CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO

DEPARTAMENTO DE COMPUTACIÓN

Implementación multinúcleo de la multiplicación escalar en curvas de Koblitz

Tesis que presenta

ARMANDO FAZ HERNÁNDEZ

para obtener el grado de

Maestro en Ciencias en Computación

Directores de tesis:
Dr. Debrup Chakraborty
Dr. Francisco José Rambó Rodríguez Henríquez

México, Distrito Federal Febrero de 2012

Implementación multinúcleo de la multiplicación escalar en curvas de Koblitz

por

Armando Faz Hernández

CINVESTAV-IPN, México

Implementación multinúcleo de la multiplicación escalar en curvas de Koblitz

por

Armando Faz Hernández

Tesis presentada ante el profesorado del Departamento de Computación

del Centro de Investigación y de Estudios Avanzados

del Instituto Politécnico Nacional

cumpliendo satisfactoriamente con los

requisitos para obtener el grado de

Maestro en Ciencias en Computación

Aprobado por el comité examinador:

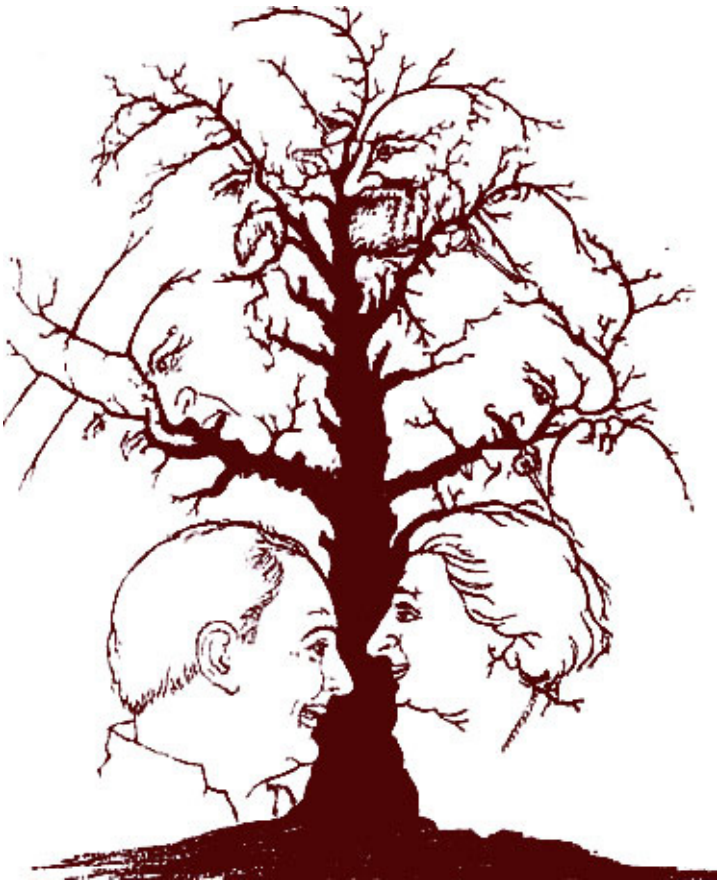
Dr. Debrup Chakraborty, Supervisor de Tesis

Dr. Francisco Rodríguez Henríquez, Supervisor de Tesis

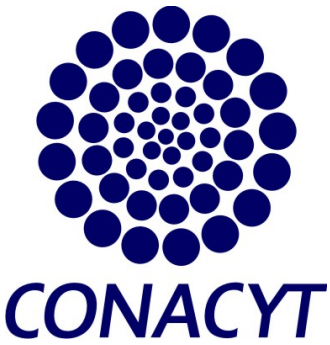
Dr. Luis Gerardo de la Fraga, Profesor Sinodal

Dr. Guillermo Benito Morales Luna, Profesor Sinodal

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional
Av. Instituto Politécnico Nacional No. 2508
Col. San Pedro Zacatenco. Distrito Federal, México



Porque gracias a su apoyo, he
realizado una de mis mejores metas.
Por todo esto, quiero que sientan que
el objetivo logrado, también es suyo.
*Con cariño para **Ma**, **Pu** y **Ri**.*



Con especial agradecimiento al Consejo Nacional de Ciencia y Tecnología por haber otorgado la beca de estudios de posgrado número 60240. Permitiendo impulsar día a día la ciencia en México.

Fue sin duda un honor ser parte del alumnado de esta gran institución, lider en investigación científica y tecnológica. Lugar donde la calidez humana y la calidad en sus aulas destacan a nivel internacional.

Al proyecto bilateral UCMexus por establecer un vínculo entre la Universidad de California en Santa Bárbara y el CINVESTAV en México, dando una aportación positiva en los estudios profesionales y de posgrado.

Resumen

La criptografía de curvas elípticas tiene alta transcendencia en las aplicaciones de seguridad informática, puesto que provee mecanismos para asegurar la privacidad de la información, la autenticación entre entidades que se comunican, así como también la integridad de los mensajes que se transmiten por un medio inseguro.

Actualmente existen algoritmos criptográficos que garantizan estos servicios de seguridad, sin embargo, algunos de ellos requieren una gran cantidad de procesamiento de cómputo. Tal es el caso de la multiplicación escalar, que es una operación fundamental para la implementación de la criptografía de curvas elípticas. Por consiguiente, es primordial que esta operación sea realizada de manera eficiente.

Esta tesis se ha enfocado en el análisis de los algoritmos y técnicas de programación que reduzcan el tiempo de cómputo de la multiplicación escalar.

Desde el punto de vista algorítmico, las curvas elípticas de Koblitz permiten que el cálculo de la multiplicación escalar pueda ser realizado rápidamente mediante la aplicación del endomorfismo de Frobenius, sin requerir el uso de doblados de puntos. La formulación de un algoritmo paralelo permitió su implementación en un procesador multinúcleo.

Los conjuntos extendidos de instrucciones presentes en las recientes arquitecturas de computadoras habilitan el procesamiento en paralelo de múltiples conjuntos de datos. Dentro de estos conjuntos, el uso del *multiplicador sin acarreo* mejora el rendimiento de las operaciones sobre campos finitos, resultando por ende, en la aceleración del cálculo de la multiplicación escalar.

Los resultados obtenidos de este trabajo de investigación ponen de manifiesto la aceleración obtenida en la paralelización de la multiplicación escalar, optimizando tanto algorítmicamente como con el uso de tecnologías recientes.

Abstract

Elliptic curve cryptography has a high significance on secure computer applications, it provides mechanisms to ensure privacy on data, authentication among communicating entities, as well as the integrity of a message sent by an insecure channel.

Nowadays, there are cryptographic algorithms that ensure these security services, however, some of them require a large amount of computer processing. Such is the case of the scalar multiplication, which is a fundamental operation for the implementation of elliptic curve cryptography. It is, therefore, essential that this operation be performed efficiently.

This thesis has focused on the analysis of algorithms and programming techniques to reduce the computation time of the scalar multiplication.

From the algorithmic standpoint, Koblitz elliptic curves allow that the computation of the scalar multiplication can be quickly performed by applying the Frobenius's endomorphism, without using point doublings. The formulation of a parallel algorithm allows its implementation in a multicore processor.

Extended instruction sets included in the latest computer architectures enable parallel processing of multiple data sets. Within these sets, the use of the *carry-less* multiplier enhances the performance of operations over finite fields, thereby resulting in acceleration of computation of scalar multiplication.

The results of this research show the speedup in the parallelization of the scalar multiplication, optimizing both algorithmically and with the use of recent technologies.

Prólogo

Interest in elliptic curve cryptography has been growing for a while now on, thanks to the development of many constructive protocols; it was discovered back in 1985 by Victor Miller and separately by Neil Koblitz.

Traditional cryptographic protocols, such as RSA, are well established and seen as "secure enough" for the immediate future, but have limited functionality. Public Key Cryptography scales better at higher security levels, however, it is still slower. Many protocols have been created, and even new sub-families of this type of cryptography.

Many efforts have been made to have better implementations of the mathematical backgrounds. There are several lines of research on this area: finding mathematical objects in the hope to exchange expensive operations for cheap or trivial ones, discovering mathematical facts to obey certain operations, analyze the implementations both in software and hardware to remove unneeded operations due to the physical architecture of the target development, and finally, make extensive use of (and propose) the new device features.

In this work, the author make extensive use of the new Intel Sandy Bridge architecture. This new processor, have several features that allow a fast arithmetic in the so-called binary fields, which are used extensively nowadays in industry to implement Elliptic Curve Cryptography.

Letting all of the innovation to the use of the new processor features would make this work dependent entirely on Intel, and the available infrastructure in the industry. The authors went further than that by exploiting the use of some mathematical object to reduce the complexity of the operations, and modified the algorithms available to create a parallel version of them.

These improvements gives a significant speed up in the scalar-point multiplication, the must used operation in elliptic curve cryptography, hence, its impact should not be seen just as speeding up an operation, both as a improvement that impacts the overall implementation

of many elliptic curve protocols.

This work not only presents these new improvements in a fashion manner, but also it can be used as an introductory material to the new Intel Sandy Bridge infrastructure, and to the world of parallel computing.

By *Luis J. Dominguez Perez, Ph.D.*
Posdoctoral Researcher
CINVESTAV-IPN
ldominguez@computacion.cs.cinvestav.mx

Lista de contenido

Lista de figuras

Lista de tablas

Lista de algoritmos

1

Introducción a la criptografía

Quienes son capaces de renunciar a la libertad esencial a cambio de una pequeña seguridad transitoria, no son merecedores ni de la libertad ni de la seguridad.

Benjamin Franklin

La criptografía es la encargada del diseño y el análisis de técnicas matemáticas que permiten establecer comunicaciones seguras ante la presencia de adversarios maliciosos.
Un sistema de comunicación básico posee dos entidades: A y B, (en la literatura criptográfica son representadas comúnmente con el nombre de Alicia y Beto, respectivamente), las cuales se comunican a través de un medio inseguro, ante la presencia de escuchas que son representados por el personaje de nombre Eva. Se desea que tanto Alicia y Beto se comuniquen sin que Eva pueda escuchar ni modificar el mensaje enviado y además, se preserve la seguridad en el intercambio de mensajes.

SECCIÓN 1.1

Sistemas criptográficos

La criptografía es un tema de suma relevencia en aplicaciones donde se desea preservar la seguridad de la información. Su estudio conlleva al uso de técnicas matemáticas y desarrollo de protocolos, que permiten manipular información sensible ante posibles entidades maliciosas.

El concepto de cifrado y ocultamiento de datos proviene desde la antigüedad, desde los primeros cifradores utilizados, como el caso del cifrador César que data del inicio de esta era, [50], hasta las técnicas más recientes donde la computadora es la herramienta vital para el desarrollo de nuevos esquemas.

El estudio de la criptografía se enfoca en desarrollar sistemas criptográficos, acordes al modelo básico de comunicación, donde dos entidades se desean comunicar a través de un canal de comunicación, el canal es comúnmente inseguro debido a la presencia de adversarios o escuchas, que pueden alterar los mensajes, infiltrar información o bien usurpar derechos de las entidades involucradas.

A continuación, se introducirá el concepto de sistemas criptográficos o criptosistemas, los cuales poseen operaciones y transformaciones matemáticas que son tema de estudio e investigación en el desarrollo de esta tesis.

Sistema criptográfico. Un criptosistema o sistema criptográfico, definido matemáticamente consiste de una tupla conformada por los siguientes elementos:

$$\{n, G, e, d, \mathcal{K}, \mathcal{M}, \mathcal{C}\} \tag{1.1}$$

El valor n es un parámetro (medido en bits) que indica el nivel de seguridad del sistema criptográfico. Un criptosistema se dice tener n bits de seguridad, si es posible recuperar una llave de $O = (n)$ bits de longitud, haciendo que el sistema sea vulnerable.

La función G es un algoritmo de generación de llaves, el cual toma como entrada un valor n, dando como resultado llaves $k \in \mathcal{K}$ de longitud $O(n)$.

A la función e se le conoce como la función de cifrado y es del tipo:

$$e: \quad \mathcal{K} \times \mathcal{M} \to \mathcal{C} \tag{1.2}$$

donde $\mathcal{K}$ es el conjunto de todas las posibles llaves, $\mathcal{M}$ es el conjunto de los mensajes válidos y $\mathcal{C}$ es el espacio de las cifras.

La función d es la función de descifrado, la cual es del tipo:

$$d: \quad \mathcal{K} \times \mathcal{C} \to \mathcal{M} \tag{1.3}$$

Para asegurar la funcionalidad del sistema criptográfico, es necesario que para cada llave, $k \leftarrow G(n)$, se cumpla la siguiente ecuación:

$$d(k, e(k, m)) = m, \quad \forall m \in \mathcal{M} \tag{1.4}$$

es decir, al momento de descifrar un texto cifrado es posible recuperar el mensaje original, si se posee la llave adecuada generada por G.

1.1.1. Clasificación

Los sistemas criptográficos modernos pueden ser clasificados en dos grupos principales:

- Los sistemas criptográficos simétricos.

  Estos esquemas se encuentran bajo la suposición de que ambas entidades comparten la llave generada por G, es decir, como su nombre lo indica, en este tipo de criptosistemas se utiliza una llave de carácter privado para cifrar y descifrar la información, de esta manera si dos entidades desean intercambiar mensajes de forma segura, ambas deben establecer una clave común y secreta.

  La criptografía de llave privada se utiliza para cifrar grandes cantidades de información a gran velocidad debido a la simpleza de las operaciones que involucra.

  Sin embargo, una de las desventajas de este sistema consiste en que para cada pareja de entidades a comunicarse, deberá existir una llave privada única, lo cual produce que el número de llaves necesarias aumente de forma cuadrática con respecto al número de entidades.

- Los sistemas criptográficos asimétricos o de llave pública.

  En este esquema el algoritmo de generación de llaves produce dos llaves:

  $$(k_{\text{pública}}, k_{\text{privada}}) \leftarrow G(n) \tag{1.5}$$

  para cada entidad involucrada en la comunicación. La llave pública, denotada por $k_{\text{pública}}$, se distribuye a todas las entidades para cifrar mensajes al destinatario. Mientras que la llave privada, k_{privada}, permanece en secreto por cada entidad, la cual es utilizada para descifrar los textos cifrados.

  Las funciones de cifrado y descifrado están relacionadas bajo la siguiente ecuación:

  $$d(k_{\text{privada}}, e(k_{\text{pública}}, m)) = m, \quad \forall m \in \mathcal{M} \tag{1.6}$$

Como se puede apreciar, la distribución de llaves viene a ser ventajosa, dado que cada entidad tiene acceso a las llaves públicas de sus pares, mientras que cada entidad guarda celosamente su llave privada. El número de llaves necesarias para un entorno de comunicación es linealmente proporcional al número de entidades involucradas.

En la mayoría de los criptosistemas de llave pública, la seguridad que ofrecen subyace en algún problema matemático difícil o bien computacionalmente intratable. Es prioritario que sea computacionalmente irrealizable el hecho de que dada la llave pública se obtenga la llave privada.

Las primeras nociones sobre criptografía de llave pública fueron presentadas por Diffie y Hellman en [23] y por Merkle en [62]. En el primer artículo se presenta el algoritmo para el establecimiento de llaves utilizando la exponenciación en un grupo multiplicativo de enteros modulo un número primo, $(\mathbb{Z}_p^*)$; así como también describen el cifrado y la firma digital utilizando funciones de sólo ida.

El esquema de cifrado RSA [69], obtiene su seguridad de la intratabilidad del problema de la factorización de números enteros, el cual dado un número entero compuesto, se ha de encontrar los factores primos que lo componen.

1.1.2. Servicios de seguridad

La criptografía no sólo concierne en cifrar y descifrar mensajes, sino que también está enfocada a resolver problemas relacionados con la seguridad de la información. Los principales objetivos de la criptografía son llamados servicios de seguridad, los cuales se describen a continuación:

- **Confidencialidad**. Avalar que todos los datos se mantengan secretos, excepto de aquellas personas que estén autorizadas para verlos.
- **Integridad de datos**. Asegurar que los datos no hayan sido alterados por una entidad no autorizada, lo cual significa que el destinatario de la información sea capaz de detectar si hubo anomalías en la transmisión.
- **Autenticación del origen de datos**. Corroborar que el mensaje proviene del emisor correcto y que no se trata de un impostor enviando un mensaje.
- **Autenticación de las entidades**. Verificar que la persona con la que se estableció la comunicación sea quien dice ser.
- **No repudio**. Impedir la negación de cualquier acto implicado en la comunicación por cualquiera de las entidades participantes.

SECCIÓN 1.2

Especificación del problema

La criptografía de curvas elípticas es un esquema de llave pública que provee los servicios de seguridad antes mencionados. La criptografía de curvas elípticas fue propuesta por Koblitz [52] y Miller [63] independientemente en 1985, demostraron que los puntos pertenecientes a una curva elíptica sobre un campo finito forman un grupo aditivo, con el cual es posible construir sistemas criptográficos respaldados bajo la complejidad del problema del logaritmo discreto.

Se han desarrollado un sin número de protocolos criptográficos con el uso de las curvas elípticas, siendo actualmente, la criptografía de curvas elípticas uno de los temas más investigados. La principal ventaja de este sistema si se compara con el criptosistema RSA [69], radica en que el nivel de seguridad que provee puede ser alcanzado con llaves de menor tamaño que las llaves de RSA.

La ley de grupo que forman los puntos de la curva elíptica es expresada como la suma de puntos. Dados dos puntos, P y Q, en la curva se puede obtener un tercer punto $R = P + Q$ perteneciente también a la curva. En el caso especial cuando se desean sumar dos puntos iguales se le conoce como doblado de puntos, la notación para describir un doblado se expresa como $Q = 2P = P + P$.

Si aplicamos k veces el operador de grupo a un punto P consigo mismo, obtendremos un punto Q, esta operación se le conoce como multiplicación escalar y se denota como $Q = kP$. Utilizando la suma y el doblado de puntos es posible calcular la multiplicación escalar de un punto mediante el algoritmo conocido como de "doblado y suma". Esta operación es de uso primordial en el desarrollo de algoritmos criptográficos y por ende se pretende que su ejecución sea rápida y de la manera más eficiente posible.

En 1991, Neal Koblitz en [54] introdujo una familia de curvas que admiten un endomorfismo (τ), el cual es utilizado en el algoritmo de multipicación escalar, permitiendo reemplazar doblados de puntos por aplicaciones de este operador.

