################################################################################
 2# CC=g++
 3# SRA=menu.cpp
 4# SRB=galois_f2n.cpp
 5# SRC=main.cpp
 6# SRD=AritmeticaCE_GF2n.cpp
 7# SRE=Graficar_GF2n.cpp
 8# #SRK=CampoGalois.cpp
 9# #SRL=Cola.cpp
10# #SRM=GraficarGF2n.cpp
11# #SRL=Aritmetica-CE-GF2n.cpp
12# LiDIA_FLAGS = -lLiDIA -lgmp -lm
13# all: cceGF2n
14#
15# cceGF2n: $(SRA) $(SRB) $(SRC) $(SRD) $(SRE)
16#         $(CC) -o $@ $^ $(LiDIA_FLAGS)
17#
18# clean:
19#         rm cceGF2n Puntos *.o GraficaGF2n.m
20# ##################################################################################
```

# Apéndice B

# Códigos fuentes

---

Archivo: `servidorCCE.x` (*Definición de todas las funciones necesarias para el envió de parámetros*)

```
1 # ###########################################################################
2 #struct Parametros{
3 #        int par;
4 #        char QX[235];
5 #        char QY[235];
6 #        char RX[235];
7 #        char RY[235];
8 #};
9 #struct puntoQ{
10 #        int ban;
11 #};
12 #program CCE_PROG {
13 #        version SERVIDORCCEVERS {
14 #                string CAMPO_M(Parametros) = 1;
15 #                string PUNTO_GX(Parametros) = 2;
16 #                string PUNTO_GY(Parametros) = 3;
17 #                string CONSTANTE_A(Parametros) = 4;
18 #                string CONSTANTE_B(Parametros) = 5;
19 #                string COFACTOR_H(Parametros) = 6;
20 #                string PRIMO_N(Parametros) = 7;
21 #                int SET_Q(Parametros) = 8;
22 #                string GET_QX(Parametros) = 9;
23 #                string GET_QY(Parametros) = 10;
24 #                int SET_R(Parametros) = 11;
25 #                string GET_RX(Parametros) = 12;
26 #                string GET_RY(Parametros) = 13;
27 #                int FINALIZAR_Q(Parametros) = 14;
28 #                int FINALIZAR_R(Parametros) = 15;
29 #                int LEER_ARCHIVO(Parametros) = 16;
30 #        } = 1;
31 #} = 0x20000001;
32 # ###########################################################################
```

---

Archivo: `servidorCCE.c` (*Funciones que permitirán el envió de parámetros del campo* $\mathbb{F}_{2^m}$ *hacia los clientes*)

```
1 # ##############################################################################
2 #char QX[400] = "0";
3 #char QY[400] = "0";
4 #char M1[1024] = "";
5 #
6 #char RX[400] = "0";
7 #char RY[400] = "0";
8 #char M2[1024] = "";
9 #
10 #char valores[8][400];
11 #
12 #int * leer_archivo_1_svc(Parametros * operandos, struct svc_req * req){
13 #       static int ban = 0;
14 #       FILE *f = fopen("parametros", "r");
15 #       if (f == NULL){
16 #               printf("Error");
17 #               return &ban;
18 #       }
19 #       fgets(valores[0], 400, f);
20 #       fgets(valores[1], 400, f);
21 #       fgets(valores[2], 400, f);
22 #       fgets(valores[3], 400, f);
23 #       fgets(valores[4], 400, f);
24 #       fgets(valores[5], 400, f);
25 #       fgets(valores[6], 400, f);
26 #       fclose(f);
27 #
28 #       ban = 1;
29 #
30 #       return &ban;
31 #}
32 #/*Regresa el valor de m para definir el campo finito binario GF(2^m)*/
33 #char ** campo_m_1_svc(Parametros * operandos, struct svc_req * req){
34 #       static char *result;
35 #       result = valores[0];
36 #       return &result;
37 #}
38 #
39 #/*Regresa el valor 'a' en la ecuacion de y^2 + x*y = x^3 + a*x^2 + b */
40 #char ** constante_a_1_svc(Parametros * operandos, struct svc_req * req){
41 #       static char *result;
42 #       result = valores[1];
43 #       return &result;
44 #}
45 #
46 #/*Regresa el valor 'b' en la ecuacion de y^2 + x*y = x^3 + a*x^2 + b */
47 #char ** constante_b_1_svc(Parametros * operandos, struct svc_req * req){
48 #       static char *result;
49 #       result = valores[2];
50 #       return &result;
51 #}
52 #
53 #/*G es un punto que vive sobre el campo GF(2^m)*/
54 #char ** punto_gx_1_svc(Parametros * operandos, struct svc_req * req){
55 #       static char* GX = valores[3];
56 #       return &GX;
57 #}
58 #
59 #/*G es un punto que vive sobre el campo GF(2^m)*/
60 #char ** punto_gy_1_svc(Parametros * operandos, struct svc_req * req){
61 #       static char* GY = valores[4];
62 #       return &GY;
63 #}
64 #
65 #/*h es el cofactor sobre el campo GF(2^m)*/
66 #char ** cofactor_h_1_svc(Parametros * operandos, struct svc_req * req){
```

```
 67#        static char* h = valores[5];
 68#        return &h;
 69#}
 70#
 71#/*n es el numero primo tal que Q = nG = punto al infinito*/
 72#char ** primo_n_1_svc(Parametros * operandos, struct svc_req * req){
 73#        static char* n = valores[6];
 74#        return &n;
 75#}
 76#
 77#int * set_q_1_svc(Parametros * operandos, struct svc_req * req){
 78#        static int qr;
 79#        qr = 1;
 80#        strcpy(QX, operandos->QX);
 81#        strcpy(QY, operandos->QY);
 82#        printf("HOST 1 : Q = \t(%s", QX);
 83#        printf(", %s)\n", QY);
 84#        if(QX == "0" && QY == ""){
 85#                qr = 0;
 86#        }
 87#        return &qr;
 88#}
 89#
 90#char ** get_qx_1_svc(Parametros * operandos, struct svc_req * req){
 91#        static char *qx2 = "0";
 92#        qx2 = QX;
 93#        return &qx2;
 94#}
 95#char ** get_qy_1_svc(Parametros * operandos, struct svc_req * req){
 96#        static char *qy2 = "0";
 97#        qy2 = QY;
 98#        return &qy2;
 99#}
100#
101#
102#int * set_r_1_svc(Parametros * operandos, struct svc_req * req){
103#        static int qr;
104#        qr = 1;
105#        strcpy(RX, operandos->RX);
106#        strcpy(RY, operandos->RY);
107#        printf("HOST 2 : R = \t(%s", RX);
108#        printf(", %s)\n", RY);
109#        if(RX == "0" && RY == "0"){
110#                qr = 0;
111#        }
112#        return &qr;
113#}
114#
115#char ** get_rx_1_svc(Parametros * operandos, struct svc_req * req){
116#        static char *rx2 = "0";
117#        rx2 = RX;
118#        return &rx2;
119#}
120#char ** get_ry_1_svc(Parametros * operandos, struct svc_req * req){
121#        static char *ry2 = "0";
122#        ry2 = RY;
123#        return &ry2;
124#}
125#
126#int * finalizar_r_1_svc(Parametros *operandos, struct svc_req * req){
127#        static int r;
128#        r = 1;
129#        return &r;
130#}
131#int * finalizar_q_1_svc(Parametros *operandos, struct svc_req * req){
132#        static int q;
```