En las curvas de Koblitz, el operador de Frobenius es una operación bastante ágil comparada con el doblado de puntos, lo cual permite acelerar notablemente el cómputo de la multiplicación escalar.

En [73; 74], Solinas presenta versiones mejoradas de los algoritmos existentes, para el cómputo de la multiplicación escalar en las curvas de Koblitz. El algoritmo consiste en expresar al escalar k en base $\mathtt{t}$, donde $\mathtt{t}$ es una raíz compleja de la ecuación característica del endomorfismo de Frobenius (se detallará en la sección 4.3.2 de la página 56). Luego mediante una expansión de factores conocida como $\omega\tau$-NAF realizar un algoritmo de τ-y-suma, análogo al de doblado y suma.

Teóricamente, los resultados que Solinas presenta son atractivos para su implementación eficiente, la interrogante evidente resulta en determinar cómo aplicar los recursos existentes en las

arquitecturas recientes para calcular la multiplicación escalar en curvas de Koblitz, y obtener una aceleración significativa en el tiempo de cómputo de esta operación.

Cabe notar que los procesadores actuales poseen conjuntos extendidos de instrucciones, que aceleran programas mediante paralelismo a diferentes niveles, siendo el más benéfico: el paralelismo a nivel de tareas.

Una herramienta vital para el desarrollo de esta tesis es la inclusión en las últimas arquitecturas de procesadores el nuevo multiplicador sin acarreo, lo cual conlleva a un incremento en la velocidad de ejecución de la aritmética del campo finito $\mathbb{F}_{2^m}$.

SECCIÓN 1.3

Solución planteada

El trabajo aquí expuesto acelera el cómputo de la multiplicación escalar en las curvas de Koblitz. Para realizar una implementación rápida, se debe buscar e investigar cuáles son los mejores algoritmos que permitan la ejecución eficiente de la aritmética de campo, la aritmética de curvas elípticas y finalmente el desarrollo de la multiplicación escalar.

Es importante aprovechar las ventajas existentes en las arquitecturas de computadoras más recientes y explotar los recursos a su máximo potencial.

Uno de los principales recursos existentes es el conjunto de instrucciones SIMD (del inglés, *Single Instruction Multiple Data*), las cuales procesan al mismo tiempo una instrucción sobre múltiples datos, experimentándose un paralelismo a nivel de datos. Las nuevas arquitecturas proveen el conjunto de instrucciones SSE (del inglés, *Streaming SIMD Instructions*), las cuales cuentan con registros de 128 bits y 256 bits haciendo que la programación se realice de forma vectorial. Estas instrucciones fueron inicialmente usadas en aplicaciones para procesamiento de imágenes y codificación de video. Nuestra propuesta explota su uso en la criptografía de curvas elípticas.

En particular, el conjunto de instrucciones `AES-NI` posee una instrucción de ensamblador que realiza el producto de dos números enteros sin producir acarreo en la suma. Esta multiplicación, conocida como multiplicación sin acarreo, es sumamente rápida en contraste con implementaciones anteriores que utilizan registros de 32 bits.

En [6], los autores proponen el cómputo de la multiplicación escalar de forma paralela, aplicando el endomorfismo de Frobenius para expresar el escalar k en función del operador τ y τ^{-1}. De esta forma es posible realizar el cómputo de la multiplicación en dos hilos de ejecución independientes, combinando al final los resultados parciales por cada hilo.

La paralelización de los algoritmos que calculan la multiplicación escalar muestra una oportunidad para acelerar este procedimiento, ya que los procesadores actuales poseen dos o más núcleos

Figura 1.1: Diagrama de capas para la implementación de un criptosistema basado en curvas elípticas.

en el mismo circuito integrado. Haciendo posible que los programas se ejecuten en forma totalmente paralela en el procesador. En el trabajo de tesis presentado se analizan diversas técnicas de paralelización sobre la multiplicación escalar.

Metodología. Para implementar la multiplicación escalar en curvas elípticas de Koblitz, se debe seguir un modelo de capas, en el cual las operaciones de un nivel son construídas mediante las operaciones realizadas en el nivel inferior. Siguiendo esta metodología y observando la figura 1.1, se puede apreciar que para la implementación de los protocolos criptográficos, es necesario desarrollar las operaciones de los niveles inferiores. En particular este trabajo se centra en el nivel donde la multiplicación escalar está presente y los niveles inferiores que la componen.

Las operaciones del nivel inferior, es decir, las operaciones del campo finito $\mathbb{F}_{2^m}$, tomarán ventaja de las instrucciones SIMD para su implementación eficiente.

En el caso de la multiplicación escalar, se pretende acelerar su tiempo de ejecución con el uso de dos núcleos, para de esta forma obtener factores de aceleración significativos.

Sección 1.4
Mapa de la tesis

En el capítulo 2 se presenta una descripción matemática de las estructuras algebraicas utilizadas a lo largo de este trabajo de investigación. Conceptos tales como grupo, anillo, campo, campo finito y campo finito binario son examinados en profundidad.

El capítulo 3 expone la teoría involucrada en la programación en paralelo, se distinguen tres niveles de paralelismo: a nivel de instrucciones, a nivel de datos y a nivel de tareas. También se

detalla el uso de instrucciones SIMD y las arquitecturas multinúcleo.

En el capítulo 4 se presenta formalmente la definición de las curvas elípticas y el grupo formado por los puntos de las curvas. Se expone la ley de grupo, las diferentes representaciones de puntos, así como el concepto de multiplicación escalar. Además, se introducen las curvas elípticas de Koblitz, las cuales presentan vastas propiedades para realizar la multiplicación escalar de forma más rápida al reemplazar los doblados de punto por aplicaciones del endomorfismo de Frobenius.

En el capítulo 5 se detallan los algoritmos utilizados para la implementación de la aritmética del campo finito $\mathbb{F}_{2^m}$. Se expone la formulación algorítmica del particionamiento de la multiplicación escalar en dos núcleos de procesamiento. Finalmente, se muestra el ambiente de programación y los resultados obtenidos por la implementación realizada en esta tesis, así como también, se hace una comparativa con trabajos de otros autores para evaluar las mejoras aquí logradas.

En el último capítulo se presentan las conclusiones y los posibles trabajos a futuro que han resultado de esta investigación.

En el apéndice A se explican las herramientas de programación utilizadas para el desarrollo de la tesis. Se profundiza el uso de funciones *Intrinsics* para el manejo de los conjuntos extendidos de instrucciones.

En el apéndice B se da una breve descripción de los compiladores utilizados, se detallan sus características y se resaltan sus principales diferencias. La herramienta OpenMP es utilizada para el desarrollo de programas en paralelo, en dicho apéndice se presentan sus caraterísticas y el modo de funcionamiento de esta biblioteca.

En el apéndice C se enlistan las publicaciones realizadas que han surgido como parte del presente trabajo de investigación.

2

Fondo matemático

En cuanto alguien comprende que obedecer leyes injustas es contrario a su dignidad de hombre, ninguna tiranía puede dominarle.

Mahatma Gandhi

En este capítulo se examinan conceptos matemáticos básicos para el desarrollo de esta tesis. Inicialmente se da una introducción a las secuencias de Lucas, las cuales están relacionadas con los parámetros de algunos algoritmos referentes a las curvas elípticas de Koblitz.

Se detallan también algunas estructuras algebraicas básicas, como el caso de grupos, anillos y campos. La teoría de los grupos cíclicos es relevante en la criptografía de curvas elípticas, debido al problema del logaritmo discreto.

Los campos, en especial los campos finitos, son la base sobre la cual están definidas las curvas elípticas. Es por tanto, importante seleccionar campos finitos donde la aritmética pueda ser eficientemente computada.

SECCIÓN 2.1

Secuencias de Lucas

Una secuencia o sucesión entera es una lista ordenada de números enteros. Una secuencia puede especificarse explícitamente a través de una fórmula para sus n-ésimos términos, o bien implícitamente estableciendo una relación entre sus elementos. A continuación se expondrá la secuencia de Lucas que será utilizada durante el desarrollo de este trabajo.

Secuencia de Lucas. Sean P y Q dos enteros y $D = P^2 - 4Q$. Entonces las raíces de

$$x^2 - Px + Q = 0, \tag{2.1}$$

son:

$$a = \frac{1}{2}(P + \sqrt{D}), \tag{2.2}$$
$$b = \frac{1}{2}(P - \sqrt{D}). \tag{2.3}$$

Ahora definimos para $n \geq 0$:

$$U_n(P,Q) = \frac{a^n - b^n}{a - b}, \tag{2.4}$$
$$V_n(P,Q) = a^n + b^n. \tag{2.5}$$

Las secuencias de Lucas están dadas por las siguientes relaciones de recurrencia:

$$U_n(P,Q) = PU_{n-1} - QU_{n-2}, \quad U_1 = 1, \quad U_0 = 0, \tag{2.6}$$
$$V_n(P,Q) = PV_{n-1} - QV_{n-2}, \quad V_1 = P, \quad V_0 = 2. \tag{2.7}$$

La tabla 2.1 muestra algunos casos específicos las secuencias de Lucas siendo la secuencia de Fibonnacci un caso particular.

Para el desarrollo de este trabajo haremos uso de las secuencias de Lucas parametrizadas como sigue:

$$U_n = U_n(\mu, 2) = \mu U_{n-1} - 2U_{n-2}, \quad U_1 = 1, \quad U_0 = 0, \tag{2.8}$$
$$V_n = V_n(\mu, 2) = \mu V_{n-1} - 2V_{n-2}, \quad V_1 = \mu, \quad V_0 = 2, \tag{2.9}$$

donde μ es una constante entera. Vastas son las propiedades de las secuencias de Lucas, algunas de ellas se describen a detalle en [73].

(P,Q)	U_n	V_n
$(1,-1)$	Números de Fibonnacci	Números de Lucas
$(2,-1)$	Números de Pell	Números de Pell-Lucas
$(1,-2)$	Números de Jacobsthal	Números de Jacobsthal-Lucas

Tabla 2.1: Casos especiales de las secuencias de Lucas.

SECCIÓN 2.2

Grupos

Un grupo es una estructura algebraica consistente de un conjunto S y de una operación binaria $\star$, denotado como $\langle S, \star \rangle$, el cual satisface las siguientes condiciones:

- **Cerradura.** Para $a, b \in S$ se cumple que $a \star b \in S$. Por lo tanto, la operación $\star$ se define como:

$$\begin{aligned} \star\colon S \times S &\rightarrow S \\ (a,b) &\mapsto a \star b \end{aligned} \tag{2.10}$$

- **Asociatividad.** Para cualquier terceta de elementos $a, b, c \in S$ se cumple que:

$$(a \star b) \star c = a \star (b \star c) \tag{2.11}$$

- **Existencia de un elemento identidad.** Existe un elemento $e \in S$ tal que:

$$e \star a = a \star e = a \tag{2.12}$$

  para todo $a \in S$.

- **Existencia de inversos.** Para cada $a \in S$ existe un elemento $a' \in S$ tal que:

$$a \star a' = e = a' \star a \tag{2.13}$$

Se le conoce como grupo aditivo cuando representamos a la operación de grupo con $+$, a la identidad del grupo con 0 y a los inversos como $-a$; mientras que le llamamos grupo multiplicativo si la operación de grupo se representa con $\cdot$, la identidad del grupo con 1 y a los inversos mediante a^{-1}.

Un grupo se llama conmutativo o abeliano si cumple la propiedad de conmutatividad, es decir, para todos $a, b \in S$:

$$a \star b = b \star a \tag{2.14}$$

La notación común al hablar de grupos consiste en escribir como $\mathbb{G}$ al grupo $\langle G, \star \rangle$.

2.2.1. Orden del grupo

El orden o cardinalidad del grupo se define como el número de elementos en el conjunto G, se denota por $\#\mathbb{G}$. Si la cardinalidad del conjunto G es finita, se dice que el grupo es finito, de otra forma decimos que el grupo tiene orden infinito.

El orden de un elemento $a \in \mathbb{G}$ es el entero positivo m más pequeño tal que: $a^m = e$ y se denota como $m = \text{ord}(a)$. Si m no existe, se dice que el orden es infinito. La notación a^m representa la aplicación del operador de grupo sobre el elemento a consigo mismo m veces.

$$a^m = \underbrace{a \star a \star a \star \ldots a}_{m \text{ veces}}$$

2.2.2. Subgrupos

Sea $\mathbb{G}$ un grupo, y sea H un subconjunto no vacío de $\mathbb{G}$, tal que:

- $a \star b \in H$ para todo $a, b \in H$,
- $a^{-1} \in H$ para todo $a \in H$,

entonces $\mathbb{H}$ forma un subgrupo de $\mathbb{G}$. Usualmente se escribe $\mathbb{H} \subset \mathbb{G}$. Si $\mathbb{G}$ es un grupo entonces $\{e\} \subset \mathbb{G}$ donde e es el elemento identidad del grupo.

Sea $a \in \mathbb{G}$, el conjunto $\{a^n : n \in \mathbb{Z}\}$ es un subgrupo de $\mathbb{G}$ generado por a, este subgrupo se denota como $\langle a \rangle$.

Si existe un elemento del grupo, $g \in \mathbb{G}$, cuyo orden sea igual al orden del grupo, $\text{ord}(g) = \#\mathbb{G}$, a este elemento se le conoce como elemento generador del grupo, el cual se denota como $\langle g \rangle = \mathbb{G}$. Un grupo es cíclico si existe un $g \in \mathbb{G}$ tal que $\langle g \rangle = \mathbb{G}$.

Teorema de Lagrange. Sea $\mathbb{G}$ un grupo finito y $\mathbb{H}$ un subgrupo de $\mathbb{G}$. Entonces el orden de $\mathbb{H}$ divide al orden de $\mathbb{G}$.

Corolario. Si $\mathbb{G}$ es un grupo cíclico de orden n, entonces por cada divisor d de n, $\mathbb{G}$ contiene exactamente un subgrupo cíclico de orden d.

2.2.3. Homomorfismo de grupos

Sean $\langle G, \star \rangle$ y $\langle H, \bullet \rangle$ dos grupos, un homomorfismo es una función $h : \mathbb{G} \to \mathbb{H}$ tal que para todos $x, y \in \mathbb{G}$ se cumple:

$$h(x \star y) = h(x) \bullet h(y). \tag{2.15}$$

Se define el núcleo o kernel de h como el conjunto:

$$\ker(h) = \{g \in \mathbb{G} \colon h(g) = e_{\mathbb{H}}\} \tag{2.16}$$

donde $e_{\mathbb{H}}$ es el elemento identidad del grupo $\mathbb{H}$. El kernel de h forma un subgrupo de $\mathbb{G}$.

Un isomorfismo de grupos se presenta cuando la función h es una biyección, mientras que un endomorfismo de grupos la función h es de la forma $h \colon \mathbb{G} \to \mathbb{G}$. Un automorfismo de grupos es a la vez un endomorfismo y un isomorfismo.

2.2.4. Logaritmo discreto

Sea $\mathbb{G}$ un grupo cíclico de n elementos. Sea $\langle g \rangle = \mathbb{G}$, por lo tanto cualquier elemento $a \in \mathbb{G}$ se puede escribir en términos de g, de la forma $a = g^k$ para algún $k \in \mathbb{Z}$. Definimos la función:

$$\log_g \colon \mathbb{G} \to \mathbb{Z}_n, \tag{2.17}$$

donde $\mathbb{Z}_n$ denota el conjunto de números enteros modulo n, con $n = \#\mathbb{G}$. La función $\log_g$ asigna a cada elemento $a \in \mathbb{G}$ el entero $k \in \mathbb{Z}_n$ tal que $a = g^k$. La función $\log_g$ representa un isomorfismo de grupos conocida como logaritmo discreto en base g.

Problema del logaritmo discreto. El problema del logaritmo discreto (PLD) consiste en resolver la siguiente instancia: dados $\mathbb{G}$ un grupo finito, un generador $g \in \mathbb{G}$ de orden n y un elemento $h \in \mathbb{G}$, se desea encontrar la solución $i = \log_g(h)$ con $i \in \mathbb{Z}_n$.

Un algoritmo ingenuo para resolver PLD consiste en realizar una búsqueda exhaustiva, la complejidad computacional en tiempo de este algoritmo es $O(2^k)$, donde $k = \log_2(m)$, siendo m el orden del grupo. En este caso, la búsqueda de la solución al problema del logaritmo discreto posee complejidad exponencial al tamaño del grupo.

Existen algoritmos sofisticados que resuelven el problema del logaritmo discreto en diferentes grupos. A continuación se listan algunos de ellos, la mayoría de ellos lo resuelven más rápido que utilizando el algoritmo ingenuo, sin embargo, ninguno de ellos lo hace con complejidad polinomial al tamaño del orden del grupo.

- Algoritmo de Shanks, conocido como “paso de infante, paso de gigante” [70]
- Algoritmo Pollard-Rho [67]
- Algoritmo Pohlig-Hellman [66]
- Algoritmo del cálculo del índice [41]
- Criba de Adleman [4]

SECCIÓN 2.3

Anillos

Un anillo es una estrucutra algebraica formada por un conjunto R junto con dos operaciones binarias $\star$ y $\bullet$, denotado como $\langle A, \star, \bullet \rangle$, tal que cumple con las siguientes condiciones:

- $\langle A, \star \rangle$ es un grupo abeliano.
- $\bullet$ tiene la propiedad asociativa.
- 0 es el elemento identidad de $\langle A, \star \rangle$ y 1 es el elemento identidad de $\langle A, \bullet \rangle$.
- $\bullet$ es una operación distributiva sobre $\star$, es decir, para todo $x, y, z \in A$ se cumple:

$$x \bullet (y \star z) = (x \bullet y) \star (x \bullet z) \tag{2.18}$$
$$(y \star z) \bullet x = (y \bullet x) \star (z \bullet x) \tag{2.19}$$

El anillo $\mathbb{A}$ se le llama anillo conmutativo si el operador $\bullet$ es conmutativo. Si para todo elemento en el anillo, $a \in \mathbb{A} \setminus \{0\}$, existe un $a' \in \mathbb{A}$ tal que $a' \bullet a = a \bullet a' = 1$, entonces el anillo se le llama anillo de división. Un dominio integral es un anillo conmutativo tal que para cualquiera dos elementos $a, b \in \mathbb{A}$, si $a \bullet b = 0$ entonces $a = 0$, ó bien $b = 0$.