```
133#        q = 1;
134#        strcpy(QX, "0");
135#        strcpy(QY, "0");
136#        strcpy(RX, "0");
137#        strcpy(RY, "0");
138#        return &q;
139#}
140# ################################################################################
```

Archivo: `conexion.cpp` (*Función que permite abrir el canal de comunicación*)

```
1# ################################################################################
2#CLIENT *conexion(char **server){
3#        CLIENT *clnt = clnt_create(*server, CCE_PROG, SERVIDORCCEVERS, "tcp");
4#
5#        if(clnt == NULL){
6#                cerr <<"Error en la creacion del cliente de rpc"<<endl;
7#                clnt_pcreateerror(*server);
8#                return (clnt);
9#        }
10#        return (clnt);
11#}
12# ################################################################################
```

Archivo: `defineCampo.cpp` (*Definición de los parametrós necesarios sobre el campo* $\mathbb{F}_{2^m}$)

```
1# ################################################################################
2#int defineCampo(char *n_tupla[max], CLIENT **clnt){
3#        Parametros p;
4#        char **a, **b, **GX, **GY, **m, **h, **n;
5#
6#        leer_archivo_1(&p,*clnt);
7#        m = campo_m_1(&p, *clnt);
8#        a = constante_a_1(&p, *clnt);
9#        b = constante_b_1(&p, *clnt);
10#        GX = punto_gx_1(&p, *clnt);
11#        GY = punto_gy_1(&p, *clnt);
12#        h = cofactor_h_1(&p, *clnt);
13#        n = primo_n_1(&p, *clnt);
14#
15#        char *m2, *a2, *b2, *GX2, *GY2, *n2, *h2;
16#        m2 = *m;
17#        a2 = *a;
18#        b2 = *b;
19#        GX2 = *GX;
20#        GY2 = *GY;
21#        h2 = *h;
22#        n2 = *n;
23#        m2[strlen(m2)-1] = '\0';
24#        a2[strlen(a2)-1] = '\0';
25#        b2[strlen(b2)-1] = '\0';
26#        GX2[strlen(GX2)-1] = '\0';
27#        GY2[strlen(GY2)-1] = '\0';
28#        h2[strlen(h2)-1] = '\0';
```

```
29#         n2[strlen(n2)-1] = '\0';
30#
31#         cout<<"\t\tGF(2^"<<m2<<")"<<endl;
32#         cout<<"\ty^2 + x*y = x^3 + "<<a2<<"*x^2 + "<<b2<<endl;
33#         cout<<"\tCofactor h = "<<h2<<"\t, # Primo n = "<<n2<<endl;
34#         cout<<" a = "<<*a<<endl;
35#         cout<<" b = "<<*b<<endl;*/
36#         cout<<"\t\tG = ("<<GX2<<", "<<GY2<<")"<<endl;
37#         if(m == NULL && a == NULL && b == NULL && GX == NULL && GY == NULL && h == NULL && n == NULL){
38#                 cerr<<"Error al llamar las  funciones"<<endl;
39#                 return -1;
40#         }
41#         else{
42#                 n_tupla[0] = &(**m);
43#                 n_tupla[1] = &(**a);
44#                 n_tupla[2] = &(**b);
45#                 n_tupla[3] = &(**GX);
46#                 n_tupla[4] = &(**GY);
47#                 n_tupla[5] = &(**h);
48#                 n_tupla[6] = &(**n);
49#                 return 1;
50#         }
51#}
52# ##############################################################################
```

---

Archivo: `ecc_DH.cpp` (*Funciónes necesarias para la creación de llaves*)

```
 1# ##############################################################################
 2#bigint llavePrivadaK(char *n_tupla[lim]){
 3#         int caracteristica = 2;
 4#         lidia_size_t m;
 5#         gf2n a;
 6#         bigint k;
 7#
 8#         m = atoi(n_tupla[0]);
 9#         m_Galois_F2n(m);
10#         c_Galois_F2n(caracteristica, m);
11#
12#         convertirGf2n(a, n_tupla[1]);
13#
14#         LlavePrivadaCE_GF2n(m, a, k, n_tupla[6]);
15#         return k;
16#}
17#void llavePublicaA(bigint k, char *n_tupla[lim], char gx[235], char gy[235]){
18#         gf2n a;
19#         Punto G, q;
20#         char cadBin[235];
21#         int limite;
22#
23#         convertirGf2n(a, n_tupla[1]);
24#
25#         convertirGf2n(G.x, n_tupla[3]);
26#
27#         convertirGf2n(G.y, n_tupla[4]);
28#
29#         CadenaBinaria(k,cadBin,limite);
30#         MultiplicacionEscalarCE_GF2n(q, G, cadBin, limite, a);
31#         convertirString(q.x, gx);
32#         convertirString(q.y, gy);
33#}
34#void m_Galois_F2n(lidia_size_t &m){
```

```
35#        gf2n_init(m);
36#         return;
37#}
38#
39#void c_Galois_F2n(int &caracteristica, lidia_size_t &m){
40#         gf2n_init(m);
41#         galois_field campo(caracteristica, m);
42#         /*galois_f2n field;
43#         field.polinomio_Irreducible(campo);
44#         field.combinaciones_Galois_F2n(campo);*/
45#         return;
46#}
47#
48#
49#void llavePublica(bigint k, char QR_x[235],char QR_y[235], char sx[235], char sy[235], char *n_tupla[lim]
50#        gf2n a;
51#        Punto QR, s;
52#        char cadBin[235];
53#        int limite;
54#
55#        convertirGf2n(a, n_tupla[1]);
56#
57#        convertirGf2n(QR.x, QR_x);
58#
59#        convertirGf2n(QR.y, QR_y);
60#
61#        CadenaBinaria(k,cadBin,limite);
62#        MultiplicacionEscalarCE_GF2n(s, QR, cadBin, limite, a);
63#        convertirString(s.x, sx);
64#        convertirString(s.y, sy);
65#}
66# ###############################################################################
```

Archivo: `valoresCampo_1.cpp` (*Establece la comunicación entre el primer cliente con los parámetros necesarios*)