Un dominio euclidiano es un par $(\mathbb{A}, \mathcal{N})$, donde $\mathbb{A}$ es un dominio integral y $\mathcal{N}$ es una función de norma euclidiana definida como:

$$\mathcal{N} \colon \mathbb{A} \setminus \{0\} \to \mathbb{N} \cup \{0\} \tag{2.20}$$

tal que se satisface la siguiente condición: si $a, b \in \mathbb{A}$ y $b \neq 0$, entonces existen elementos $q, r \in \mathbb{A}$ tal que $a = qb + r$, con $r = 0$ ó $\mathcal{N}(r) < \mathcal{N}(b)$.

SECCIÓN 2.4

Campos

Un campo es una estructura algebraica compuesta por un conjunto F de elementos y dos operaciones binarias $\star$ y $\bullet$, dichas operaciones forman un campo cuando se cumplen las siguientes propiedades:

- **Cerradura.** Para todo elemento $a, b \in F$, se cumple que $a \star b \in F$ y que $a \bullet b \in F$.

- **Asociatividad.** Para toda tercia $a,b,c \in F$, se cumplen las siguientes ecuaciones:

$$\begin{aligned} a\star(b\star c) &= (a\star b)\star c \\ a\bullet(b\bullet c) &= (a\bullet b)\bullet c \end{aligned}$$

- **Conmutatividad.** Para todo elemento $a,b \in F$, se cumplen las siguientes ecuaciones:

$$\begin{aligned} a\star b &= b\star a \\ a\bullet b &= b\bullet a \end{aligned}$$

- **Existencia de elemento identidad.** Para todo $a \in F$, existe un elemento llamado identidad aditiva, denotado como 0, tal que $a\star 0 = a$, así como también, existe un elemento llamado identidad multiplicativa, denotado como 1, tal que $a\bullet 1 = a$.

- **Existencia de inversos.** Para cada elemento $a \in F$, existe un elemento $a' \in F$ tal que $a\star a' = 0$. Mientras que para cada elemento $a \in F \setminus \{0\}$ existe un elemento a^{-1} tal que $a\bullet a^{-1} = 1$.

- **Distributividad** El operador $\bullet$ se debe distribuir sobre el operador $\star$, es decir, para todo $a,b,c \in F$ se cumple la siguiente ecuación:

$$a\bullet(b\star c) = (a\bullet b)\star(a\bullet c)$$

Alternativamente un campo puede ser definido como dos grupos abelianos $\langle F,0,\star\rangle$ y $\langle F\setminus\{0\},1,\bullet\rangle$, cuya operación $\bullet$ es distributiva sobre $\star$.

2.4.1. Campos finitos

El orden del campo, indica el número de elementos pertenecientes al campo. Si el orden es finito, se dice que el campo es finito, o también llamado campo de Galois (GF).

Para todo número p primo y $k \in \mathbb{Z}^+$ existe exactamente un campo finito de orden p^k, donde p es la característica del campo. Comúnmente denotado como $\mathbb{F}_{p^k}$ o también mediante GF(p^k).

Dos campos son isomórfos si estructuralmente son iguales salvo la representación de sus elementos, la cual puede ser diferente.

El campo no trivial más pequeño, es $\langle \mathbb{F}_2, \oplus, \odot\rangle$, el cual posee dos elementos $\{0,1\}$ y las operaciones están descritas en la figura 2.1.

$\oplus$	0	1
0	0	1
1	1	0

$\odot$	0	1
0	0	0
1	0	1

Figura 2.1: Operadores del campo $\mathbb{F}_2$

2.4.2. Extensión de campo

Sea $\mathbb{F}$ un subcampo de $\mathbb{E}$, es decir, $\mathbb{F}$ es un subconjunto de $\mathbb{E}$ y es por sí mismo un campo. En este caso a $\mathbb{E}$ se le conoce como una extensión del campo $\mathbb{F}$.

Si $\mathbb{F}_q$ es un campo finito de orden $q = p^k$, entonces $\mathbb{Z}_p$ es un subcampo de $\mathbb{F}_q$. Por lo tanto, $\mathbb{F}_q$ puede ser visto como una extensión del campo $\mathbb{Z}_p$.

Polinomios irreducibles. Sea $p(x) \in \mathbb{F}[x]$ un polinomio de grado al menos 1, se dice que $p(x)$ es irreducible sobre $\mathbb{F}$ si no puede ser escrito como el producto de dos polinomios no triviales en $\mathbb{F}[x]$.

La notación $\mathbb{F}[x]/(p(x))$ denota al conjunto de las clases de equivalencia de los polinomios en $\mathbb{F}[x]$ módulo $p(x)$, donde $p(x)$ es un polinomio irreducible de grado k. Cualquier elemento de la extensión del campo $\mathbb{F}_{p^k}$ puede ser representado como un polinomio de grado $k-1$, con coeficientes en $\mathbb{F}_p$. Bajo esta representación se tiene que $\mathbb{F}_{p^k}$ es isomórfo a $\mathbb{F}[x]/(p(x))$.

2.4.3. Automorfismo de Frobenius

Sea $\langle \mathbb{F}_q, +, \bullet \rangle$ un campo finito de orden $q = p^k$. Se define la función ϕ_q como:

$$\begin{array}{rccc} \phi_q: & \mathbb{F}_q & \rightarrow & \mathbb{F}_q \\ & a & \mapsto & a^q \end{array} \tag{2.21}$$

a esta función se le conoce como el automorfismo de Frobenius. Este automorfismo es una operación lineal en el campo $\mathbb{F}_q$, de forma que para todos $x, y \in \mathbb{F}_q$ se cumplen las siguientes ecuaciones:

$$\phi_q(x+y) = \phi_q(x) + \phi_q(y) \tag{2.22}$$

$$\phi_q(x \bullet y) = \phi_q(x) \bullet \phi_q(y) \tag{2.23}$$

Para desmostrar la identidad anterior, se aplica el Teorema del Binomio:

$$(x+y)^q = \sum_{i=0}^{p} \binom{p}{i} x^{p-i} y^i = x^p + \sum_{i=1}^{p-1} \binom{p}{i} x^{p-i} y^i + y^p \tag{2.24}$$

al ser un campo de característica p, todos los coeficientes $\binom{p}{i} \equiv 0 \pmod{p}$ para $0 < i < p$, entonces:

$$(x+y)^p = x^p + y^p \tag{2.25}$$

para finalizar se demuestra por inducción sobre el exponente k de p^k. La demostración completa de estas identidades se encuentra en el apéndice C de [79].

2.4.4. Aritmética del campo $\mathbb{F}_{2^m}$

Para los fines del presente trabajo de investigación centraremos el estudio en las extensiones de campos de característica 2, también llamados campos finitos binarios.

Es preciso describir cómo se realiza la aritmética en el campo $\mathbb{F}_{2^m}$, por lo que a continuación se presentan los algoritmos principales para el desarrollo de algunas operaciones básicas tales como la suma, la multiplicación, el cálculo de inversos y otras más.

Suma

Para sumar dos elementos $a(x), b(x) \in \mathbb{F}_2[x]$, se procede a realizar la suma polinomial, utilizando el algoritmo 1 se puede encontrar la suma de dos elementos, donde el operador $\oplus$ representa la adición del campo $\mathbb{F}_2$ descrita en la figura 2.1.

Algoritmo 1 Suma en $\mathbb{F}_{2^m}$.

Entrada: $a(x), b(x) \in \mathbb{F}_{2^m}$

Salida: $c(x) \in \mathbb{F}_{2^m}$ tal que $c(x) = a(x) + b(x)$

1: **for** $i = 0$ **to** $m-1$ **do**
2: $\quad c_i x^i \leftarrow (a_i \oplus b_i) x^i$
3: **end for**
4: **return** $c(x)$

Multiplicación

La multiplicación de dos elementos en el campo se lleva a cabo mediante dos fases:

- La multiplicación polinomial.

  El algoritmo de Karatsuba presentado en [51], es una técnica para realizar una multiplicación polinomial de forma eficiente. El algoritmo posee complejidad subcuadrática, (exactamente $O(n^{\ln 3})$), a diferencia del método "tradicional de escuela" cuya complejidad es $O(n^2)$.

Sean $a(x), b(x) \in \mathbb{F}_2[x]$, si ambos polinomios son de grado menor a m, éstos se pueden multiplicar dividiendo a cada operando en dos polinomios de grado $l = \lceil \frac{m}{2} \rceil$, denotamos a estos polinomios mediante A_0, A_1, B_0 y B_1:

$$\begin{aligned} a(x) &= A_1 x^l + A_0 \\ b(x) &= B_1 x^l + B_0 \end{aligned}$$

para después realizar 3 multiplicaciones de polinomios de la siguiente forma:

$$\begin{aligned} c(x) &= a(x)b(x) \\ &= (A_1 x^l + A_0)(B_1 x^l + B_0) \\ &= A_1 B_1 x^{2l} + [(A_1 + A_0)(B_1 + B_0) - (A_1 B_1) - (A_0 B_0)]x^l + A_0 B_0 \end{aligned} \tag{2.26}$$

Este proceso puede ser repetido recursivamente para resolver los productos de polinomios hasta que los productos sean fácilmente computables. El algoritmo 2, mostrado a continuación, expone el método de Karatsuba para multiplicar dos polinomios de grado t.

Algoritmo 2 Multiplicación polinomial de Karatsuba.

Entrada: $a(x), b(x) \in \mathbb{F}_2[x]$ de grado t.

Salida: $c(x) \in \mathbb{F}_2[x]$ de grado $2t$ tal que $c(x) = a(x)b(x)$

1: **if** $t = 0$ **then**
2: **return** $c_0 \leftarrow a_0 \odot b_0$
3: **end if**
4: $l \leftarrow \lceil \frac{t}{2} \rceil$
5: $A_1 \leftarrow (a_{t-1}, \ldots, a_l), \quad A_0 \leftarrow (a_{l-1}, \ldots, a_0)$
6: $B_1 \leftarrow (b_{t-1}, \ldots, b_l), \quad B_0 \leftarrow (b_{l-1}, \ldots, b_0)$
7: $\alpha \leftarrow \text{Karatsuba}(A_0, B_0)$
8: $\beta \leftarrow \text{Karatsuba}(A_0 + A_1, B_0 + B_1)$
9: $\gamma \leftarrow \text{Karatsuba}(A_1, B_1)$
10: **return** $c(x) = \gamma x^{2l} + (\beta - \gamma - \alpha)x^l + \alpha$

- La reducción módulo $p(x)$, donde $p(x)$ es el polinomio irreducible del campo $\mathbb{F}_2[x]/(p(x))$.

  Sea $c(x)$ el polinomio obtenido después de la multiplicación polinomial, el cual tiene grado $2m-2$:

$$c(x) = c_{2m-2}x^{2m-2} + \cdots + c_m x^m + c_{m-1}x^{m-1} + \cdots + c_1 x + c_0$$

Con el fin de obtener un elemento en el campo $\mathbb{F}_{2^m}$, se debe encontrar el residuo módulo $p(x)$, si denotamos al polinomio irreducible como $p(x) = x^m + r(x)$, para algún $r(x) \in \mathbb{F}_2[x]$, $\deg(r(x)) < m$, entonces la reducción polinomial se realiza mediante la siguiente fórmula:

$$c(x) \equiv (c_{2m-2}x^{m-2} + \cdots + c_m)r(x) + c_{m-1}x^{m-1} + \cdots + c_1x + c_0 \pmod{p(x)} \quad (2.27)$$

El algoritmo 3 muestra la manera de realizar la multiplicación de dos elementos en el campo $\mathbb{F}_{2^m}$. Cabe notar que en el algoritmo 2, la recursión termina hasta que el polinomio es una constante, sin embargo, la recursión puede ser detenida antes si se cuenta con un multiplicador polinomial más eficiente.

Algoritmo 3 Multiplicación en el campo $\mathbb{F}_{2^m}$.

Entrada: $a(x), b(x) \in \mathbb{F}_{2^m}$ de grado menor a m.

Salida: $c(x) \in \mathbb{F}_{2^m}$ tal que $c(x) = a(x)b(x) \pmod{p(x)}$

1: $c'(x) \leftarrow$ Karatsuba$(a(x), b(x))$ {Vía algoritmo 2}

2: **return** $c(x) \equiv c'(x) \pmod{p(x)}$ {Vía ecuación (2.27).}

Elevar al cuadrado

Elevar al cuadrado consiste en multiplicar un elemento del campo consigo mismo. Como se revisó en la sección 2.4.3 en la página 16, el automorfismo de Frobenius es una operación lineal en el campo $\mathbb{F}_{2^m}$. Aprovechando esta propiedad, expondremos cómo la aplicación del automorfismo de Frobenius puede ser computada más rápidamente que el realizar una multiplicación arbitraria.

Sea $a(x) \in \mathbb{F}_{2^m}$ denotado como:

$$a(x) = \sum_{i=0}^{m-1} a_i x^i$$

al elevar al cuadrado se obtiene:

$$a(x)^2 = \left[\sum_{i=0}^{m-1} a_i x^i\right]^2 = \sum_{i=0}^{m-1} a_i x^{2i} \quad (2.28)$$

Si se toma el vector de coeficientes de a, el cuadrado de a es fácilmente calculado al intercalar ceros entre cada coeficiente:

$$\begin{aligned} a &\rightarrow a^2 \\ (a_{m-1}, \ldots, a_2, a_1, a_0) &\rightarrow (a_{2m-2}, 0, \ldots, a_2, 0, a_1, 0, a_0) \end{aligned}$$

Una vez que se insertan los ceros se obtiene $a(x)^2$, el cual es un polinomio de grado $2m-2$. Después se le aplica la reducción modular $b(x) \equiv a(x)^2 \bmod p(x)$, (ecuación (2.27)) para obtener un elemento $b(x) \in \mathbb{F}_{2^m}$.

Raíz cuadrada

La raíz cuadrada de un elemento $a \in \mathbb{F}_{2^m}$ consiste en encontrar un elemento $b \in \mathbb{F}_{2^m}$ tal que $b^2 = a$, denotado como $b = \sqrt{a}$. A continuación, se describe la forma de calcular la raíz cuadrada de elementos en el campo, la cual fue expuesta en [30]. El método básico para calcular $\sqrt{a}$, $a \in \mathbb{F}_{2^m}$, está basado en el Teorema Pequeño de Fermat: $a^{2^m} = a$. Entonces $\sqrt{a}$ puede ser expresado como $\sqrt{a} = a^{2^{m-1}}$, requiriendo $m-1$ elevaciones al cuadrado de a.

Un método aún más eficiente es obtenido bajo la observación de que $\sqrt{a}$ puede ser expresado en términos de $\sqrt{x}$. Sea $a(x) \in \mathbb{F}_{2^m}$, dado que la operación de elevar al cuadrado es lineal, entonces $\sqrt{a}$ puede ser escrito como:

$$\sqrt{a} = \left(\sum_{i=0}^{m-1} a_i x^i \right)^{2^{m-1}} = \sum_{i=0}^{m-1} a_i (x^{2^{m-1}})^i$$

Suponiendo que m es impar, podemos separar $a(x)$ en potencias pares e impares:

$$\begin{aligned} \sqrt{a} &= \sum_{i=0}^{\frac{m-1}{2}} a_{2i} (x^{2^{m-1}})^{2i} + \sum_{i=0}^{\frac{m-3}{2}} a_{2i+1} (x^{2^{m-1}})^{2i+1} \\ &= \sum_{i=0}^{\frac{m-1}{2}} a_{2i} x^i + \sum_{i=0}^{\frac{m-3}{2}} a_{2i+1} (x^{2^{m-1}}) x^i \\ &= \sum_{i \text{ es par}} a_i x^{\frac{i}{2}} + \sqrt{x} \sum_{i \text{ es impar}} a_i x^{\frac{i-1}{2}} \end{aligned} \tag{2.29}$$

De esta forma podemos realizar el cálculo de la raíz cuadrada al extraer de a dos vectores $a_{\text{par}} = (a_{m-1}, \ldots, a_4, a_2, a_0)$ y $a_{\text{impar}} = (a_{m-2}, \ldots, a_5, a_3, a_1)$, (suponiendo que m es impar), para después realizar una multiplicación de campo del vector a_{impar} con el valor constante de $\sqrt{x}$ y finalmente sumar ese resultado con a_{par}.

Cuando el polinomio irreducible posee ciertas características que hacen que la raíz cuadrada sea fácilmente computada, al polinomio se le conoce como poliniomio amigable con las raíces cuadradas. En [6; 11] se describen algunas de las características de estos polinomios.

Inversión

La inversión de campo comúnmente se realiza mediante el algoritmo extendido de Euclides, sin embargo, el algoritmo de Itoh-Tsujii es usualmente más eficiente en el campo $\mathbb{F}_{2^m}$. Este algoritmo fue presentado en [45], el cual fue adaptado a base polinomial en [34] y consiste en generar una secuencia recursiva de elementos.

Dado que el grupo multiplicativo $\mathbb{F}_{2^m}^*$ es cíclico y de orden 2^m-1, para cualquier elemento $a \in \mathbb{F}_{2^m}^*$ se cumple que:

$$a^{-1} = a^{2^m-2} = (a^{2^{m-1}-1})^2 \tag{2.30}$$

Definimos la secuencia $(\beta_k(a) = a^{2^k-1})_{k\in\mathbb{N}}$, para la cual $\beta_0(a) = 1$ y $\beta_1(a) = a$. Si tomamos la ecuación (2.30) podemos escribir la inversión en términos de β como sigue:

$$a^{-1} = (\beta_{m-1}(a))^2 \tag{2.31}$$

Antes de poder encontrar el término $\beta_{m-1}(a)$ se expondrá el concepto de cadena de adición. Una cadena de adición es una secuencia finita de n enteros $\{u_i\}_{i\in[0,n]}$, tal que para cada $i \in [2,n]$ existen números $j,k \in [0,i)$ que satisfacen $u_i = u_j + u_k$.

Para calcular el término $\beta_{m-1}(a)$, es necesario realizar la cadena de adición más corta sobre el término $m-1$. Es fácilmente verificable que para dos números enteros, $j,k \geq 0$, se cumple la siguiente ecuación:

$$\beta_{k+j}(a) = \beta_k(a)^{2^j}\beta_j(a)$$

Para calcular el término $\beta_i(a)$, es suficiente con conocer dos términos $\beta_j(a)$ y $\beta_k(a)$ tal que $i = j+k$. En el algoritmo 4 se presenta la forma de evaluar la inversión del grupo multiplicativo $\mathbb{F}_{2^m}^*$ utilizando el algoritmo de Itoh-Tsujii. Este algoritmo consiste en t multiplicaciones y t elevaciones al cuadrado consecutivas.