```
 1# ###############################################################################
 2#int asignarQ(char QX[235], char QY[235], CLIENT **clnt){
 3#        Parametros p;
 4#        int *result;
 5#        result = 0;
 6#        strcpy(p.QX, QX);
 7#        strcpy(p.QY, QY);
 8#        result = set_q_1(&p, *clnt);
 9#}
10#int regresaRX(CLIENT **clnt, char r_x[235]){
11#        Parametros p;
12#        char **RX;
13#        RX = get_rx_1(&p, *clnt);
14#        strcpy(r_x, *RX);
15#        if(strcmp(*RX, "0") == 0){
16#                return 0;
17#        }
18#        return 1;
19#}
20#int regresaRY(CLIENT **clnt, char r_y[235]){
21#        Parametros p;
22#        char **RY;
23#        RY = get_ry_1(&p, *clnt);
24#        strcpy(r_y, *RY);
25#        if(strcmp(*RY,"0") == 0){
```

```
26#                 return 0;
27#         }
28#         return 1;
29#}
30#
31#int finalizarQ(CLIENT **clnt){
32#         Parametros p;
33#         finalizar_q_1(&p, *clnt);
34#}
35#
36#int finalizarR(CLIENT **clnt){
37#         Parametros p;
38#         finalizar_r_1(&p, *clnt);
39#}
40# ##############################################################################
```

---

Archivo: `valoresCampo_2.cpp` (*Establece la comunicación entre el segundo cliente con los parámetros necesarios*)

```
 1# ##############################################################################
 2#int asignarR(char RX[235], char RY[235], CLIENT **clnt){
 3#         Parametros p;
 4#         int *result;
 5#         result = 0;
 6#         strcpy(p.RX, RX);
 7#         strcpy(p.RY, RY);
 8#         result = set_r_1(&p, *clnt);
 9#}
10#
11#int regresaQX(CLIENT **clnt, char q_x[235]){
12#         Parametros p;
13#         char **QX;
14#         QX = get_qx_1(&p, *clnt);
15#         strcpy(q_x, *QX);
16#         if(strcmp(*QX, "0") == 0){
17#                 return 0;
18#         }
19#         return 1;
20#}
21#
22#int regresaQY(CLIENT **clnt, char q_y[235]){
23#         Parametros p;
24#         char **QY;
25#         QY = get_qy_1(&p, *clnt);
26#         strcpy(q_y, *QY);
27#         if(strcmp(*QY, "0") == 0){
28#                 return 0;
29#         }
30#         return 1;
31#}
32#
33#int finalizarR(CLIENT **clnt){
34#         Parametros p;
35#         finalizar_r_1(&p, *clnt);
36#}
37#
38#int finalizarQ(CLIENT **clnt){
39#         Parametros p;
40#         finalizar_q_1(&p, *clnt);
41#}
42# ##############################################################################
```

Archivo: `maquina1.cpp` (*Funciones que realizan la petición de parámetros hacia el servidor del primer cliente*)

```
 1# ##################################################################################
 2#define maximo 8
 3#using namespace std;
 4#int main(int argc, char* argv[]){
 5#        if(argc != 2){
 6#                cerr<<"Uso: ./maquina1  direccion"<<endl;
 7#                return 1;
 8#        }
 9#        cout<<"\t\tMAQUINA1\n";
10#
11#        CLIENT *conn;
12#        int cofactor;
13#        char *n_tupla[maximo];
14#        char qx[235], qy[235];
15#        bigint k;
16#
17#        conn = conexion(&argv[1]);
18#        defineCampo(&(*n_tupla), &conn);
19#
20#        k = llavePrivadaK(&(*n_tupla));
21#        cofactor = atoi(n_tupla[5]);
22#        cout<<"k = \t\t\t "<<k<<endl;
23#        k *= cofactor;
24#        cout<<"k_1 = Cofactor*K = \t "<<k<<endl;
25#
26#        llavePublicaA(k, &(*n_tupla), qx, qy);
27#
28#        cout<<"Q = K_1*G =\t\t ("<<qx<<", "<<qy<<")"<<endl;
29#
30#        char r_x[235], r_y[235];
31#        char sx[235] = "0", sy[235] = "0";
32#        int res = 0, rx, ry;
33#        int pid, status;
34#        bigint cont = 0;
35#        if((pid = fork())==0){
36#                while(res == 0 && cont < 1000000000){
37#                        res = asignarQ(qx,qy, &conn);
38#                        sleep(3);
39#                        cont++;
40#                }
41#        }
42#        else{
43#                waitpid(pid, &status, 0);
44#                rx = regresaRX(&conn, r_x);
45#
46#                ry = regresaRY(&conn, r_y);
47#
48#                cout<<"R = k_2*G =\t\t ("<<r_x<<", "<<r_y<<")"<<endl;
49#
50#                if(rx == 0 && ry == 0){
51#                        cout<<"S = k_1*R =\t\t ("<<sx<<", "<<sy<<")"<<endl;
52#                        cout<<"El host 2 No responde\n";
53#                        finalizarQ(&conn);
54#                }
55#                else{
56#                        llavePublica(k, r_x, r_y, sx, sy, &(*n_tupla));
57#                        cout<<"S = k_1*R =\t\t ("<<sx<<", "<<sy<<")"<<endl;
58#
```

```
59#                 }
60#                 finalizarR(&conn);
61          }
62#}
63# ################################################################################
```

Archivo: `maquina2.cpp` (*Funciones que realizan la petición de parámetros hacia el servidor del segundo cliente*)

```
 1# ################################################################################
 2#define maximo 8 //TIENE EL MISMO TAMANIO EN EL ARCHIVO 'defineCampo.h'
 3#using namespace std;
 4#int main(int argc, char* argv[]){
 5#         if(argc != 2){
 6#                 cerr<<"Uso: ./maquina1  direccion"<<endl;
 7#                 return 1;
 8#         }
 9#
10#         cout<<"\t\tMAQUINA2\n";
11#
12#         CLIENT *conn;
13#         int cofactor;
14#         char *n_tupla[maximo];
15#         char rx[235], ry[235];
16#         bigint k;
17#
18#         conn = conexion(&argv[1]);
19#         defineCampo(&(*n_tupla), &conn);
20#
21#         k = llavePrivadaK(&(*n_tupla));
22#         cofactor = atoi(n_tupla[5]);
23#         cout<<"K = \t\t\t "<<k<<endl;
24#         k *= cofactor;
25#         cout<<"k_2 = Cofactor*K = \t "<<k<<endl;
26#
27#         llavePublicaA(k, &(*n_tupla), rx, ry);
28#
29#
30#         cout<<"R = K_2*G =\t\t ("<<rx<<", "<<ry<<")"<<endl;
31#
32#         char q_x[235], q_y[235];
33#         char sx[235] = "0", sy[235] = "0";
34#         int res = 0, qx, qy;
35#         int pid, status;
36#         bigint cont = 0;
37#         if((pid = fork())==0){
38#                 while(res == 0 && cont < 1000000000){
39#                         res = asignarR(rx,ry, &conn);
40#                         sleep(2);
41#                         cont++;
42#                 }
43#         }
44#         else{
45#                 waitpid(pid, &status, 0);
46#
47#                 qx = regresaQX(&conn, q_x);
48#                 qy = regresaQY(&conn, q_y);
49#                 cout<<"Q = K_1*G =\t\t ("<<q_x<<", "<<q_y<<")"<<endl;
50#
51#                 if(qx == 0 && qy == 0){
52#                         cout<<"S = k_2*Q =\t\t ("<<sx<<", "<<sy<<")"<<endl;
```