Algoritmo 4 Inversión en el grupo multiplicativo $\mathbb{F}_{2^m}^*$ usando algoritmo de Itoh-Tsujii.

Entrada: $a \in \mathbb{F}_{2^m}^*$, una secuencia $\{u_i\}_{i\in[0,t]}$ de longitud t tal que $u_t = m-1$.
Salida: $a^{-1} \in \mathbb{F}_{2^m}^*$.

1: $\beta_{u_0}(a) = a$
2: **for** $i = 1$ **to** t **do**
3: $\quad \beta_{u_i}(a) = \left(\beta_{u_j}(a)\right)^{2^{u_k}} \cdot \beta_{u_k}(a)$ tal que $u_i = u_j + u_k$
4: **end for**
5: **return** $\beta_{u_t}(a)^2$

Inversión simultánea

Algunas veces es necesario calcular múltiples inversiones de elementos del campo, sin embargo por ser una operación costosa en términos de multiplicaciones, en [64] se expone una técnica

sofisticada capaz de realizar inversos múltiples elementos mediante una sola inversión y una serie de productos; a esta técnica se le conoce como el truco de Montgomery, el cual se describe a continuación.

Sean $x_1, \ldots, x_n \in \mathbb{F}_{2^m}$ los elementos a los cuales se les desea calcular su inverso, para ello se realiza el producto:

$$X = \prod_{i=1}^{n} x_i$$

y se obtiene $Y = X^{-1}$. Una vez que se ha obtenido la inversión de X, se pueden obtener todos los elementos inversos mediante:

$$x_i^{-1} = Y \cdot \prod_{j \neq i} x_j$$

lo cual puede ser eficientemente calculado utilizando solamente $3(n-1)$ multiplicaciones.

Resumen

En este capítulo se revisaron algunos tópicos matemáticos relevantes para el desarrollo de esta tesis. Iniciando con las secuencias de Lucas, cuyas propiedades matemáticas son de transcendencia en este trabajo.

Las estructuras algebraicas son a menudo utilizadas para definir conceptos abstractos, la existencia de grupos, anillos y campos son ejemplo de estas estructuras.

Un grupo está definido mediante un conjunto y una operación binaria que actúa sobre ese conjunto. Para que la operación forme un grupo se deben cumplir las siguientes propiedades: cerradura, asociatividad, existencia de un elemento identidad y la existencia de inversos. Adicionalmente, si la operación es conmutativa, se dice que el grupo formado es abeliano.

En algunos grupos el problema del logaritmo discreto es computacionalmente intratable, el cual luce atractivo para aplicaciones criptográficas basadas en este problema.

Un anillo lo conforma un conjunto, una operación binaria que forma un grupo abeliano y una segunda operación binaria que es distributiva sobre la primera operación. Esta última operación cumple con las siguientes propiedades: cerradura, asociatividad, existencia de un elemento identidad.

Un campo está formado por un conjunto (finito o infinito) junto con dos operaciones binarias, las cuales forman dos grupos abelianos, y una de ellas posee la propiedad de distributividad sobre la otra. Un campo finito binario es el grupo finito más simple que existe al poseer sólo dos elementos y se denota como $\mathbb{F}_2 = \langle \{0,1\}, \oplus, \odot \rangle$.

Las extensiones de campo forman un espacio vectorial sobre el campo base. En el caso del campo $\mathbb{F}_{2^m}$, el cálculo de la raíz cuadrada y la elevación al cuadrado son operaciones lineales, lo cual facilita su cómputo.

En este trabajo, las operaciones realizadas sobre la extensión de campo $\mathbb{F}_{2^m}$ son los bloques fundamentales para el desarrollo de la aritmética de curvas elípticas, la cual se analizará en el capítulo 4.

3

Programación en paralelo

Pies, para que los quiero si tengo alas para volar.

Frida Kahlo

UN programa puede ser visto como una secuencia de instrucciones que son ejecutadas por un procesador de forma secuencial. La mayoría de estas instrucciones manejan accesos a memoria y/o realizan operaciones con los registros del procesador.

Una computadora está limitada a un conjunto de instrucciones que puede ejecutar y a un conjunto de registros sobre los cuales operar. El conjunto de instrucciones y registros que posee determina la arquitectura de la computadora.

Los conjuntos extendidos de instrucciones proveen a los procesadores de características avanzadas para la ejecución de los programas en paralelo.

Una fase importante al momento de optimizar un programa consiste en realizar la mejor selección de instrucciones. La relación entre la mejor elección de instrucciones y la velocidad de ejecución resultante se encuentra principalmente en la forma de explotar el paralelismo en los procesadores modernos.

SECCIÓN 3.1

Aceleración de un programa paralelo

El factor de aceleración es un indicador para determinar el beneficio potencial al utilizar cómputo paralelo. Típicamente se mide como el tiempo que toma un programa en completarse de manera secuencial, dividido contra el tiempo que toma ejecutarlo en un sistema de N procesos en paralelo [33]. La aceleración $S(N)$ de un programa paralelo se encuentra definida como:

$$S(N) = \frac{T_P(1)}{T_P(N)} \tag{3.1}$$

donde $T_P(1)$ es el tiempo de procesamiento de un algoritmo en su versión secuencial, mientras que $T_P(N)$ es el tiempo de procesamiento utilizando N unidades de procesamiento paralelo.

En la situación ideal de un algoritmo totalmente paralelizable, (cuando el tiempo de comunicación entre las unidades de procesamiento y el uso de memoria es desdeñable), se tiene que $T_P(N) = T_P(1)/N$, lo cual de acuerdo a la ecuación (3.1) permite una aceleración:

$$S(N) = N \tag{3.2}$$

Este crecimiento lineal es inusualmente visto en los programas de acuerdo a múltiples factores, como lo son la sincronización, el uso de secciones críticas, los bloqueos mutuos, etcétera, los cuales se examinarán en las siguientes secciones.

3.1.1. Ley de Amdahl

Sea A un algoritmo o tarea que está compuesta por una fracción paralelizable f y una fracción secuencial $1-f$. El tiempo necesario para procesar la tarea A en una sola unidad de procesador se encuentra dada por:

$$T_P(1) = t_s \tag{3.3}$$

donde t_s es el tiempo de procesamiento de la versión secuencial del algoritmo A, como se puede observar en la figura 3.1(a).

Por otro lado, si se cuenta con N unidades de procesamiento paralelo, el tiempo que toma en ejecutar el algoritmo A es la suma del tiempo que el procesador necesita para ejecutar la parte secuencial más el tiempo necesario para ejecutar la parte paralelizable, por tanto se tiene:

$$T_P(N) = (1-f)t_s + \frac{f}{N}t_s \tag{3.4}$$

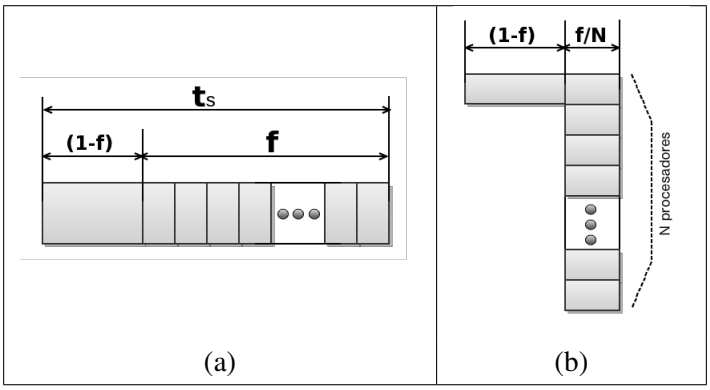


Figura 3.1: Procesamiento secuencial y en paralelo de un programa.

donde la única aceleración se presenta en la parte paralelizable que es distribuida sobre las N unidades de procesamiento. En la figura 3.1(b) se puede verificar gráficamente esta ecuación.

De acuerdo a la definición de aceleración en la ecuación (3.1), se tiene:

$$\begin{aligned} S(N) &= \frac{T_P(1)}{T_P(N)} \\ &= \frac{t_s}{(1-f)t_s + \frac{f}{N}t_s} \\ &= \frac{1}{(1-f) + \frac{f}{N}} \end{aligned} \tag{3.5}$$

a esta última ecuación se le conoce como la ley de Amdahl descrita en [8], de la cual se obtienen las siguientes conclusiones:

- Si se cuenta con infinitas unidades de procesamiento, es decir:

$$\lim_{N\to\infty} S(N) = \frac{1}{1-f} \tag{3.6}$$

el límite de la aceleración obtenida es igual a la fracción de la parte secuencial. Esto quiere decir que el grado de paralelismo de un algoritmo es dependiente del algoritmo mismo, no así de los recursos con los que se cuenten.

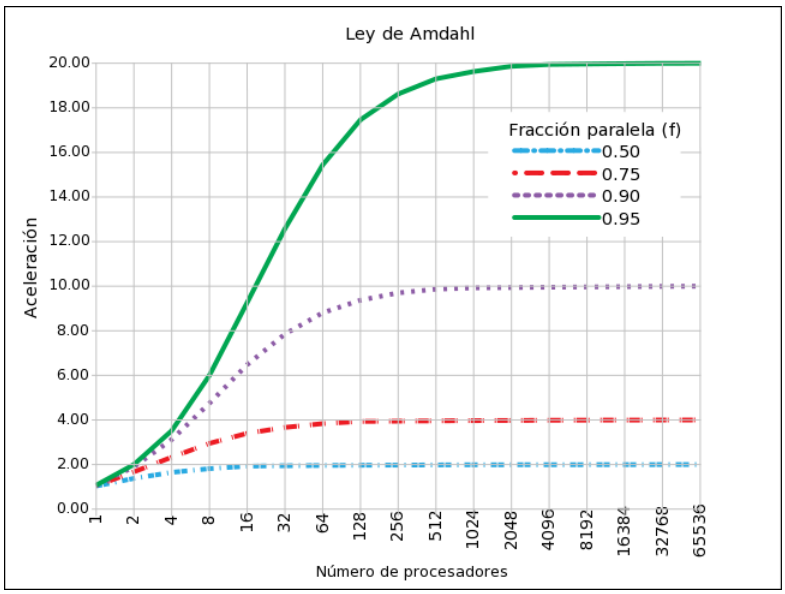


Figura 3.2: Comportamiento de la ley de Amdahl para diferentes valores de f.

- La ley de Amdahl puede servir como guía para determinar el impacto de una mejora en el algoritmo sobre el rendimiento y la forma de distribuir los recursos, en pro de la relación costo-rendimiento.
- En la figura 3.2 se muestra la relación entre el número de procesadores contra diferentes valores de f, como se puede observar el límite de la aceleración tiende a un valor constante dado por la ecuación (3.6).
- Con el fin de obtener una aceleración considerable se debe cumplir la siguiente desigualdad:

$$1 - f \ll \frac{f}{N} \tag{3.7}$$

lo cual indica que la fracción f debe estar cercana a la unidad, especialmente cuando N es grande.

Para mayores detalles sobre la ley de Amdahl, por favor refiérase a [42] en la sección 1.9, así como también en el capítulo 1 de [33].

3.1.2. Ley de Gustafson

Gustafson en [39], observó que el paralelismo en una aplicación se aumenta conforme el tamaño del problema se incrementa. Las predicciones de acuerdo a la ley de Amdahl son un tanto pesimistas,

puesto que supone que la fracción del código paralelizable es fija y no depende del tamaño del problema.

Para derivar la fórmula de Gustafson-Barsis de la aceleración, suponemos que ante la existencia de N unidades de procesamiento en paralelo, el tiempo que toma en procesar la tarea en N unidades de procesamiento es normalizada a 1, es decir:

$$T_P(N) = 1 \tag{3.8}$$

Cuando el programa se ejecuta en un solo procesador se debe realizar la parte secuencial $1-f$ más la parte paralela f:

$$T_P(1) = (1-f) + Nf \tag{3.9}$$

donde N es el número de unidades de procesamiento. Por lo tanto, la aceleración $S(N)$ es igual a:

$$S(N) = 1 + (N-1)f \tag{3.10}$$

Como se observa en la figura 3.3, se puede obtener una cantidad significativa de aceleración a pesar de tener valores pequeños para f, esta aceleración mejora conforme N se vuelve mayor. Para asegurar una aceleración significativa, se debe cumplir la siguiente desigualdad:

$$f(N-1) \gg 1 \tag{3.11}$$

comparando esta última con la desigualdad (3.7), es notorio que las restricciones para obtener mejores aceleraciones son mucho más relajadas en el caso de la ley de Gustafson-Barsis.

La ley de Gustafson-Barsis determina el límite en la aceleración que puede obtenerse cuando se paraleliza un programa, y plantea que un programa con datos suficientemente grandes puede ser paralelizado de manera eficiente, obteniendo una aceleración proporcional al número de unidades de procesamiento.

SECCIÓN 3.2

Paradigmas de programación paralela

Cambiar el enfoque para convertir el modelo de programación secuencial hacia uno paralelo, consiste fundamentalmente en descomponer el problema en distintas tareas, las cuales en conjunción resuelven el problema inicialmente planteado.

Es necesario identificar las dependencias entre las tareas, para de esta forma identificar las tareas que son candidatas a paralelizarse. Existen diferentes construcciones para implementar el paralelismo en un programa, los cuales son: paralelismo a nivel de instrucción, paralelismo a nivel de datos y el paralelismo a nivel de tareas. Estas técnicas se describen a detalle en el resto del capítulo.

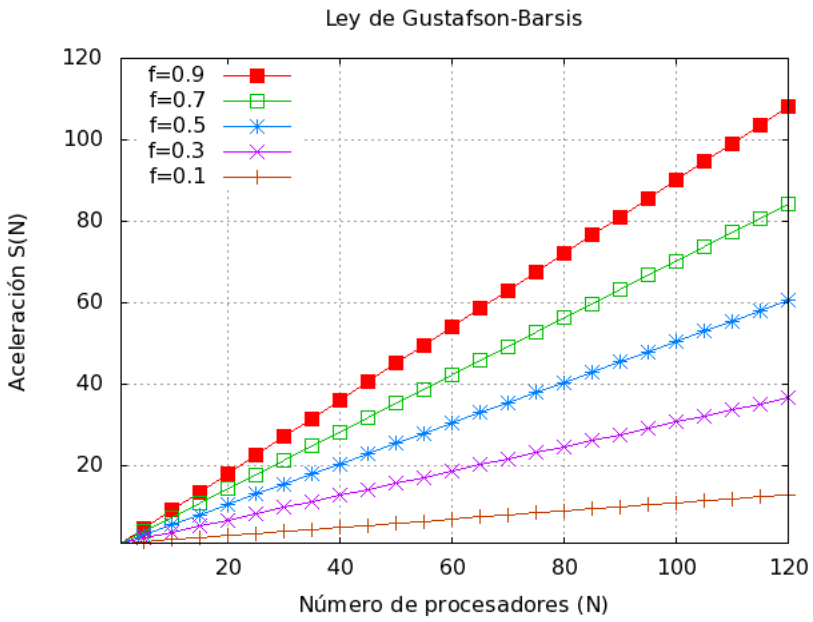


Figura 3.3: Comportamiento de la ley de Gustafson-Barsis para diferentes valores de f.

SECCIÓN 3.3

Paralelismo a nivel de instrucción

Dado un conjunto de instrucciones cuyas entradas y salidas son independientes entre sí, entonces el procesador puede realizarlas en paralelo. A este tipo de paralelismo se le conoce como paralelismo a nivel de instrucción. Un programa con alto nivel de paralelismo a nivel de instrucción resulta en ejecuciones más rápidas. También puede ser referido como el número de operaciones que una computadora puede procesar simultáneamente.

3.3.1. Computadora superescalar

Una computadora superescalar es aquella que intenta ejecutar en cada ciclo de reloj varias instrucciones en paralelo, a pesar de que la máquina esté ejecutando un programa secuencial [68]. La idea de la ejecución superescalar fue originalmente referida por Agerwala y Cocke en [5].

Un programa secuencial será ejecutado correctamente bajo la siguiente premisa: una instrucción puede iniciar su ejecución una vez que haya finalizado la instrucción previa a ésta; lo cual representa un problema para los procesadores superescalares.

La primer tarea para un procesador superescalar consiste en identificar, para cada programa,

cuáles instrucciones presentan dependencias con la instrucción a ejecutar. Para realizar esto, el procesador debe verificar si los operandos involucrados, ya sean registros o bien localidades de memoria, interfieren con otros operandos de alguna instrucción sucesiva.

3.3.2. Modelo de tubería

Un procesador superescalar realiza su mejor esfuerzo en ejecutar múltiples operaciones por ciclo de reloj, con el fin de lograr cierto nivel de paralelismo. En lugar de que el procesador sea capaz de ejecutar *n* operaciones por ciclo de reloj, se puede obtener el mismo rendimiento al descomponer las unidades funcionales del procesamiento, acelerando la velocidad de reloj por un factor de *n*, pero ejecutando una instrucción por ciclo de reloj [49].

Los procesadores actuales implementan la ejecución de instrucciones utilizando el modelo de tubería, el cual está basado en la observación de que la ejecución de instrucciones se lleva a cabo mediante múltiples fases.

Para instrucciones independientes las fases se pueden translapar, de esta manera mientras una instrucción está siendo ejecutada, la instrucción siguiente estará siendo decodificada y la siguiente a ésta puede estar siendo cargada de memoria y así sucesivamente. En la figura 3.4 se puede observar el comportamiento del modelo de tubería.

El modelo de tubería permite reducir la latencia entre instrucciones acelerando el rendimiento, es decir, el número de instrucciones que pueden ser ejecutadas por unidad de tiempo.

3.3.3. Arquitectura VLIW

A finales de la década de 1970, emergió un nuevo estilo de paralelismo a nivel de instrucción, llamado en inglés *very long instruction word* (VLIW) [68]. Una máquina VLIW es una computadora que a diferencia de una computadora superescalar, las instrucciones en paralelo deben ser explícitamente empaquetadas por el compilador en palabras de gran longitud. El término y el concepto de la arquitectura VLIW, fue inventado por Josh Fisher a inicios de 1980, publicado en [26; 65].

Un programa a ejecutarse en una arquitectura VLIW debe especificar exactamente cuáles operaciones deberán ser ejecutadas, así como también cuándo una operación debe ser tratada como independiente de todas las demás operaciones que están siendo ejecutadas al mismo tiempo.

En una arquitectura VLIW es importante distinguir entre una instrucción y una operación. Una operación es una unidad de cómputo, por ejemplo, una suma, una carga de memoria, un salto, etcétera, las cuales son comúnmente referidas como instrucciones bajo el contexto de arquitecturas secuenciales. Una instrucción VLIW es un conjunto de operaciones que se pretende se ejecuten simultáneamente.