```
53#                     cout<<"El host 1 No responde\n";
54#                     finalizarR(&conn);
55#             }
56#             else{
57#                     llavePublica(k, q_x, q_y, sx, sy, &(*n_tupla));
58#                     cout<<"S = k_2*Q =\t\t ("<<sx<<", "<<sy<<")"<<endl;
59#             }
60#
61#             finalizarQ(&conn);
62#     }
63#}
64# ##################################################################################
```

Archivo: `Makefile` (*Archivo que permite la compilación de todos los archivos fuente del segundo programa*)

```
 1# ##################################################################################
 2#       # host@usuario$ rpcgen -C servidorCCE.x
 3#
 4# #cc servidorCCE.c servidorCCE_svc.c servidorCCE_xdr.c -o servidorCCE
 5#CC = cc
 6#arcc0 = servidorCCE.c
 7#arcc1 = servidorCCE_svc.c
 8#arcc2 = servidorCCE_xdr.c
 9#all: servidorCCE
10#servidorCCE:  $(arcc0) $(arcc1) $(arcc2)
11#       $(CC) -o $@ $^
12#
13# #g++ maquina1.cpp servidorCCE_clnt.c servidorCCE_xdr.c -o cliente
14#GCC = g++
15#arc0 = maquina1.cpp
16#arc1 = servidorCCE_clnt.c
17#arc2 = servidorCCE_xdr.c
18#arc3 = conexion.cpp
19#arc4 = defineCampo.cpp
20#arc5 = ecc_DH.cpp
21#arc6 = AritmeticaCE_GF2n.cpp
22#
23#arc7 = valoresCampo_1.cpp
24#LiDIA_FLAGS = -lLiDIA -lgmp -lm
25#all: host1
26#host1: $(arc0) $(arc1) $(arc2) $(arc3) $(arc4) $(arc5) $(arc6) $(arc7)
27#       $(GCC) -o $@ $^ $(LiDIA_FLAGS)
28#
29#arcM2 = maquina2.cpp
30#arcM2_7 = valoresCampo_2.cpp
31#all: host2
32#host2: $(arcM2) $(arc1) $(arc2) $(arc3) $(arc4) $(arc5) $(arc6) $(arcM2_7)
33#       $(GCC) -o $@ $^ $(LiDIA_FLAGS)
34#clean:
35#       rm servidorCCE servidorCCE_svc.c servidorCCE_xdr.c host1 host2 servidorCCE.h servidorCCE_clnt.c
36# ##################################################################################
```

# Apéndice C

# Códigos fuentes

---

Archivo: `GF2n.vhdl` (*Funciones que realizan las multiplicaciones de Karatsuba y la reducción modular además de la operación de elevar al cuadrado*)

```
 1# ##############################################################################
 2#library ieee;
 3#use ieee.std_logic_1164.all;
 4#use ieee.std_logic_arith.all;
 5#use ieee.std_logic_unsigned.all;
 6#
 7#package GF2n is
 8#       function cuadrado (a: bit_vector(232 downto 0)) return bit_vector;
 9#       function M4b (a, b: bit_vector(3 downto 0)) return bit_vector;
10#       function M8b (a, b: bit_vector(7 downto 0)) return bit_vector;
11#       function M16b (a, b: bit_vector(15 downto 0)) return bit_vector;
12#       function M32b (a, b: bit_vector(31 downto 0)) return bit_vector;
13#       function M64b (a, b: bit_vector(63 downto 0)) return bit_vector;
14#       function M128b (a, b: bit_vector(127 downto 0)) return bit_vector;
15#       function M233b (a, b: bit_vector(232 downto 0)) return bit_vector;
16#       function RM (red: bit_vector(464 downto 0)) return bit_vector;
17#end GF2n;
18#
19#package body GF2n is
20#       function M4b(a, b: bit_vector(3 downto 0)) return bit_vector is
21#               variable x, x2, x3, y, y2, y3, z,z2, z3: bit_vector(2 downto 0):=(others => '0');
22#               variable xr, yr, zr: bit_vector(6 downto 0):=(others => '0');
23#               variable c: bit_vector(7 downto 0):=(others => '0');
24#       begin
25#               x(0) := a(2) and b(2);
26#               x2(1) := (a(3) and b(2)) xor (a(2) and b(3));
27#               x3(2) := a(3) and b(3);
28#               xr(6 downto 4) := (x xor x2) xor x3;
29#
30#               y(0) := (a(2) and b(0)) xor (a(0)and b(2));
31#               y2(1) := ((a(3)and b(0))xor(a(2)and b(1))) xor ((a(1)and b(2))xor(a(0)and b(3)));
32#               y3(2) := (a(3)and b(1))xor(a(1)and b(3));
33#               yr(4 downto 2) := (y2 xor y3) xor y;
34#
35#               z(0) := a(0) and b(0);
36#               z2(1) := (a(1)and b(0))xor(a(0)and b(1));
37#               z3(2) := a(1)and b(1);
38#               zr(2 downto 0) := (z2 xor z3) xor z;
```