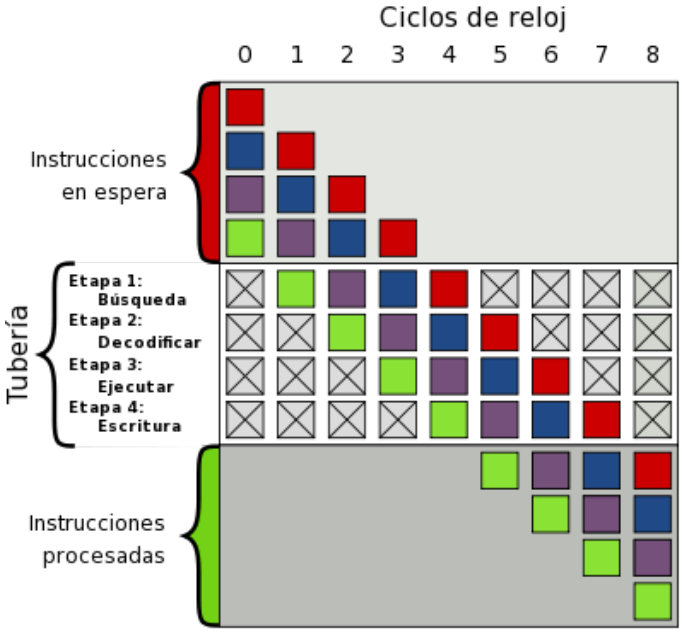


Figura 3.4: Ejecución de modelo de tubería

Es tarea del compilador decidir cuáles operaciones deberán ir en cada instrucción. A este proceso se le conoce como *calendarización*. Conceptualmente, el compilador calendariza un programa al emular en tiempo de compilación el flujo de datos del programa. Todas las operaciones que se suponen inician al mismo tiempo son empaquetadas en una sola instrucción VLIW.

3.3.4. Dependencia de datos

Una limitante que reduce el grado de paralelismo a nivel de instrucción es la dependencia de datos entre instrucciones. A continuación se darán algunos argumentos acerca de esta problemática.

La dependencia principal entre instrucciones es la dependencia de datos, la cual se presenta cuando la salida de una instrucción i es la entrada de una instrucción j. Asímismo, las dependencias de datos poseen transitividad, es decir, si existe una dependencia entre una instrucción i y una instrucción k, y también existe una dependencia entre la instrucción k y la instrucción j, entonces existirá una dependencia entre la instrucción i y j. Por lo tanto, este tipo de instrucciones no pueden ser procesadas en paralelo.

Otro tipo de dependencia de datos que frecuentemente aparece en los programas, son aquellas relacionadas con el uso de sentencias condicionales. Resulta impredecible ejecutar las instrucciones que prosiguen a una sentencia condicional, dado que no se tiene certeza, hasta el tiempo de ejecu-

ción, sobre los valores involucrados en la condición. Tal es el caso de sentencias `if-then-else` o ciclos condicionales como `while` o `do-while`.

Para lograr un alto nivel de paralelismo a nivel de instrucción se recomienda tanto como sea posible evitar el uso de sentencias condicionales.

3.3.5. Ejecución fuera de orden

La mayoría de los procesadores ejecutan las instrucciones fuera de orden, es decir, el procesador adelanta la ejecución de instrucciones que no sean dependientes con el fin de lograr mayor velocidad de ejecución.

Una de las ventajas de la ejecución fuera de orden es que una vez que el programa ha sido compilado, se puede obtener un rendimiento razonable en diferentes arquitecturas. Dado que el procesador puede calendarizar las instrucciones de acuerdo a la información conocida en tiempo de ejecución.

Una característica importante de los procesadores que permiten la ejecución fuera de orden es que usualmente contienen más registros. Este incremento de registros es usado para resolver dependencias de nombre, al asignar dinámicamente los registros básicos de la arquitectura hacia otros registros disponibles. A este proceso se le conoce como renombrado de registros.

Para mayores detalles acerca de las técnicas de renombrado de registros consúltese [42].

3.3.6. Predicción de ramificaciones

Debido a la necesidad de cumplir con las dependencias de control, existen riesgos cuando se presentan ramificaciones de código, las cuales perjudican el rendimiento del modelo de tubería.

Una forma de reducir el número de ramificaciones, es desenrollando los ciclos (conocido en inglés como *loop unrolling*), lo cual implica evitar las sentencias condicionales en los ciclos cuando se tiene certeza de los límites del ciclo.

Además se puede reducir la pérdida de rendimiento al tratar de predecir el comportamiento de las ramificaciones. Este comportamiento puede ser predecido tanto estáticamente, en tiempo de compilación, como también dinámicamente, es decir en tiempo de ejecución.

El método más simple para predecir las ramificaciones consiste en tomar una de ellas y ejecutarla, sin embargo en promedio la probabilidad de tomar la ramificación incorrecta es alta (alrededor de 0.5).

Una técnica más precisa consiste en predecir en base a información recolectada de ejecuciones previas. El punto clave que hace que esta técnica sea relevante, es que el comportamiento de las ramificaciones expone una distribución bimodal, es decir, una ramificación individual es muy sesgada

hacia el estado en que es ejecutada o no ejecutada.

Para realizar la predicción dinámica de ramificaciones se utiliza una tabla histórica, la cual es un área de memoria indexada mediante los bits menos significativos de la dirección de la instrucción en la ramificación. La memoria contiene un bit que indica si la rama fue ejecutada o no. Con esto no se asegura que la predicción haya sido correcta, dado que la predicción es un indicio que se supone correcto. Si el indicio fuése incorecto el bit de predicción es invertido. Para reducir la probabilidad de fracaso se utilizan esquemas de 2 bits. Logrando en promedio, una probabilidad de tomar la ramificación incorrecta de entre (0.01 - 0.18), (cfr. [42]).

SECCIÓN 3.4

Paralelismo a nivel de datos

Cuando el mismo procesamiento es aplicado a un conjunto de datos independientes, esto puede convertirse en un procesamiento en paralelo. A este tipo de paralelismo se le conoce como paralelismo a nivel de datos. El procesamiento por lotes toma ventaja de este enfoque.

En [28], Flynn categoriza a las computadoras de alto desempeño en 4 clases:

- *Single Instruction Stream - Single Data Stream* (SISD)
- *Single Instruction Stream - Multiple Data Stream* (SIMD)
- *Multiple Instruction Stream - Single Data Stream* (MISD)
- *Multiple Instruction Stream - Multiple Data Stream* (MIMD)

en donde el término *stream* se utiliza para referirse a la secuencia de datos o de instrucciones que una máquina procesa durante la ejecución de un programa. Actualmente es muy común que las arquitecturas de computadoras implementen alguna o varias categorías.

La primer categoría (SISD) es el modo de ejecución habitual de las computadoras, en el cual se ejecuta en un solo procesador un conjunto de instrucciones que utilizan datos de un espacio de memoria específico.

Detallando la categoría SIMD, es aquella que realiza en el mismo instante, una operación sobre un conjunto de datos. Los procesadores actuales poseen conjuntos extendidos de instrucciones que los dotan de instrucciones SIMD, éstos son cada vez más utilizados para el desarrollo de programas de alto rendimiento.

Frecuentemente se utilizan instrucciones vectoriales al implementar este paradigma, las cuales mantienen múltiples valores en un registro vectorial, es decir, una operación aritmética se aplica

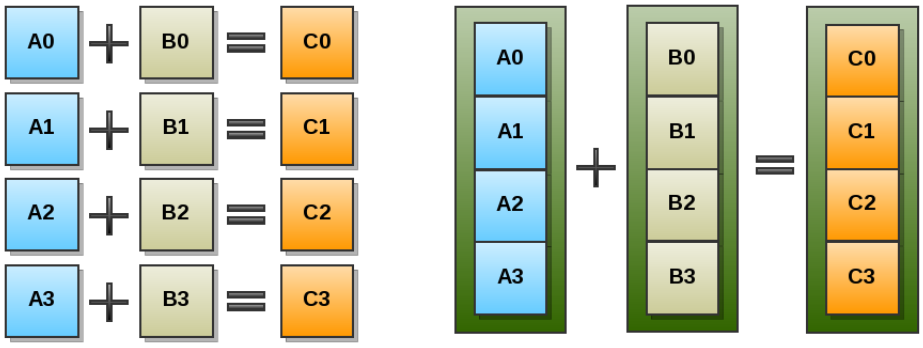


Figura 3.5: Comparación entre cuatro instrucciones convencionales y una instrucción SIMD.

sobre *n* valores de *b* bits almacenados en un registro de *nb* bits, tal como se muestra en la figura 3.5.

3.4.1. Streaming SIMD Extensions

El conjunto de instrucciones *Streaming SIMD Extensions* (usaremos la abreviación SSE por el resto del capítulo) [75] aparecieron en 1999 en los procesadores de Intel® dotándolos de 70 nuevas instrucciones, la mayoría de las cuales trabajan sobre datos de punto flotante.

Las nuevas instrucciones incrementan el rendimiento de un programa, al realizar la misma operación sobre múltiples datos. Para dar soporte a las instrucciones SSE, la memoria que contiene los datos debe estar alineada a 16 bytes, lo cual significa que la dirección de memoria debe ser un múltiplo de 16, ésto con el propósito de hacer un manejo eficiente de memoria.

Es importante remarcar que estas instrucciones pueden ser vistas como instrucciones vectoriales, ya que operan sobre datos en localidades consecutivas de memoria.

Los procesadores con el juego de instrucciones SSE agregan ocho registros adicionales de 128 bits, conocidos como los registros `XMM0` al `XMM7`. Estos registros son versátiles, pues pueden guardar desde 1 valor de 128 bits, 2 valores de 64 bits, 4 valores de 32 bits, 8 valores de 16 bits o 16 valores de 8 bits como se puede observar en la figura 3.6.

3.4.2. SSE2

En el año 2000 se presentó el nuevo conjunto de instrucciones SSE2, proporcionando a los procesadores de 144 nuevas instrucciones, en [44] se demuestra su uso en la multiplicación de números enteros de mayor magnitud que el tamaño de la palabra nativa en la máquina.

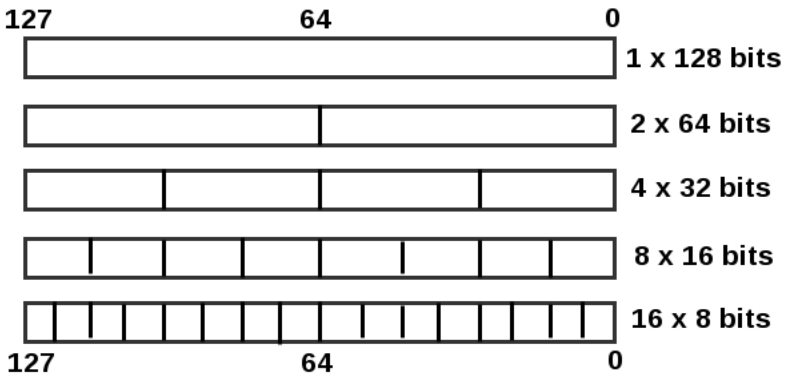


Figura 3.6: Versatilidad de los registros `XMM` provistos por SSE.

Dentro de las principales características que provee este nuevo conjunto de instrucciones se encuentran:

- Permite mezclar operaciones enteras SIMD con operaciones de punto flotante.
- Incluye un conjunto de instrucciones relacionadas al manejo de la memoria caché, con el propósito de evitar cargas innecesarias de memoria desde la RAM o disco.
- Incluye 8 registros `XMM` adicionales a los existentes en SSE, teniendo un total de 16 registros del `XMM0` al `XMM15`.

3.4.3. SSE3 y SSSE3

Para 2004 se dió a conocer las nuevas instrucciones Prescott (PNI [20]), conocidas como SSE3, agregan 13 nuevas instrucciones que aceleran el rendimiento de SSE y SSE2. El cambio más notable, es la capacidad de trabajar horizontalmente en el registro, a diferencia de las instrucciones anteriores que trabajaban verticalmente. Este enfoque horizontal permite realizar una operación entre valores que estan guardados en un solo registro, en la figura 3.7 se muestra el procesamiento horizontal de una suma de enteros.

Dos años más tarde, se dieron a conocer 32 nuevas instrucciones bajo el nombre de *Supplemental Streaming SIMD Extensions* (SSSE3 [21]), Las cuales agregan funcionalidad para el manejo de números negativos, valores absolutos, operaciones horizontales y operaciones tales como corrimientos y permutaciones de bits.

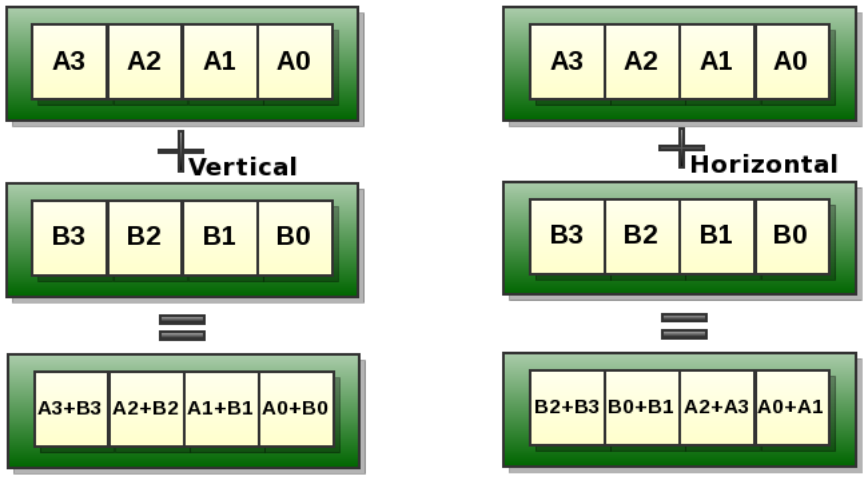


Figura 3.7: Diferencia entre el procesamiento vertical y el procesamiento horizontal en los registros XMM.

3.4.4. SSE4

Intel® provee 54 nuevas instrucciones, las primeras 47 son referenciadas mediante el conjunto SSE4.1 presentes en la arquitectura Penryn, mientras que las 7 restantes son conocidas como SSE4.2, las cuales están disponibles a partir de la arquitectura Nehalem.

Destaca la instrucción para la evaluación del algoritmo CRC32 (código de redundancia cíclica descrito extensamente en [71]), así como también algunas operaciones para el procesamiento de cadenas de caracteres y la operación de conteo de bits sobre un registro de 64 bits, destinada para acelerar aplicaciones de reconocimiento de patrones.

3.4.5. AES-NI

Una de las últimas extensiones a los procesadores es el conjunto de instrucciones AES-NI, [35; 36], disponible en la familia de procesadores Intel Core®, éstos basados en la micro-arquitectura de 32 nanómetros llamada Westmere.

Este nuevo conjunto de instrucciones consiste en 7 nuevas instrucciones, 6 de ellas se refieren a la implementación empotrada de funciones para desarrollar el algoritmo de cifrado simétrico AES, descrito extensamente en [22; 24]. Antes del lanzamiento de este conjunto de instrucciones, se habían realizado diversos esfuerzos por acelerar el rendimiento de AES, tales como [76; 77; 78]. Las instrucciones relacionadas con AES son:

- AESENC y AESDEC realiza una ronda de cifrado o descifrado, respectivamente, del algoritmo

AES, toma como argumentos el estado (mensaje) y la llave de ronda, obtiene como salida el estado procesado después de una ronda del algoritmo.

- `AESENCLAST` y `AESDECLAST` realiza la última ronda del algoritmo de AES. Dado que ésta es diferente al resto de las rondas.
- `AESKEYGENASSIST` es utilizada para generar las llaves de ronda usadas para el cifrado.
- `AESIMC` es usada para convertir las llaves de ronda de cifrado de forma que sean útiles para la fase de descifrado.

Éstas son las 6 instrucciones que permiten cifrar volúmenes de información mediante AES. La implementación reportada en [36] utiliza el conjunto de instrucciones AES-NI, utilizando una llave privada de 128 bits, ahí se logra una velocidad de alrededor 1.27 cpb. (ciclos de reloj por byte) para cifrar un mensaje de 1 KB.

PCLMULQDQ. La séptima instrucción perteneciente al conjunto de instrucciones AES-NI, se llama `PCLMULQDQ`, la cual realiza la multiplicación polinomial de elementos en $\mathbb{F}_{2^{64}}$, tal como se describió en la sección 2.4.4 de la página 17, donde se introdujo el concepto de multiplicación polinomial.

Esta nueva instrucción presentada en [37; 38], fue utilizada por primera vez para implementar el modo de operación *Galois Counter Mode* (AES-GCM).

La latencia de esta instrucción actualmente oscila entre 8 a 14 ciclos de reloj, logrando menor latencia si se utiliza consecutivamente con datos que no presenten dependencias entre sí, en promedio la latencia efectiva es de 10 ciclos de reloj, de acuerdo a lo reportado en [29].

Esta operación es costosa comparada con la multiplicación de enteros que tan sólo toma 3 ciclos de reloj, en la misma arquitectura. Lo cual levanta especulaciones acerca de que Intel® haya optado por establecer un compromiso entre área del circuito y el rendimiento del mismo.

3.4.6. AVX

El conjunto de instrucciones AVX (del inglés *Advanced Vector Extensions*) presentado en [3; 25], es una extensión a los procesadores modernos, la cual provee de 16 nuevos registros de 256 bits, renombrando a los anteriores `XMM0-XMM15` por `YMM0-YMM15`, este nuevo conjunto de instrucciones está presente a partir de la arquitectura Sandy Bridge de Intel® liberada en enero de 2011.

Una característica a remarcar es el uso de instrucciones de tres operandos, es decir, al realizar una operación el resultado de dos operandos fuente se carga en un tercer operando destino. A este

tipo de código también se le conoce como código no-destructivo, ya que en implementaciones anteriores de SSE, las operaciones cargaban el resultado de la operación en uno de los operandos fuente, lo cual destruía el dato que estaba anteriormente almacenado. Para evitar esta destrucción de datos, el compilador realiza cargas temporales en registros o direcciones de memoria, resultando en más instrucciones de código. Mediante código de tres operandos es posible direccionar tres registros sin cargas adicionales de memoria.

A diferencia de las instrucciones SSE que requieren que los datos esten alineados a 16 bytes, en el caso de las instrucciones AVX, los datos deben estar alineados a 32 bytes.