```
39#
40#             c(6 downto 0) := (zr xor yr) xor xr;
41#             return c;
42#      end M4b;
43#
44#      function M8b(a, b: bit_vector(7 downto 0)) return bit_vector is
45#             variable CH, CL, CM, BHAL, M: bit_vector(7 downto 0);
46#             variable karatsuba: bit_vector(15 downto 0);
47#      begin
48#             --AH * BH
49#             CH := M4b(a(7 downto 4), b(7 downto 4));
50#
51#             --AL * BL
52#             CL := M4b(a(3 downto 0), b(3 downto 0));
53#
54#             --(AH + AL) * (BH + BL)
55#             CM := M4b((a(7 downto 4) xor a(3 downto 0)),(b(7 downto 4)xor b(3 downto 0)));
56#
57#             --CORRIMIENTOS
58#
59#             --CH_L & CL_H
60#             BHAL := CH(3 downto 0) & CL(7 downto 4);
61#
62#             --M1 + M2 + M3 + M4
63#             M := ((CH xor CL) xor CM) xor BHAL;
64#
65#             --CH_H & M & CL_L
66#             karatsuba := CH(7 downto 4) & M & CL(3 downto 0);
67#
68#             return karatsuba;
69#      end M8b;
70#
71#      function M16b(a, b: bit_vector(15 downto 0)) return bit_vector is
72#             variable CH, CL, CM, BHAL, M: bit_vector(15 downto 0);
73#             variable karatsuba: bit_vector(31 downto 0);
74#      begin
75#             CH := M8b(a(15 downto 8), b(15 downto 8));
76#             CL := M8b(a(7 downto 0), b(7 downto 0));
77#
78#             CM := M8b((a(15 downto 8) xor a(7 downto 0)),(b(15 downto 8)xor b(7 downto 0)));
79#
80#             BHAL := CH(7 downto 0) & CL(15 downto 8);
81#             M := ((CH xor CL) xor CM) xor BHAL;
82#             karatsuba := CH(15 downto 8) & M & CL(7 downto 0);
83#             return karatsuba;
84#      end M16b;
85#
86#      function M32b(a, b: bit_vector(31 downto 0)) return bit_vector is
87#             variable CH, CL, CM, BHAL, M: bit_vector(31 downto 0);
88#             variable karatsuba: bit_vector(63 downto 0);
89#      begin
90#             CH := M16b(a(31 downto 16), b(31 downto 16));
91#             CL := M16b(a(15 downto 0), b(15 downto 0));
92#
93#             CM := M16b((a(31 downto 16) xor a(15 downto 0)),(b(31 downto 16)xor b(15 downto 0)));
94#
95#             BHAL := CH(15 downto 0) & CL(31 downto 16);
96#             M := ((CH xor CL) xor CM) xor BHAL;
97#             karatsuba := CH(31 downto 16) & M & CL(15 downto 0);
98#             return karatsuba;
99#      end M32b;
100#
101#     function M64b(a, b: bit_vector(63 downto 0)) return bit_vector is
102#            variable CH, CL, CM, BHAL, M: bit_vector(63 downto 0);
103#            variable karatsuba: bit_vector(127 downto 0);
104#     begin
```

```
105#                CH := M32b(a(63 downto 32), b(63 downto 32));
106#                CL := M32b(a(31 downto 0), b(31 downto 0));
107#
108#                CM := M32b((a(63 downto 32) xor a(31 downto 0)),(b(63 downto 32)xor b(31 downto 0)));
109#
110#                BHAL := CH(31 downto 0) & CL(63 downto 32);
111#                M := ((CH xor CL) xor CM) xor BHAL;
112#                karatsuba := CH(63 downto 32) & M & CL(31 downto 0);
113#                return karatsuba;
114#        end M64b;
115#
116#        function M128b(a, b: bit_vector(127 downto 0)) return bit_vector is
117#                variable CH, CL, CM, BHAL, M: bit_vector(127 downto 0);
118#                variable karatsuba: bit_vector(255 downto 0);
119#        begin
120#                CH := M64b(a(127 downto 64), b(127 downto 64));
121#                CL := M64b(a(63 downto 0), b(63 downto 0));
122#
123#                CM := M64b((a(127 downto 64) xor a(63 downto 0)),(b(127 downto 64)xor b(63 downto 0)));
124#
125#                BHAL := CH(63 downto 0) & CL(127 downto 64);
126#                M := ((CH xor CL) xor CM) xor BHAL;
127#                karatsuba := CH(127 downto 64) & M & CL(63 downto 0);
128#                return karatsuba;
129#        end M128b;
130#
131#        function M233b(a, b: bit_vector(232 downto 0)) return bit_vector is
132#                variable CH, CL, CM, BHAL, M: bit_vector(255 downto 0);
133#                variable karatsuba: bit_vector(464 downto 0);
134#                variable ceros: bit_vector(22 downto 0):=(others => '0');
135#        begin
136#                CH := M128b(ceros & a(232 downto 128), ceros & b(232 downto 128));
137#                CL := M128b(a(127 downto 0), b(127 downto 0));
138#
139#                CM := M128b((( ceros & a(232 downto 128)) xor a(127 downto 0)), \
140#                ((ceros & b(232 downto 128))xor b(127 downto 0)));
141#
142#                BHAL := CH(127 downto 0) & CL(255 downto 128);
143#                M := ((CH xor CL) xor CM) xor BHAL;
144#                karatsuba := CH(208 downto 128) & M & CL(127 downto 0);
145#                return karatsuba;
146#        end M233b;
147#
148#        function RM (red: bit_vector(464 downto 0)) return bit_vector is
149#                variable c1, c5: bit_vector(232 downto 0):=(others=>'0');
150#                variable c2: bit_vector(158 downto 0):=(others=>'0');
151#                variable c3, c4: bit_vector(72 downto 0):=(others=>'0');
152#        begin
153#                c1(232) := red(232);
154#                c1(231 downto 0) := red(231 downto 0) xor red(464 downto 233);
155#                c2 := c1(232 downto 74) xor red(391 downto 233);
156#                c3 := c1(72 downto 0) xor red(464 downto 392);
157#                c4 := c2(72 downto 0) xor red(464 downto 392);
158#                c5 := c2(158 downto 73) & c4 & c1(73) & c3;
159#                return c5;
160#        end RM;
161#
162#        function cuadrado(a: bit_vector(232 downto 0)) return bit_vector is
163#                variable c:bit_vector(232 downto 0);
164#                variable binario: std_logic_vector(7 downto 0);
165#        begin
166#                for i in 0 to 73 loop
167#                        binario := conv_std_logic_vector(i,8);
168#                        if binario(0) = '0' then
169#                                c(i) := a(i/2) xor a((i + 392) / 2);
170#                        else
```

```
171#                                c(i) := a((i + 233)/2);
172#                        end if;
173#                end loop;
174#
175#                for i in 74 to 146 loop
176#                        binario := conv_std_logic_vector(i,8);
177#                        if binario(0) = '0' then
178#                                c(i) := a(i/2) xor a((i + 318) / 2);
179#                        else
180#                                c(i) := a((i + 233)/2) xor a((i+159)/2);
181#                        end if;
182#                end loop;
183#
184#                for i in 147 to 232 loop
185#                        binario := conv_std_logic_vector(i,8);
186#                        if binario(0) = '0' then
187#                                c(i) := a(i/2);
188#                        else
189#                                c(i) := a((i + 233)/2) xor a((i+159)/2);
190#                        end if;
191#                end loop;
192#                return c;
193#        end cuadrado;
194#end GF2n;
195# ################################################################################
```

# Apéndice D

# Códigos fuentes

---

Archivo: `M23.nb` (*Multiplicación escalar sobre* $E(\mathbb{F}_{2^{23}})$)

```
1 # ################################################################################
2 #m=23;
3 #k="11111111111111011010011";
4 #lim = StringLength[k] + 1;
5 #a=0;
6 #Gx=z^22 + z^21 + z^20 + z^18 + z^14 + z^10 + z^7 + z^6 + z^3 + z + 1;
7 #Gy=z^21 + z^20 + z^19 + z^18 + z^17 + z^15 + z^10 + z^8 + z^7 + z^6 + z^5 + z^2 + 1;
8 #Rx=Gx;
9 #Ry=Gy;
10 #PolinomioIrreducible=z^23 + z^5 + 1;
11 #
12 #For[i=2,i<lim, i++,
13 #        c=StringTake[k,{i}];
14 #        (*Doblado*)
15 #        {d,{inv,u}}=PolynomialMod[PolynomialExtendedGCD[Rx, PolinomioIrreducible],2];
16 #        Lamda=PolynomialMod[(Rx + Ry*inv),{PolinomioIrreducible,2}];
17 #        X3=PolynomialMod[(Lamda^2 + Lamda + a),{PolinomioIrreducible,2}];
18 #        Y3=PolynomialMod[(Rx^2 + Lamda*X3 + X3),{PolinomioIrreducible,2}];
19 #        Rx = X3;
20 #        Ry = Y3;
21 #        If[c=="1",{
22 #                (*Suma*)
23 #                {d,{inv2,u}}=PolynomialMod[PolynomialExtendedGCD[Gx + Rx, PolinomioIrreducible],2];
24 #                Lamda2=PolynomialMod[(Gy + Ry)*inv2,{PolinomioIrreducible,2}];
25 #                XX3=PolynomialMod[(Lamda2^2 + Lamda2 + Gx + Rx + a),{PolinomioIrreducible,2}];
26 #                YY3=PolynomialMod[Lamda2*(Gx + XX3) + XX3 + Gy,{PolinomioIrreducible,2}];
27 #                Rx = XX3;
28 #                Ry = YY3;
29 #                },{0}
30 #        ]
31 #]
32 #Print[Rx]
33 #Print[Ry]
34 #
35 #During evaluation of In[12]:= 1 + z + z^3 + z^6 + z^7 + z^11 + z^17 + z^18
36 #
37 #During evaluation of In[12]:= z^2 + z^3 + z^6 + z^8 + z^10 + z^12 + z^14 + z^16 + z^21 + z^22
38 # ################################################################################
```

Archivo: `M89.nb` (*Multiplicación escalar sobre* $E(\mathbb{F}_{2^{89}})$)

```
1 # ################################################################################
2 #m = 89;
3 #k = "1111111111111111111111111111111111111111111111111111111111111111110011010010001100 \
4 #1010110";
5 #lim = StringLength[k] + 1;
6 #a = 0;
7 #Gx = z^85 + z^84 + z^82 + z^81 + z^79 + z^78 + z^77 + z^73 + z^72 + z^71 + z^70 + z^66 + \
8 #z^65 + z^62 + z^59 + z^58 + z^57 + z^55 + z^45 + z^41 + z^39 + z^37 + z^34 + z^32 + z^29 + \
9 #z^28 + z^27 + z^26 + z^25 + z^22 + z^20 + z^17 + z^11 + z^10 + z^7 + z^6;
10 #
11 #Gy = z^87 + z^82 + z^78 + z^76 + z^75 + z^73 + z^72 + z^71 + z^70 + z^67 + z^65 + z^63 + \
12 #z^60 + z^59 + z^58 + z^57 + z^56 + z^55 + z^54 + z^53 + z^51 + z^48 + z^47 + z^46 + z^41 + \
13 #z^39 + z^38 + z^37 + z^36 + z^35 + z^33 + z^32 + z^31 + z^30 + z^29 + z^28 + z^26 + z^25 + \
14 #z^22 + z^21 + z^17 + z^14 + z^4 + z;
15 #Rx = Gx;
16 #Ry = Gy;
17 #PolinomioIrreducible = z^{89} + z^{38} + 1;
18 #
19 #For[i=2,i<lim,i++,
20 #        c=StringTake[k,{i}];
21 #        (*Doblado*)
22 #        {d,{inv,u}}=PolynomialMod[PolynomialExtendedGCD[Rx, PolinomioIrreducible],2];
23 #        Lamda=PolynomialMod[(Rx + Ry*inv),{PolinomioIrreducible,2}];
24 #        X3=PolynomialMod[(Lamda^2 + Lamda + a),{PolinomioIrreducible,2}];
25 #        Y3=PolynomialMod[(Rx^2 + Lamda*X3 + X3),{PolinomioIrreducible,2}];
26 #        Rx = X3;
27 #        Ry = Y3;
28 #        If[c=="1",{
29 #                (*Suma*)
30 #                {d,{inv2,u}}=PolynomialMod[PolynomialExtendedGCD[Gx + Rx, PolinomioIrreducible],2];
31 #                Lamda2=PolynomialMod[(Gy + Ry)*inv2,{PolinomioIrreducible,2}];
32 #                XX3=PolynomialMod[(Lamda2^2 + Lamda2 + Gx + Rx + a),{PolinomioIrreducible,2}];
33 #                YY3=PolynomialMod[Lamda2*(Gx + XX3) + XX3 + Gy,{PolinomioIrreducible,2}];
34 #                Rx = XX3;
35 #                Ry = YY3;
36 #                },{0}
37 #        ]
38 #]
39 #Print[Rx]
40 #Print[Ry]
41 #
42 #During evaluation of In[12]:= z^88 + z^83 + z^81 + z^80 + z^78 + z^76 + z^75 + z^73 + z^72 + \
43 #z^70 + z^69 + z^67 + z^66 + z^65 + z^63 + z^59 + z^57 + z^56 + z^55 + z^48 + z^47 + z^46 + \
44 #z^45 + z^44 + z^43 + z^41 + z^39 + z^38 + z^36 + z^35 + z^34 + z^33 + z^32 + z^31 + z^29 + \
45 #z^28 + z^27 + z^26 + z^25 + z^21 + z^20 + z^19 + z^18 + z^15 + z^14 + z^13 + z^12 + z^10 + z
46 #
47 #During evaluation of In[12]:= z^86 + z^85 + z^83 + z^82 + z^78 + z^74 + z^72 + z^68 + z^67 + \
48 #z^66 + z^65 + z^59 + z^58 + z^54 + z^53 + z^52 + z^51 + z^50 + z^49 + z^48 + z^47 + z^44 + \
49 #z^42 + z^39 + z^38 + z^35 + z^27 + z^25 + z^24 + z^22 + z^21 + z^18 + z^17 + z^15 + z^11 + \
50 #z^8 + z^7 + z^6 + z^3 + z^2 + 1
51 # ################################################################################
```

Archivo: `M233.nb` (*Multiplicación escalar sobre* $E(\mathbb{F}_{2^{233}})$)