Un procesador con tecnología AVX puede ejecutar código escrito en SSE, sin embargo AVX proporciona el esquema de código VEX, el cual agrega un prefijo a las instrucciones SSE de un programa en tiempo de compilación, produciendo que se aceleren las operaciones SSE y extendiendo los registros `XMM` a los registros `YMM`. Además el prefijo VEX permite incrementar el tamaño en bits del código de operación de la instrucción, lo cual dará soporte para nuevos conjuntos de instrucciones, como se detalla en [3].

Las principales aplicaciones de las instrucciones AVX están enfocadas al procesamiento de cálculos con datos de punto flotante.

SECCIÓN 3.5

Paralelismo a nivel de tareas

En esta sección se detallan las técnicas de programación referentes al paralelismo a nivel de tareas, perteneciente a la categoría MIMD de la taxonomía de Flynn.

Existen programas que involucran muchas tareas que son parcialmente independientes, ocasionalmente algunas de ellas requieren intercambio de información.

Las arquitecturas MIMD tienen múltiples procesadores, cada uno ejecuta una secuencia independiente de instrucciones. En muchos de los casos, cada procesador ejecuta un proceso diferente. Se define un proceso, como el segmento de código que puede ser ejecutado independientemente, el estado del proceso contiene toda la información necesaria para ejecutar el programa en el procesador.

Muchos de los procesadores que implementan el paradigma MIMD, dan soporte a la ejecución en paralelo a través de:

- Hilos o procesos ligeros.
- Arquitecturas multinúcleo.
- Memoria distribuida

3.5.1. Programación con hilos

Es útil tener la capacidad de ejecutar múltiples procesos en un sólo programa, compartiendo el código, así como su espacio de direcciones. Cuando múltiples procesos comparten código y datos de esta forma, a estos procesos se les conoce como hilos.

Actualmente el término hilo es a menudo utilizado para referirse a un flujo de ejecución a través del código del proceso, con su propio contador de programa, registros del sistema y pila de datos. Los hilos son la forma común para mejorar el rendimiento de un programa.

Un hilo intenta reducir el costo extra proveniente del cambio de contexto cuando se utilizan procesos. Un sistema con exactamente un hilo es equivalente a un proceso clásico. Cada hilo pertenece a exactamente un proceso, de tal forma que ningún hilo puede existir fuera de un proceso.

Para lograr el funcionamiento adecuado de la ejecución de un programa con múltiples hilos es necesario administrar el uso del procesador, ya que es el recurso principal de cómputo. A esta administración se le conoce como calendarización de procesos, y es el administrador de procesos quien se encarga de remover el proceso actual que se está ejecutando y realizar la selección del siguiente proceso a ejecutar. La calendarización del procesador es la base de un sistema multiprogramado, cuya finalidad consiste en aumentar el uso del procesador.

Un sistema multihilo generalmente reparte el tiempo de procesador entre cada hilo, haciéndolo tan rápido que da la apariencia de que los hilos se ejecutan al mismo tiempo.

A finales de 1990, la idea de ejecutar simultáneamente instrucciones provienentes de múltiples hilos, fue implementada en el procesador Pentium 4, bajo el nombre de tecnología *Hyper-Threading*, [19]. La tecnología *Hyper-Threading* (HT) impulsa el rendimiento al permitir que varios hilos se ejecuten al mismo tiempo en un sólo procesador, compartiendo los mismos recursos del procesador. Un procesador con tecnología HT consiste en dos procesadores lógicos, cada uno puede ser individualmente detenido, interrumpido o redireccionado a ejecutar un hilo en específico. Cada procesador lógico contiene su propio conjunto de registros de datos, registros de segmento, registros de control, así como también su propio controlador avanzado de interrupciones programables (APIC). En la figura 3.8, a la izquierda se presenta un procesador tradicional, mientras que a la derecha se observa la arquitectura de un sistema con tecnología HT.

Algunas veces a los hilos se les conoce como procesos ligeros, similitudes y diferencias son listadas en la tabla 3.1.

3.5.2. Cómputo con múltiples núcleos de procesamiento

Al inicio de la década de 1990, los diseñadores de procesadores se han enfocado en las ventajas de incluir en un solo circuito integrado dos o más unidades de procesamiento, estas unidades de

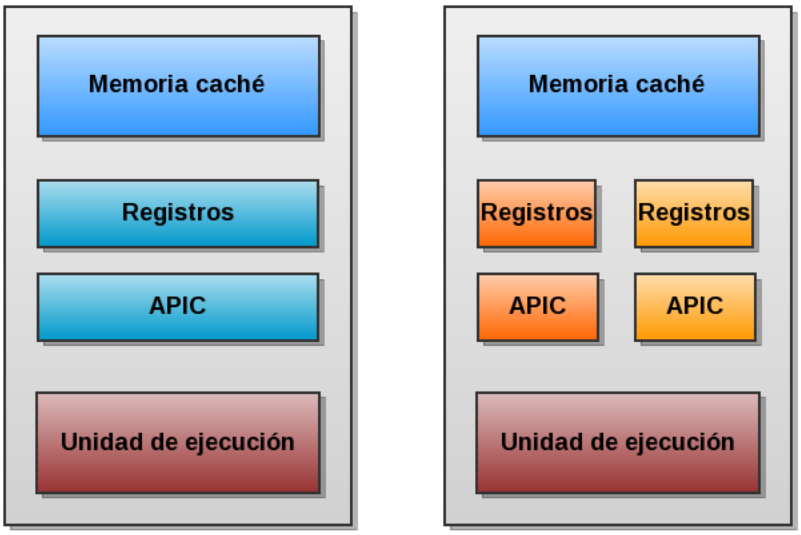


Figura 3.8: Comparación entre un procesador convencional (izquierda) y uno con la tecnología *Hyper-Threading* (derecha).

Proceso	Hilo
Son llamados procesos pesados.	Son llamados procesos ligeros.
El cambio de contexto necesita comunicarse con el sistema operativo.	El cambio de contexto de un hilo no necesita comunicarse con el sistema operativo, sólo causa una interrupción al kernel.
Cada proceso ejecuta el mismo código, sin embargo cada uno tiene su propia memoria.	Todos los hilos comparten el mismo conjunto de apuntadores a archivos y direcciones de memoria.
Mayor cantidad de recursos al momento de ejecutarse.	Baja cantidad de recursos al utilizar hilos
Cada proceso es independiente de los otros.	Un hilo puede leer y escribir en la memoria de otro hilo e incluso causar la finalización de otro hilo.

Tabla 3.1: Diferencias entre hilos y procesos descritas en [33]

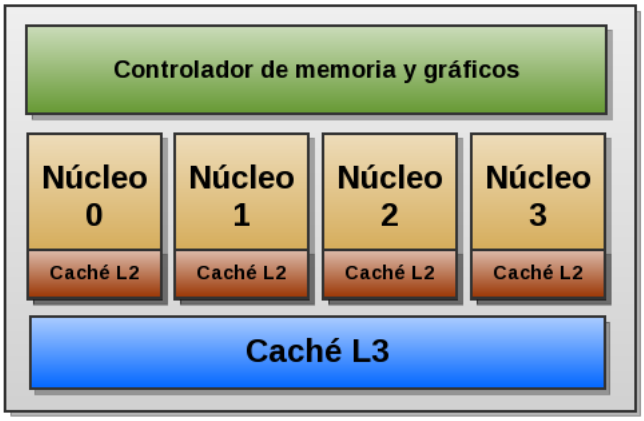


Figura 3.9: Arquitectura básica de un procesador multinúcleo.

procesamiento son denominadas como núcleos, los cuales permiten ejecutar secuencias de instrucciones de forma totalmente paralela.

La programación multinúcleo es el paradigma de programación que supone que existen n unidades de procesamiento independientes dentro de un sistema de cómputo. Para el sistema operativo, cada núcleo de ejecución representa un procesador lógico con recursos propios. Cada núcleo debe ser controlado separadamente, y el sistema operativo puede asignar diferentes programas a diferentes núcleos para obtener una ejecución en paralelo.

La plataforma multinúcleo permite a los desarrolladores optimizar los programas al particionar diferentes cargas de trabajo en diferentes núcleos de procesamiento, lo cual resulta en la aceleración del rendimiento de la aplicación.

En la figura 3.9 se muestra la arquitectura básica de un procesador multinúcleo. El circuito integrado puede contener dos o más núcleos de procesamiento, cada uno con unidades de aritmética de enteros, punto flotante y acceso directo a memoria. La memoria caché (L3) que comparten permite que exista comunicación entre los núcleos. A continuación se describen brevemente los tipos de memoria disponibles por el procesador.

Jerarquía de memoria. En las arquitecturas multinúcleo, la memoria juega un papel importante en el rendimiento del sistema. La memoria se encuentra repartida por todo el sistema, a través de niveles jerárquicos, los cuales se observan en la figura 3.10 y se detallan a continuación:

- Los registros del procesador son el área de memoria de más rápido acceso para el procesador, sin embargo carecen de gran capacidad.

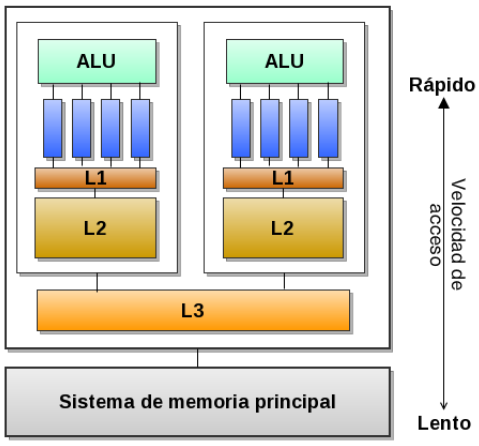


Figura 3.10: Jerarquía de memoria en un procesador multinúcleo.

- Después de los registros, la memoria caché L1 es un espacio de pequeño de memoria, de apenas 16KB, la memoria caché L1 está usualmente localizada dentro del procesador, y es utilizada para capturar los bytes de instrucciones o datos usados frecuentemente.

- El nivel de caché L2 es un área de memoria mayor que el nivel L1, siendo en cambio más lenta. Su tamaño ronda entre los 256 KB. El espacio de memoria L2 puede almacenar fragmentos de las instrucciones más utilizadas. También almacena bytes con direcciones cercanas a los bytes almacenados en el nivel L1.

- La memoria caché L3 es un nivel añadido a la arquitectura para intercambiar datos entre unidades de procesamiento o núcleos. A pesar de ser más lenta que las anteriores, posee capacidades del orden de 6-8MB.

- La memoria RAM o memoria principal se encuentra fuera del procesador, su velocidad es muy lenta comparada con la memoria caché. Los datos e instrucciones del programa a ser ejecutados se encuentran aquí y después pasan por la memoria caché. A diferencia de la memoria caché, la memoria RAM puede ser expandida.

3.5.3. Cómputo distribuido

Un sistema de cómputo distribuido, también conocido como multiprocesador con memoria distribuida, es un sistema en el cual los elementos de procesamiento están interconectados ya sea vía red o vía punto a punto. Cada computadora posee su propia memoria y recursos para ejecutar un

programa. Aunque no existe una definición única para un sistema distribuido, es común encontrar las siguientes características:

- Consiste en un conjunto de computadoras independientes, con memoria propia.
- Las entidades de procesamiento pueden comunicarse mediante el paso de mensajes.
- El objetivo principal del sistema consiste en resolver un problema computacional grande.

Un sistema distribuido puede ser fácilmente expandido al agregar más unidades de procesamiento, a diferencia de los sistemas multinúcleo donde los núcleos se encuentran empotrados dentro del circuito integrado.

Resumen

En este capítulo se introdujeron aspectos relacionados al cómputo paralelo. A modo de cuantificar el grado de paralelismo obtenido en un programa se utiliza el concepto de aceleración.

La ley de Amdahl y la ley de Gustafson-Barsis modelan el comportamiento de un sistema paralelo al aumentar el número de unidades paralelas con respecto a la aceleración obtenida.

Actualmente, existen tres paradigmas de programación en paralelo, los cuales pueden ser combinados para obtener un mayor rendimiento de la aplicación. El paralelismo a nivel de instrucción consiste en procesar instrucciones independientes en paralelo. A pesar de que esta tarea suena trivial, aparecen problemas desafiantes cuando existen dependencias de datos entre instrucciones. La predicción de ramificaciones y la ejecución fuera de orden solventan esta problemática.

El paralelismo a nivel de datos, permite particionar la entrada de un problema, logrando que se ejecute el mismo procesamiento sobre diferentes datos. Arquitecturas recientes poseen conjuntos de instrucciones SIMD, dentro de las que destaca el juego de instrucciones SSE de Intel®.

La última clase de paralelismo es el paralelismo a nivel de tareas, el cual provee a las arquitecturas de unidades de procesamiento parcialmente independientes, y en algunos casos totalmente independientes (como en el caso del cómputo distribuido). Las arquitecturas multinúcleo han surgido recientemente en procesadores convencionales que están al alcance del usuario promedio, estas arquitecturas son atractivas para la implementación eficiente de programas, sin la necesidad de una inversión grande en infraestructura.

4

Aritmética de curvas elípticas

No hay nada como volver a un lugar que permanece sin cambios para descubrir cómo has cambiado tú.

Nelson Mandela

En este capítulo se revisan los conceptos básicos del grupo formado por los puntos de una curva elíptica. Se revisa la aritmética de curvas elípticas, se introduce el concepto de multiplicación escalar, así como también los algoritmos más conocidos para su implementación eficiente.

Se describen las curvas elípticas de Koblitz y algunas de las propiedades básicas con las que cuenta, una de las más importantes, es el endomorfismo de Frobenius, el cual es utilizado para realizar la multiplicación escalar sin el uso de doblado de puntos.

Para desarrollar la multiplicación de tal manera, es necesario desarrollar expansiones $\omega\tau$-NAF. Finalmente, se exponen los algoritmos para el cálculo de la multiplicación escalar utilizando el endomorfismo de Frobenius.

SECCIÓN 4.1

Curvas elípticas

4.1.1. Introducción a las curvas elípticas

Una curva elíptica E sobre un campo K está definida por la ecuación de Weierstrass:

$$E: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{4.1}$$

donde $a_1, a_2, a_3, a_4, a_6 \in K$ y $\Delta \neq 0$, Δ es el discriminante de E definido como sigue:

$$\begin{aligned}
\Delta &= -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\
d_2 &= a_1^2 + 4a_2 \\
d_4 &= 2a_4 + a_1a_3 \\
d_6 &= a_3^2 + 4a_6 \\
d_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2
\end{aligned}$$

Algunas veces se escribe E/K, o bien $E(K)$ para enfatizar que los puntos de la curva recaen sobre el campo K.

Dos curvas E_1 y E_2 sobre K son isomórfas si existen $u, r, s, t \in K$ y $u \neq 0$ tal que el cambio de variables:

$$(x, y) \rightarrow (u^2x + r, u^3y + u^2sx + t)$$

transforma a la curva E_1 en la curva E_2. Si tomamos la ecuación (4.1) de la curva elíptica sobre el campo binario $\mathbb{F}_{2^m}$, y se realiza el siguiente un cambio admisible de variables:

$$(x, y) \rightarrow \left(a_1^2x + \frac{a_3}{a_1}, a_1^3y + \frac{a_1^2a_4 + a_3^2}{a_1^3}\right)$$

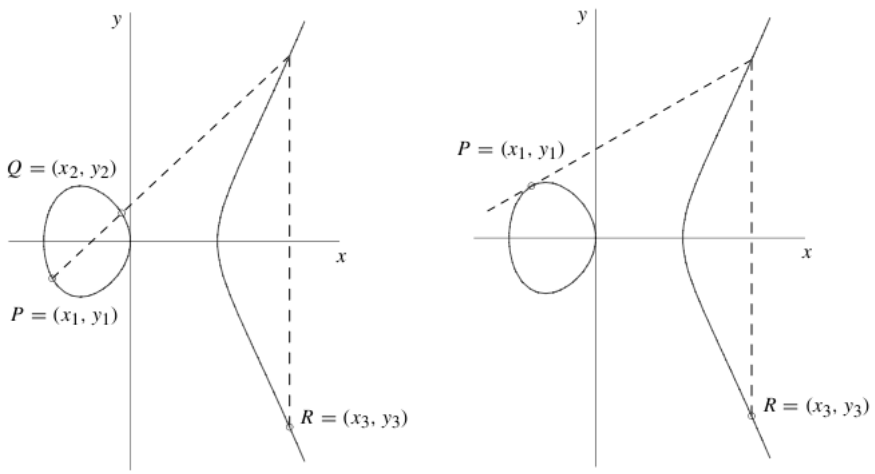
se obtiene una curva isomórfa a la de la ecuación (4.1):

$$y^2 + xy = x^3 + ax^2 + b \tag{4.2}$$

donde $a, b \in \mathbb{F}_{2^m}$ y $\Delta = b$. A la ecuación (4.2) se le conoce como la ecuación simplificada de Weierstrass permitiendo mayor flexibilidad al establecer la curva con sólo dos parámetros.

4.1.2. Ley de grupo

El conjunto de puntos que satisfacen la ecuación de la curva $E(\mathbb{F}_{2^m})$ forman un grupo aditivo abeliano, con el punto al infinito (O) como elemento identidad. Este grupo puede ser utilizado para la construcción de sistemas criptográficos.



(a) Suma de puntos (b) Doblado de puntos

Figura 4.1: Ley de grupo en la curva $E(\mathbb{R})$.

Para describir la suma de puntos comúnmente se recurre a su interpretación geométrica en el campo de los reales, $\mathbb{R}$, de la siguiente manera: sean P y Q dos puntos escogidos arbitrariamente en la curva E, trácese una línea recta secante que pase por P y Q, esta línea intersecará a la curva elíptica en un tercer punto R'. Entonces, se obtiene el punto $R = P + Q$ al reflejar el punto R' sobre el eje de las abscisas, (como se muestra en la figura 4.1(a)).

Un caso especial ocurre cuando $P = Q$, conocido como doblado de puntos. En este escenario se traza una línea recta tangente a la curva elíptica en el punto P, la línea recta intersecará a la curva en un punto R', para obtener $R = 2P$ se refleja el punto R' sobre el eje de las abscisas, (como se puede observar en la figura 4.1(b)).