```
1 # ##############################################################################
2 #m = 233;
3 #k = "100100111011001100010010110000000011101100011100001110010010100001110110110010011 1011 \
4 #1111101001001111111011000110000001001101010100110010010011011010011100101001101111111111111 \
5 #111111111111111111111111110011010001011101010110010011 11";
6 #lim = StringLength[k] + 1;
7 #
8 #a = 0;
9 #Gx = z^232 + z^230 + z^229 + z^228 + z^225 + z^221 + z^220 + z^217 + z^215 + z^213 + \
10 #z^212 + z^211 + z^209 + z^207 + z^202 + z^200 + z^197 + z^196 + z^195 + z^193 + z^190 \
11 #+ z^189 + z^188 + z^187 + z^186 + z^185 + z^182 + z^181 + z^180 + z^177 + z^176 + z^172 \
12 #+ z^171 + z^169 + z^167 + z^166 + z^165 + z^164 + z^160 + z^157 + z^155 + z^152 + z^151 \
13 #+ z^150 + z^149 + z^148 + z^145 + z^141 + z^139 + z^138 + z^137 + z^136 + z^135 + z^134 \
14 #+ z^133 + z^132 + z^130 + z^124 + z^122 + z^119 + z^116 + z^114 + z^112 + z^110 + z^109 \
15 #+ z^105 + z^104 + z^103 + z^101 + z^98 + z^92 + z^91 + z^88 + z^87 + z^86 + z^81 + z^78 \
16 #+ z^77 + z^75 + z^73 + z^72 + z^71 + z^70 + z^69 + z^68 + z^66 + z^64 + z^59 + z^57 + \
17 #z^54 + z^51 + z^50 + z^47 + z^44 + z^43 + z^42 + z^40 + z^38 + z^37 + z^35 + z^34 + z^33 \
18 #+ z^31 + z^30 + z^29 + z^27 + z^26 + z^25 + z^24 + z^23 + z^21 + z^19 + z^18 + z^16 + \
19 #z^14 + z^13 + z^8 + z^5 + z^2 + z;
20 #
21 #Gy = z^232 + z^231 + z^230 + z^228 + z^227 + z^225 + z^224 + z^222 + z^220 + z^217 + \
22 #z^216 + z^214 + z^213 + z^212 + z^211 + z^210 + z^208 + z^207 + z^206 + z^205 + z^203 + \
23 #z^202 + z^199 + z^198 + z^197 + z^195 + z^188 + z^187 + z^184 + z^183 + z^181 + z^180 + \
24 #z^178 + z^177 + z^176 + z^175 + z^174 + z^173 + z^172 + z^170 + z^169 + z^168 + z^163 + \
25 #z^162 + z^161 + z^160 + z^158 + z^156 + z^154 + z^152 + z^150 + z^148 + z^147 + z^145 + \
26 #z^142 + z^141 + z^138 + z^137 + z^136 + z^135 + z^134 + z^130 + z^125 + z^122 + z^121 + \
27 #z^120 + z^119 + z^117 + z^115 + z^111 + z^110 + z^107 + z^106 + z^104 + z^103 + z^100 + \
28 #z^99 + z^97 + z^96 + z^95 + z^94 + z^93 + z^92 + z^88 + z^87 + z^83 + z^81 + z^79 + z^78 \
29 #+ z^77 + z^75 + z^73 + z^72 + z^71 + z^68 + z^67 + z^65 + z^64 + z^62 + z^60 + z^58 + \
30 #z^57 + z^55 + z^54 + z^53 + z^47 + z^46 + z^40 + z^36 + z^30 + z^28 + z^26 + z^25 + z^23 \
31 #+ z^22 + z^21 + z^20 + z^19 + z^17 + z^15 + z^14 + z^13 + z^10 + z^9 + z^7 + z^5 + z + 1;
32 #
33 #Rx = Gx;
34 #Ry = Gy;
35 #PolinomioIrreducible = z^{233} + z^{74} + 1;
36 #
37 #For[i=2,i<lim, i++,
38 #       c=StringTake[k,{i}];
39 #       (*Doblado*)
40 #       {d,{inv,u}}=PolynomialMod[PolynomialExtendedGCD[Rx, PolinomioIrreducible],2];
41 #       Lamda=PolynomialMod[(Rx + Ry*inv),{PolinomioIrreducible,2}];
42 #       X3=PolynomialMod[(Lamda^2 + Lamda + a),{PolinomioIrreducible,2}];
43 #       Y3=PolynomialMod[(Rx^2 + Lamda*X3 + X3),{PolinomioIrreducible,2}];
44 #       Rx = X3;
45 #       Ry = Y3;
46 #       If[c=="1",{
47 #               (*Suma*)
48 #               {d,{inv2,u}}=PolynomialMod[PolynomialExtendedGCD[Gx + Rx, PolinomioIrreducible],2];
49 #               Lamda2=PolynomialMod[(Gy + Ry)*inv2,{PolinomioIrreducible,2}];
50 #               XX3=PolynomialMod[(Lamda2^2 + Lamda2 + Gx + Rx + a),{PolinomioIrreducible,2}];
51 #               YY3=PolynomialMod[Lamda2*(Gx + XX3) + XX3 + Gy,{PolinomioIrreducible,2}];
52 #               Rx = XX3;
53 #               Ry = YY3;
54 #               },{0}
55 #       ]
56 #]
57 #Print[Rx]
58 #Print[Ry]
59 #
60 #During evaluation of In[12]:= 1 + z + z^2 + z^3 + z^4 + z^6 + z^9 + z^10 + z^13 + z^15 + \
61 #z^16 + z^21 + z^24 + z^25 + z^27 + z^31 + z^33 + z^35 + z^36 + z^40 + z^44 + z^47 + z^49 \
62 #+ z^50 + z^54 + z^56 + z^58 + z^62 + z^63 + z^64 + z^65 + z^69 + z^70 + z^71 + z^74 + z^75 \
```