La ecuación (4.3) nos permite calcular la suma de dos puntos P y Q con coordenadas (x_1, y_1) y (x_2, y_2), respectivamente. La ecuación (4.4) describe las operaciones necesarias para realizar el doblado de un punto con coordenadas (x_1, y_1). A pesar de que la deducción de estas ecuaciones es sencilla, se omite su derivación por razones de espacio, para ver los detalles precisos consúltese [80].

$$\lambda = \frac{y_1 + y_2}{x_1 + x_2} \qquad x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \qquad y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \tag{4.3}$$

$$\lambda = x_1 + \frac{y_1}{x_1} \qquad x_3 = \lambda^2 + \lambda + a \qquad y_3 = x_1^2 + \lambda x_3 + x_3 \tag{4.4}$$

Dado un punto $P = (x, y)$, el inverso de un punto, denotado como -*P*, tiene las coordenadas $(x, x+y)$.

Se puede observar que tanto en el cálculo del doblado como en la suma de puntos, ambas operaciones requieren el uso de una inversión en el campo. Si el campo es $\mathbb{F}_{2^m}$ la inversión es altamente costosa comparada con las demás operaciones de campo.

4.1.3. Orden del grupo

Sea E una curva elíptica sobre $\mathbb{F}_q$. El número de puntos que satisfacen a la ecuación de la curva, denotado por $\#E(\mathbb{F}_q)$, es llamado el orden de E. De acuerdo al Teorema del Intervalo de Hasse ([72]) este número se encuentra acotado por:

$$q + 1 - 2\sqrt{q} \leq \#E(\mathbb{F}_q) \leq q + 1 + 2\sqrt{q} \tag{4.5}$$

4.1.4. Representación de puntos

La representación de puntos en una curva elíptica queda determinada por un par ordenado $(x, y) \in \mathbb{F}_{2^m}$, y es conocida como representación del punto en coordenadas afines.

Sin embargo, la ley de grupo en coordenadas afines obliga a calcular inversos multiplicactivos en $\mathbb{F}^*_{2^m}$, esta operación aritmética es considerada cara. De manera alternativa, los puntos de una curva elíptica también pueden ser representados utilizando coordenadas proyectivas.

La representación de puntos mediante coordenadas proyectivas permite realizar operaciones de suma y doblado de puntos sin realizar inversiones de campo, sin embargo, se requieren 3 coordenadas para representar a un punto así como el aumento en el número de multiplicaciones de campo para realizar las operaciones. A continuación se introduce la representación de puntos en coordenadas proyectivas.

Coordenadas proyectivas. Sean c y d enteros positivos, se define una relación de equivalencia $\sim$ como el conjunto $\mathbb{F}^3_{2^m} \setminus \{0,0,0\}$ (tripletas sobre $\mathbb{F}_{2^m}$ diferentes a cero) tales que:

$$(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2) \text{ si y sólo si } X_1 = \lambda^c X_2, Y_1 = \lambda^d Y_2, Z_1 = \lambda Z_2, \tag{4.6}$$

para algún $\lambda \in \mathbb{F}^*_{2^m}$.

Si $Z \neq 0$ entonces $(X/Z^c, Y/Z^d, 1)$ es un elemento representativo de la clase de equivalencia $(X : Y : Z)$. De este modo dado un punto en coordenadas afines es fácil convertirlo a su versión proyectiva simplemente asignando a Z el valor de 1. Dado un punto en coordenadas proyectivas (X, Y, Z), para convertirlo a coordenadas afines se procede de la siguiente manera:

$$\left(\frac{X}{Z^c}, \frac{Y}{Z^d}\right) \rightarrow (x, y)$$

Si $Z = 0$, entonces al conjunto $\{(X,Y,0) : X,Y \in \mathbb{F}_{2^m}\}$ se le conoce como línea al infinito y corresponde a la versión proyectiva del punto al infinito en las coordenadas afines.

Coordenadas proyectivas López-Dahab (LD). Estas coordenadas propuestas en [59] asignan el valor de $c = 1$ y $d = 2$ en la ecuación (4.6), de esta forma el punto proyectivo $(X : Y : Z)$ corresponde al punto afín $(X/Z, Y/Z^2)$.

La ecuación proyectiva de la curva elíptica se obtiene al realizar el cambio de variables $(x,y) \rightarrow \left(\frac{X}{Z}, \frac{Y}{Z^2}\right)$ en la ecuación (4.2):

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4 \tag{4.7}$$

donde $a, b \in \mathbb{F}_{2^m}$.

Mientras que el punto al infinito O se representa por $(1 : 0 : 0)$, y el negativo del punto $(X : Y : Z)$ se calcula como $(X : XZ + Y : Z)$.

Se presentan las fórmulas para la suma de puntos (algoritmo 5) y el doblado de puntos (algoritmo 6) usando coordenadas proyectivas LD, siendo una característica notable el hecho de que estos algoritmos evitan el cómputo de inversiones de campo, al precio de un incremento en el número de multiplicaciones.

La suma mixta se define como la operación que toma un punto en coordenadas afines y un punto en coordenadas proyectivas y calcula la suma de ambos en coordenadas proyectivas. Esta operación involucra menos operaciones de campo, aprovechando que una de las coordenadas se encuentra fija como $(Z_1 = 1)$. En el algoritmo 7 se presentan las operaciones para calcular la suma mixta de puntos.

En la tabla 4.1, mostrada a continuación, se resume el costo de las operaciones de suma y doblado de puntos tomando como unidades la cantidad de multiplicaciones (M), elevaciones al cuadrado (S) e inversiones (I). Como se puede notar, la diferencia entre el costo computacional que toma el realizar un doblado comparado con una suma de puntos es de más del triple.

Sistema de coordenadas	Suma de puntos	Suma mixta	Doblado de puntos
Afines	1I + 1M + 2S	No aplica	1I + 1M + 1S
López-Dahab	14M + 7S	8M + 5S	4M + 5S

Tabla 4.1: Costo de realizar sumas y doblados de puntos en coordenadas afines y proyectivas LD.

Algoritmo 5 Suma de puntos en coordenadas proyectivas LD ([59])

Entrada: $P = (X_1, Y_1, Z_1)$ y $Q = (X_2, Y_2, Z_2)$

Salida: $R = P + Q = (X_3, Y_3, Z_3)$

1: $A_0 \leftarrow Y_1 \cdot Z_0^2$
2: $A_1 \leftarrow Y_0 \cdot Z_1^2$
3: $B_0 \leftarrow X_1 \cdot Z_0$
4: $B_1 \leftarrow X_0 \cdot Z_1$
5: $C \leftarrow A_0 + A_1$
6: $D \leftarrow B_0 + B_1$
7: $E \leftarrow Z_0 \cdot Z_1$
8: $F \leftarrow D \cdot E$
9: $Z_2 \leftarrow F^2$
10: $G \leftarrow D^2 \cdot (F + aE^2)$
11: $H \leftarrow C \cdot F$
12: $X_2 \leftarrow C^2 + H + G$
13: $I \leftarrow D^2 \cdot B_0 \cdot E + X_2$
14: $J \leftarrow D^2 \cdot A_0 + X_2$
15: $Y_2 \leftarrow H \cdot I + Z_2 \cdot J$
16: **return** (X_2, Y_2, Z_2)

Algoritmo 6 Doblado de puntos en coordenadas proyectivas LD ([40])

Entrada: $P = (X_1, Y_1, Z_1)$ y el parámetro $a \in \{0, 1\}$ de la curva.

Salida: $R = 2P = (X_3, Y_3, Z_3)$.

1: **if** $P = O$ **then**
2: **return** O
3: **end if**
4: $T_1 \leftarrow Z_1^2$
5: $T_2 \leftarrow X_1^2$
6: $Z_3 \leftarrow T_1 \cdot T_2$
7: $X_3 \leftarrow T_2^2$
8: $T_1 \leftarrow T_1^2$
9: $T_2 \leftarrow T_1 \cdot b$
10: $X_3 \leftarrow X_3 + T_2$
11: $T_1 \leftarrow Y_1^2$
12: **if** $a = 1$ **then**
13: $T_1 \leftarrow T_1 + Z_3$
14: **end if**
15: $T_1 \leftarrow T_1 + T_2$
16: $Y_3 \leftarrow X_3 \cdot T_1$
17: $T_1 \leftarrow T_2 \cdot Z_3$
18: $Y_3 \leftarrow Y_3 + T_1$
19: **return** (X_3, Y_3, Z_3)

Algoritmo 7 Suma mixta de puntos en coordenadas proyectivas LD ([40])

Entrada: $P = (X_1, Y_1, Z_1)$ y $Q = (x_2, x_2)$ tal que $a \in \{0, 1\}$

Salida: $R = P + Q = (X_3, Y_3, Z_3)$

1: **if** $Q = O$ **then**
2: **return** P
3: **end if**
4: **if** $P = O$ **then**
5: **return** $(x_2, y_2, 1)$
6: **end if**
7: $T_1 \leftarrow Z_1 \cdot x_2$
8: $T_2 \leftarrow Z_1^2$
9: $X_3 \leftarrow X_1 + T_1$
10: $T_1 \leftarrow Z_1 \cdot X_3$
11: $T_3 \leftarrow T_2 \cdot y_2$
12: $Y_3 \leftarrow Y_1 + T_3$
13: **if** $X_3 = 0$ **then**
14: **if** $Y_3 = 0$ **then**
15: **return** $2(x_2, y_2, 1)$
{Usando algoritmo 6.}
16: **else**
17: **return** O
18: **end if**
19: **end if**
20: $Z_3 \leftarrow T_1^2$
21: $T_3 \leftarrow T_1 \cdot Y_3$
22: **if** $a = 1$ **then**
23: $T_1 \leftarrow T_1 + T_2$
24: **end if**
25: $T_2 \leftarrow X_3^2$
26: $X_3 \leftarrow T_2 \cdot T_1$
27: $T_2 \leftarrow Y_3^2$
28: $X_3 \leftarrow X_3 + T_2$
29: $X_3 \leftarrow X_3 + T_3$
30: $T_2 \leftarrow x_2 \cdot Z_3$
31: $T_1 \leftarrow Z_3^2$
32: $T_3 \leftarrow T_3 + Z_3$
33: $Y_3 \leftarrow T_3 \cdot T_2$
34: $T_2 \leftarrow x_2 + y_2$
35: $T_3 \leftarrow T_1 \cdot T_2$
36: $Y_3 \leftarrow Y_3 + T_3$
37: **return** (X_3, Y_3, Z_3)

SECCIÓN 4.2

Multiplicación escalar

En esta sección se describe una operación sobre los puntos de una curva elíptica, conocida como multiplicación escalar. Esta operación toma un punto en la curva elíptica P y un escalar $k \in \mathbb{Z}$ y calcula un punto Q obtenido mediante la aplicación de k veces el operador de grupo $+$ sobre el punto P consigo mismo, la operación se denota como $Q = [k]P$:

$$Q = [k]P = \underbrace{P + P + \cdots + P}_{k \text{ veces}} \tag{4.8}$$

Se cumple que para todos los puntos en la curva elíptica $P \in E(\mathbb{F}_{2^m})$, si $d = \#E(\mathbb{F}_{2^m})$ entonces $[d]P = O$. El orden de un punto es el entero r más pequeño tal que $[r]P = O$ y $r|\#E(\mathbb{F}_{2^m})$. De esta forma el escalar k puede ser reducido al conjunto $\mathbb{Z}_r$.

A continuación se describen algoritmos básicos para el cálculo de la multiplicación escalar, utilizando las operaciones de suma y doblado de puntos.

4.2.1. Multiplicación escalar de izquierda a derecha

La forma más intuitiva de realizar la multiplicación escalar $[k]P$ consiste en representar a un escalar k de n bits en base 2, esta representación puede ser vista como un polinomio de variable x:

$$K(x) = \sum_{i=0}^{n-1} k_i x^i$$

tal que $k = K(2)$.

La regla de Horner es un método eficiente para la evaluación de polinomios con complejidad $O(n)$, donde n es el grado del polinomio que se evalua.

La evaluación realiza incondiconalmente doblados de puntos sobre un acumulador Q y se examinan los bits desde la potencia más significativa hacia la menos significativa, para determinar si en este paso o iteración es necesario sumar el punto P al acumulador Q.

$$\begin{aligned}
[k(2)]P &= [k_{n-1}2^{n-1}]P + [k_{n-2}2^{n-2}]P + \cdots + [k_2 2^2]P + [k_1 2]P + [k_0]P \\
&= 2\left([k_{n-1}2^{n-2}]P + [k_{n-2}2^{n-3}]P + \cdots + [k_2 2]P + [k_1]P\right) + [k_0]P \\
&\vdots \\
&= 2(2(\ldots 2(2P + [k_{n-2}]P) + [k_{n-3}]P) + \cdots + [k_1]P) + [k_0]P
\end{aligned}$$

El algoritmo 8 lista las operaciones necesarias para el cálculo de la multiplicación escalar utilizando la regla de Horner. Por la forma en que las potencias del polinomio son evaluadas, a este método se le llama multiplicación escalar de izquierda a derecha.

Algoritmo 8 Multiplicación escalar de izquierda a derecha.

Entrada: $k \in \mathbb{Z}_r, P \in E$ de orden r.

Salida: $Q \in E$ tal que $Q = [k]P$.

1: $Q \leftarrow O$
2: **for** $i = n-1$ **downto** 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** Q

4.2.2. Multiplicación escalar de derecha a izquierda

A continuación se explicará la idea principal del funcionamiento del algoritmo de derecha a izquierda para el método de doblado y suma. Sea k un escalar de n bits, tal que:

$$\begin{aligned} [k]P &= \sum_{i=0}^{n-1} [k_i 2^i]P \\ &= [k_{n-1}2^{n-1}]P + [k_{n-2}2^{n-2}]P + \cdots + [k_2 2^2]P + [k_1 2^1]P + [k_0 2^0]P \end{aligned}$$

y $k_i \in \{0,1\}$. De esta manera se construye un algoritmo iterativo que utiliza un acumulador inicializado en $Q \leftarrow O$ y un punto $R \leftarrow P$.

En la iteración i-ésima se suma en el acumulador Q el múltiplo $R = [2^i]P$ si $k_i = 1$, por el contrario si $k_i = 0$ la suma no se realiza. En cada iteración el punto R se actualiza mediante $R \leftarrow [2]R$.

El algoritmo 9 describe el proceso de multiplicación escalar de derecha a izquierda.

Algoritmo 9 Multiplicación escalar de derecha a izquierda.

Entrada: $k \in \mathbb{Z}_r, P \in E$ de orden r.

Salida: $Q \in E$ tal que $Q = [k]P$.

1: $Q \leftarrow O$
2: $R \leftarrow P$
3: **for** $i = 0$ **to** $n-1$ **do**
4: **if** $k_i = 1$ **then**
5: $Q \leftarrow Q + R$
6: **end if**
7: $R \leftarrow 2R$
8: **end for**
9: **return** Q

SECCIÓN 4.3

Curvas elípticas de Koblitz

También conocidas como curvas binarias anómalas (ABC, del inglés *Anomalous Binary Curves*), las curvas E_a están definidas por la ecuación:

$$E_a : y^2 + xy = x^3 + ax^2 + 1, \quad a \in \{0,1\} \tag{4.9}$$

Son llamadas curvas de Koblitz puesto que en [54] se demostró por primera vez que el cálculo de la multiplicación escalar se puede realizar de forma eficiente reemplazando los doblados de puntos por aplicaciones del operador Frobenius.

El conjunto de puntos que cumplen con la ecuación de la curva de Koblitz E_a en el campo $\mathbb{F}_2$ junto con el punto al infinito O como elemento identidad, forma un grupo abeliano aditivo, denotado como $E_a(\mathbb{F}_2)$:

$$\begin{aligned} E_1(\mathbb{F}_2) &= \{(0,1), O\} \\ E_0(\mathbb{F}_2) &= \{(0,1),(1,0),(1,1), O\} \end{aligned}$$

Sea $f = \#E_a(\mathbb{F}_2)$, f toma el valor de 2 si $a = 1$ y de 4 si $a = 0$, en general:

$$f = 2^{2-a} \tag{4.10}$$

4.3.1. Propiedades básicas

Curvas definidas sobre extensiones. Denotamos con $E_a(\mathbb{F}_{2^m})$ al grupo de puntos de la curva elíptica E_a definidos en la extensión m-ésima $\mathbb{F}_{2^m}$ del campo $\mathbb{F}_2$.

La extensión de este campo debe ser cuidadosamente elegida de tal forma que sea difícil calcular el logaritmo discreto de sus elementos. Si m es primo, entonces $\#E_a(\mathbb{F}_{2^m}) = f \cdot r$, con r un número primo grande, y éste es un caso de interés criptográfico.

El subgrupo principal. Si $\#E_a(\mathbb{F}_{2^m}) = f \cdot r$, definimos el subgrupo principal como el subgrupo de orden r. Es común que muchas de las operaciones criptográficas se realicen en el subgrupo principal.

En [53], se da el valor del orden del grupo, el cual puede ser calculado mediante la siguiente identidad:

$$\#E_a(\mathbb{F}_{2^m}) = 2^m + 1 - V_m \tag{4.11}$$

donde V_m es un término de la secuencia de Lucas definido en la sección 2.1 de la página 10. Tomando la ecuación (4.10), el orden del subgrupo principal es:

$$r = 2^{m+a-2} - 2^{a-2}(V_m - 1) \tag{4.12}$$

4.3.2. Endomorfismo de Frobenius

Sea $E_a(\mathbb{F}_2)$ el grupo formado por los puntos de una curva elíptica de Koblitz. El endomorfismo de Frobenius, denotado por τ, está definido como sigue:

$$\begin{aligned} \tau\colon E_a(\mathbb{F}_2) &\rightarrow E_a(\mathbb{F}_2) \\ (x,y) &\rightarrow (x^2, y^2) \\ O &\rightarrow O \end{aligned} \tag{4.13}$$

De la ecuación de suma de puntos, se puede constatar que para cualquier $(x,y) \in E_a(\mathbb{F}_2)$, se cumple la siguiente ecuación:

$$(x^4, y^4) + 2(x,y) = \mu \cdot (x^2, y^2)$$

donde $\mu = (-1)^{1-a}$. Reescribiendo en términos del endomorfismo de Frobenius, τ se tiene que $\forall P \in E_a(\mathbb{F}_2)$ se cumple:

$$\begin{aligned} \tau(\tau(P)) + 2P &= \mu\tau(P) \\ (\tau^2 + 2)P &= \mu\tau(P) \end{aligned}$$

es decir:

$$\tau^2 + 2 = \mu\tau, \quad \text{donde } \mu = (-1)^{1-a} \tag{4.14}$$

A la ecuación (4.14) se conoce como ecuación característica del endormorfismo de Frobenius. Con èsta es posible reemplazar la operación de doblado de puntos por aplicaciones del endomorfismo de Frobenius, cuyo costo computacional es notablemente menor, permitiendo acelerar en gran medida el costo computacional de la multiplicación escalar.