```
63#+ z^77 + z^79 + z^81 + z^83 + z^84 + z^85 + z^87 + z^88 + z^92 + z^93 + z^94 + z^95 + z^96 \
64#+ z^98 + z^99 + z^101 + z^104 + z^106 + z^107 + z^112 + z^113 + z^114 + z^116 + z^117 + \
65#z^121 + z^123 + z^125 + z^128 + z^130 + z^131 + z^134 + z^135 + z^138 + z^139 + z^144 + \
66#z^147 + z^149 + z^151 + z^152 + z^153 + z^156 + z^158 + z^159 + z^162 + z^163 + z^165 + \
67#z^167 + z^169 + z^170 + z^171 + z^172 + z^174 + z^175 + z^176 + z^177 + z^180 + z^181 + \
68#z^182 + z^184 + z^185 + z^188 + z^190 + z^191 + z^192 + z^193 + z^194 + z^195 + z^199 + \
69#z^200 + z^202 + z^203 + z^204 + z^205 + z^206 + z^208 + z^209 + z^210 + z^213 + z^215 + \
70#z^217 + z^218 + z^219 + z^220 + z^222 + z^223 + z^225 + z^227 + z^228 + z^231 + z^232
71#
72#During evaluation of In[12]:= 1 + z + z^3 + z^5 + z^8 + z^9 + z^10 + z^14 + z^16 + z^17 + \
73#z^18 + z^21 + z^24 + z^27 + z^29 + z^31 + z^33 + z^35 + z^36 + z^37 + z^38 + z^39 + z^41 + \
74#z^42 + z^44 + z^47 + z^50 + z^53 + z^57 + z^59 + z^60 + z^61 + z^63 + z^65 + z^66 + z^68 + \
75#z^69 + z^71 + z^75 + z^76 + z^77 + z^82 + z^85 + z^87 + z^91 + z^92 + z^93 + z^94 + z^95 + \
76#z^97 + z^99 + z^100 + z^101 + z^102 + z^105 + z^106 + z^108 + z^110 + z^113 + z^115 + z^116 \
77#+ z^117 + z^118 + z^120 + z^123 + z^124 + z^125 + z^130 + z^132 + z^133 + z^134 + z^136 + \
78#z^139 + z^141 + z^142 + z^143 + z^145 + z^147 + z^149 + z^150 + z^153 + z^156 + z^157 + z^158 \
79#+ z^159 + z^162 + z^163 + z^165 + z^167 + z^168 + z^170 + z^172 + z^173 + z^174 + z^175 + \
80#z^176 + z^177 + z^180 + z^182 + z^184 + z^185 + z^187 + z^190 + z^192 + z^193 + z^194 + z^197 \
81#+ z^198 + z^199 + z^200 + z^202 + z^203 + z^206 + z^208 + z^209 + z^213 + z^214 + z^215 + \
82#z^220 + z^222 + z^224 + z^226 + z^228 + z^229 + z^231 + z^232
83# ##########################################################################################
```

# Bibliografía

[1] M. Lucena. " Criptografía y Seguridad en Computadores". Tercera Edición, pp. 29 , 2003.

[2] R. Kammer. Federal Information Processing Standards Publication. *Data Encryption Standard (DES)*, FIPS PUB 46-3, 1999.

[3] V. Bravo, A. Araujo. Fundación Centro Nacional de Desarrollo e Investigación en Tecnologías Libres. *Criptografía*, CENDITEL, 2008.

[4] L. Ham, W. J. Hsin and M. Mehta *Authentificated Diffie-Hellman key agreement protocol using a single cryptographic assumption. Proceedings of the IEEE no. 20041041*, 2005.

[5] R. Villegas, Çomparativa de seguridad de algortimos de cifrado asimétrico", Tesis: Especialidad en seguridad informática y tecnologías de la información, Escuela Superior de Ingeniería Mecánica y Eléctrica del Instituto Politecnico Nacional, D.F., México, 2009

[6] I. Curry. *An Introduction to Crytography and Digital Signatures*, Version 2.0, 2001.

[7] *RSA Security Management*, [En línea]. Disponible: http://www.rsa.com/

[8] A. de J. Garcia y C. M. Penagos. Implementación Eficiente de Algoritmos Criptográficos Usando Curvas de Koblitz. 2008.

[9] A. Menezes, P. van Oorschot and S. Vanstone, Handbook of applied cryptography, CRC Press, 2007

[10] G. Belingueres *Introducción a los criptosistemas de curva elíptica.*

[11] U.S. Department of commerce/National Institute of Standards and Technology Federal Information Processing standards publication 2000 January 27.

[12] P. Longa, Criptografía: fundamentos, aplicaciones e implementación, University of Waterloo

[13] M. Hernandez, M. J. Vera, R. Mayol, Seminario de seguridad informática *Criptografía de curvas elípticas*, Universidad de los Andes-Mérida, pag. 9.

[14] D. Hankerson, A. Menezes and S. Vanstone, Guide to Elliptic Curve Cryptography, Springer-Verlag, pp. 79-82, 2003

[15] M. E. Briggs *An Introduction to the General Number Field Sieve*, Faculty of the Virginia Polytechnic Institute and State University, Blacksburg, Virginia. April 17, 1998.

[16] S. Bai, R. P. Brent *On the Efficiency of Pollard's Rho Method for Discrete Logarithms*, Department of Computer Science and Centre for Mathematics and its Applications, Australian National University, Canberra.

[17] J. S. Howell *The Index Calculus Algorithm for Discrete Logarithms*. 1998.

[18] J. Fan, D. V. Bailey, L. Batina, T. Guneysu, C. Paar and I. Verbauwhede. *Breaking elliptic curve cryptosystems using reconfigurable hardware*,

[19] *The GNU Multiple Precision Arithmetic Library*, [En línea]. Disponible: http://gmplib.org

[20] S. Hamdy. *LiDIA, A library for computational number theory*, 2004, [En línea]. Disponible: http://www.informatik.tu-darmstadt.de/TI/LiDIA

[21] *GNU Octave*, [En línea]. Disponible: http://www.gnu.org/software/octave

[22] Non-regulatory federal agency within the U.S. Department of Commerce, *National Institute of Standards and Technology*, [En línea]. Disponible: http://www.nist.gov/index.html

[23] H. Fan, J. Sun, M. Gu and K. Lam, Overlap-free Karatsuba-Ofman Polynomial Multiplication Algorithms, IET Information security, vol. 4, no. 1, pp. 8-14, 2010

[24] J. Cruz, "Multiplicación Escalar en Curvas de Koblitz: Arquitectura en Hardware Reconfigurable", Tesis: Maestro en Ciencias, Centro de Investigación y de Estudios Avanzados del Instituto Politecnico Nacional, D.F., México, 2005

[25] S. Hernández, "Multiplicación Escalar en Curvas Elípticas empleando Bisección de Punto: una Arquitectura en Hardware Reconfigurable", Tesis: Maestro en Ciencias, Centro de Investigación y de Estudios Avanzados del Instituto Politecnico Nacional, D.F., México, 2006