SECCIÓN 4.4

Trabajos previos en curvas de Koblitz

A continuación se resumen las principales aportaciones que a lo largo de los años se han reportado para el cálculo eficiente de la multiplicación escalar en curvas de Koblitz tomando ventaja del endomorfismo de Frobenius.

Primera aproximación. Koblitz en [54] propuso transformar el escalar k a base t donde t es una raíz de la ecuación característica del endomorfismo de Frobenius. Si se toma el dominio euclidiano $\mathbb{Z}[\tau]$ con la función norma N definida como:

$$\begin{aligned} N: \quad \mathbb{Z}[\tau] &\rightarrow \mathbb{N}\cup\{0\} \\ d_0+d_1\tau &\mapsto d_0^2+\mu d_0 d_1+2d_1^2 \end{aligned} \tag{4.15}$$

entonces cualquier elemento en $\mathbb{Z}[\tau]$ tiene factorización única de la forma $k=\sum \varepsilon_i \tau^i$, donde $\varepsilon_i \in \{0,1\}$, permitiendo realizar la multiplicación escalar con el uso de aplicaciones del endomorfismo de Frobenius, al realizar $[k]P=\sum \varepsilon_i \tau^i P$.

Koblitz notó que la longitud de las expansiones τ es de aproximadamente el doble de los bits de la representación binaria de k. Lo cual aumenta en promedio al doble el número de sumas de punto.

Expansiones equivalentes. Meier y Staffelbach en [61], solucionaron el problema planteado por Koblitz, al notar que dado un punto $P \in E_a(\mathbb{F}_{2^m})$, se cumple que:

$$P=(x,y)=(x^{2^m},y^{2^m})=\tau^m P$$

$$O=P-P=\tau^m P-P=(\tau^m-1)P$$

Dados dos elementos $\gamma,\rho \in \mathbb{Z}[\tau]$, existe una relación de equivalencia $\gamma \equiv \rho \pmod{\tau^m-1}$, es decir existe un elemento $\kappa \in \mathbb{Z}[\tau]$ para el cual:

$$\gamma P=\rho P+\kappa(\tau^m-1)P=\rho P+\kappa O=\rho P$$

De esta forma cualquier elemento γ puede ser reducido (mod τ^m-1) y obtener un elemento $\gamma \equiv \rho$ de longitud menor.

El subgrupo principal. Es común que las operaciones criptográficas sean realizadas en el subgrupo principal, es decir, utilizando los puntos cuyo orden es un número primo grande, denotado por r. Solinas excluye a los puntos del subgrupo más pequeño al reducir $(\mod \delta)$ donde δ se define como:

$$\delta = \frac{\tau^m - 1}{\tau - 1}. \tag{4.16}$$

Al realizar esto la longitud de las expansiones es a lo más $m+a$, donde a es la constante de la curva E_a de la ecuación (4.9) como se detalla en [73].

Para dividir en el anillo $\mathbb{Z}[\tau]$, Solinas desarrolló un algoritmo en el que dados (γ, δ) se obtiene (ρ, κ) tal que $\gamma = \kappa\delta + \rho$, y ρ de norma mínima. El algoritmo 10 detalla la forma de obtener el cociente y el residuo dos elementos en $\mathbb{Z}[\tau]$.

Algoritmo 10 División en el anillo $\mathbb{Z}[\tau]$

Entrada: Un dividendo $\gamma = c_0 + c_1\tau$ y un divisor $\delta = d_0 + d_1\tau$
Salida: Un cociente $\kappa = q_0 + q_1\tau$ y un residuo $\rho = r_0 + r_1\tau$

1: $g_0 \leftarrow c_0 d_0 + \mu c_0 d_1 + 2c_1 d_1$
2: $g_1 \leftarrow c_1 d_0 - c_0 d_1$
3: $N \leftarrow d_0^2 + \mu d_0 d_1 + 2d_1^2$
4: $\lambda_0 \leftarrow g_0/N$
5: $\lambda_1 \leftarrow g_1/N$
6: $(q_0, q_1) \leftarrow \text{Redondeo}(\lambda_0, \lambda_1)$ {Utilizando el algoritmo 11}
7: $r_0 \leftarrow c_0 - d_0 q_0 - 2s_1 q_1$
8: $r_1 \leftarrow c_1 - d_1 q_0 - d_0 q_1 - \mu d_1 q_1$
9: **return** $(q_0, q_1), (r_0, r_1)$

Utilizando NAF. Solinas en [73] introdujo el concepto de expansión τ-NAF, dicha transformación obtiene una expansión $\sum \varepsilon_i \tau^i$ con coeficientes $\varepsilon_i \in \{-1, 0, 1\}$; con la propiedad de que dados dos coeficientes consecutivos a lo más uno de ellos es diferente a cero, logrando así que en promedio, el peso Hamming de la expansión τ-NAF sea de $\frac{1}{3}$.

En el mismo artículo se generaliza el concepto a $\omega\tau$-NAF. En dicha representación si se toman ω coeficientes consecutivos en la expansión, entonces a lo más uno de ellos es diferente de cero. Es claramente verificable que τ-NAF es equivalente a 2τ-NAF. La longitud de las expansiones $\omega\tau$-NAF es a lo más $m+a+3$ como se describe en [73].

Algoritmo 11 Redondeo de un elemento $\lambda \in \mathbb{Z}[\tau]$

Entrada: $\lambda_0, \lambda_1 \in \mathbb{R}$ tal que $\lambda_0 + \lambda_1\tau$

Salida: $q_0, q_1 \in \mathbb{Z}$ tal que $q_0 + q_1\tau$

1: $f_i \leftarrow \text{Redondeo}(\lambda_i)|_{i\in\{0,1\}}$ {Usando la ecuación (4.20)}
2: $\eta_i \leftarrow \lambda_i - f_i|_{i\in\{0,1\}}$
3: $h_i \leftarrow 0|_{i\in\{0,1\}}$
4: $\eta \leftarrow 2\eta_0 + \mu\eta_1$
5: **if** $\eta \geq 1$ **then**
6: **if** $\eta_0 - 3\mu\eta_1 < -1$ **then**
7: $h_1 \leftarrow \mu$
8: **else**
9: $h_0 \leftarrow 1$
10: **end if**
11: **else**
12: **if** $\eta_0 + 4\mu\eta_1 \geq 2$ **then**
13: $h_1 \leftarrow \mu$
14: **end if**
15: **end if**
16: **if** $\eta < -1$ **then**
17: **if** $\eta_0 - 3\mu\eta_1 \geq 1$ **then**
18: $h_1 \leftarrow -\mu$
19: **else**
20: $h_0 \leftarrow -1$
21: **end if**
22: **else**
23: **if** $\eta_0 + 4\mu\eta_1 < -2$ **then**
24: $h_1 \leftarrow -\mu$
25: **end if**
26: **end if**
27: $q_i \leftarrow f_i + h_i|_{i\in\{0,1\}}$
28: **return** (q_0, q_1)

Solinas 1997 contra Solinas 2000. Tres años después de dar a conocer los algoritmos en [73], Jerome Solinas hizo público un segundo artículo [74]. En ambos promueve el uso de expansiones ωτ-NAF, no obstante en el segundo utiliza diferentes representantes de congruencia. He aquí una breve explicación de las diferencias en estas dos versiones:

- En [73] los representantes son números en el conjunto $\{0, \pm 1, \pm 3, \ldots, \pm 2^{\omega-1} - 1\}$. La longitud de las expansiones ωτ-NAF es mayor a $(m + a + 3)$ al verificarse experimentalmente con enteros aleatorios en [7].
- En [74] se modifican los representantes de congruencia, los cuales son ahora términos $\{\pm \alpha_u\}$ para $u \in \{1, 3, \ldots, 2^{\omega-1} - 1\}$. La longitud de las expansiones ωτ-NAF es a lo más $(m + a + 3)$ como se demostró en ese documento.

SECCIÓN 4.5

Cómputo de expansiones ωτ-NAF

Para poder hacer el reemplazo de los doblados de puntos es necesario convertir al escalar k a una expansión ωτ-NAF. En esta sección se describen brevemente los algoritmos utilizados, la mayoría de ellos son explicados extensamente en [74].

4.5.1. Reducción parcial

La transformación que dado un escalar $k \in \mathbb{Z}$ obtiene un elemento $\rho \in \mathbb{Z}[\tau]$ se denota como:

$$\rho \leftarrow k \text{ partmod } \delta \tag{4.17}$$

para $\delta = \frac{\tau^m - 1}{\tau - 1}$. El elemento $\rho = \rho_0 + \rho_1 \tau$ es la representación canónica del escalar k en el anillo $\mathbb{Z}[\tau]$. Para obtener el elemento ρ se utiliza el algoritmo de división en $\mathbb{Z}[\tau]$ desarrollado por Solinas (algoritmo 10), tomando en cuenta que:

$$N(\delta) = \#E_a(\mathbb{F}_{2^m})/f = r \tag{4.18}$$

El algoritmo 12 enlista los pasos necesarios para calcular la reducción parcial. Las constantes s_0 y s_1 se encuentran definidas como sigue:

$$s_i = \frac{(-1)^i}{f}(1 - \mu U_{m+3-a-i})|_{i \in \{0,1\}} \tag{4.19}$$

donde $U_{m+3-a-1}$ es un término de la secuencia de Lucas definida en la página 10. Las constantes s_i están completamente determinadas al momento de elegir la extensión de campo así como la curva elíptica de Koblitz.

Algoritmo 12 Reducción parcial modulo $\delta = \frac{\tau^m - 1}{\tau - 1}$

Constantes: s_0, s_1 definidas por la ecuación (4.19).

Entrada: $k \in \mathbb{Z}$

Salida: (ρ_0, ρ_1) tal que $\rho_0 + \rho_1\tau \leftarrow k$ partmod δ

1: $d_0 \leftarrow s_0 + \mu s_1$
2: $\lambda_0 \leftarrow s_0 k / r$
3: $\lambda_1 \leftarrow s_1 k / r$
4: $(q_0, q_1) \leftarrow$ Redondeo(λ_0, λ_1) {Utilizando el algoritmo 11.}
5: $\rho_0 \leftarrow k - d_0 q_0 - 2 s_1 q_1$
6: $\rho_1 \leftarrow s_1 q_0 - s_0 q_1$
7: **return** (ρ_0, ρ_1)

División aproximada. Nótese que el cálculo de (λ_0, λ_1) en el algoritmo 12 involucra una división por r y una multiplicación de enteros de $\frac{m}{2}$ bits, sin embargo este término se puede calcular de forma eficiente utilizando el algoritmo 13.

Al algoritmo 13 se le conoce como división aproximada porque calcula un elemento $\rho' \in \mathbb{Z}[\tau]$ tal que $\rho' \equiv k \bmod \delta$, pero no necesariamente de norma mínima. Se denota con C el número de bits de precisión de la aproximación, es decir, $\Pr[\rho' = \rho] < 2^{-(C-5)}$.

En el algoritmo 13 la constante V_m es un término de la secuencia de Lucas definido en la sección 2.1. La función Redondeo: $\mathbb{R} \to \mathbb{Z}$ está definida como sigue:

$$\text{Redondeo}: x \to \left\lfloor x + \frac{1}{2} \right\rfloor \tag{4.20}$$

4.5.2. Expansión ωτ-NAF

Las expansiones ωτ-NAF son análogas a las expansiones ordinarias ω-NAF. Una expansión ωτ-NAF es un polinomio con variable τ y de grado $l \leq m + a + 3$, que cumple la propiedad de no adyacencia (NAF), es decir, para ω coeficientes consecutivos a lo más uno de ellos es diferente a cero. El número de coeficientes diferentes a cero en una expansión ωτ-NAF es entonces en promedio $\frac{l}{\omega+1}$.

Haciendo un breve resumen, en el método para calcular la expansión ω-NAF de un entero n, en cada iteración se examinan los ω bits menos significativos de n con el objetivo de conocer a cual clase impar de equivalencia pertenece, es decir $n \equiv u \pmod{2^\omega}$. Después se actualiza $n \leftarrow \frac{n-u}{2}$, lo cual permite que n sea divisible por 2^ω, asegurando que los siguientes ω bits sean igual a 0; este proceso se repite hasta que $n = 0$.

Algoritmo 13 División aproximada por r con C bits de precisión.

Constantes: $C, K \leftarrow \frac{m+5}{2} + C, V_m$.

Entrada: $k, s_i|_{i\in 0,1}$.

Salida: $\lambda_i \leftarrow s_i k/r$ con C bits de precisión.

1: $k' \leftarrow \left\lfloor \frac{k}{2^{m-K-2+a}} \right\rfloor$
2: $g \leftarrow s_i k'$
3: $h \leftarrow \left\lfloor \frac{g}{2^m} \right\rfloor$
4: $j \leftarrow V_m h$
5: $l \leftarrow$ Redondeo $\left(\frac{g+j}{2^{K-C}} \right)$ {Usando la ecuación (4.20).}
6: **return** $\lambda_i \leftarrow \frac{l}{2^C}$

Para calcular la representación ωτ-NAF de un elemento $\rho_0 + \rho_1\tau \in \mathbb{Z}[\tau]$, se debe examinar a cuál clase de equivalencia $(\mathrm{mod}\ \tau^\omega)$ pertenece el elemento ρ, para determinar lo anterior Solinas en [73] introduce el uso de ϕ_ω, el cual es un homomorfismo de anillos definido de la siguiente manera:

$$\begin{aligned} \phi_\omega: \quad \mathbb{Z}[\tau] &\rightarrow \mathbb{Z}/2^\omega\mathbb{Z} \\ u_0 + u_1\tau &\rightarrow u_0 + u_1 t_\omega \end{aligned} \tag{4.21}$$

donde el término t_ω se define como:

$$t_\omega = 2U_{\omega-1}U_\omega^{-1} \pmod{2^\omega} \tag{4.22}$$

donde los factores U_ω y $U_{\omega-1}$ son términos de la secuencia de Lucas definidos en la sección 2.1.

El kernel de ϕ_ω consiste de todos los elementos $\alpha \in \mathbb{Z}[\tau]$ tales que $\phi_\omega(\alpha) = 0$ si y sólo si α es divisible por τ^ω. Cada clase de equivalencia $(\mathrm{mod}\ \tau^\omega)$ impar corresponde bajo ϕ_ω con las clases de equivalencia $(\mathrm{mod}\ 2^\omega)$ impares. Usando este homomorfismo se examinan los ω bits menos significativos de $\phi_\omega(\rho)$, para determinar la clase de equivalencia $(\mathrm{mod}\ 2^\omega)$ a la que pertenece.

Los representantes principales de las clases de equivalencias se denotan por el conjunto $\{\alpha_u\}$ definido como sigue:

$$\{\alpha_u\} = \{\beta_u + \gamma_u\tau\} = \{u \bmod \tau^\omega \,|\, u \in I\} \qquad \tau^\omega = U_\omega\tau - 2U_{\omega-1} \tag{4.23}$$

$$I = \{1, 3, 5, \ldots, 2^{\omega-1} - 1\} \tag{4.24}$$

donde U_ω y $U_{\omega-1}$ son términos de la secuencia de Lucas. Los representantes $\{\alpha_u\}$ son calculados mediante el algoritmo de división en $\mathbb{Z}[\tau]$ (algoritmo 10).

El algoritmo 14 calcula la expansión ωτ-NAF de un elemento $\rho \in \mathbb{Z}[\tau]$. A dicha expansión la denotaremos como $\sum_{i=0}^{l-1} u_i\tau^i$ donde $u_i \in \{\alpha_u\}_{u\in I}$ y $l \leq m + a + 3$. [74].

Algoritmo 14 $\omega\tau$-NAF de un elemento $\rho \in \mathbb{Z}[\tau]$

Constantes: ω, μ, t_w de la ecuación (4.22), $\{\alpha_u = \beta_u + \gamma_u\tau\}_{u \in I}$ de la ecuación (4.23).

Entrada: $\rho = \rho_0 + \rho_1\tau \in \mathbb{Z}[\tau]$.

Salida: $\sum_{i=0}^{l-1} u_i\tau^i \leftarrow \omega\tau\text{-NAF}(\rho)$ tal que $l \leq m + a + 3$.

1: $i \leftarrow 0$
2: **while** $\rho_0 \neq 0$ **or** $\rho_1 \neq 0$ **do**
3: **if** ρ_0 es impar **then**
4: $u \leftarrow \rho_0 + \rho_1 t_\omega \pmod{2^\omega}$
5: **if** $u > 0$ **then**
6: $\xi \leftarrow 1$
7: **else**
8: $\xi \leftarrow -1$, $u \leftarrow -u$
9: **end if**
10: $(\rho_0, \rho_1) \leftarrow (\rho_0 - \xi\beta_u, \rho_1 - \xi\gamma_u)$
11: $u_i \leftarrow \xi\alpha_u$
12: **else**
13: $u_i \leftarrow 0$
14: **end if**
15: $i \leftarrow i + 1$
16: $(\rho_0, \rho_1) \leftarrow (\rho_1 + \mu\rho_0/2, -\rho_0/2)$
17: **end while**
18: **return** $(u_{i-1}, u_{i-2}, \ldots, u_1, u_0)$

4.5.3. Expansiones ωτ-NAF aleatorias

En [54], Koblitz acreditó a Lenstra la idea de producir aleatoriamente una expansión τ-NAF, en lugar de obtenerla a partir de un entero con los algoritmos antes mencionados. En ese documento, se propone la generación de expansiones τ-NAF aleatorias utilizando un generador con la siguiente distribución de probabilidad:

$$e = \begin{cases} 0 & \Pr[e=0] = \frac{1}{2} \\ 1 & \Pr[e=1] = \frac{1}{4} \\ -1 & \Pr[e=-1] = \frac{1}{4} \end{cases} \tag{4.25}$$

Para generar una expansión τ-NAF, se toma un coeficiente del generador, y por cada coeficiente $e_i \neq 0$, el siguiente coeficiente se fija a $e_{i+1} = 0$.

En [56], Lange y Shparlinski demostraron que las expansiones aleatorias, de longitud $l = m-1$, están bien distribuidas sobre el campo $\mathbb{Z}_r$, donde r es el orden del grupo $E_a(\mathbb{F}_{2^m})$.

En el caso de expansiones ωτ-NAF aleatorias, se construye un generador análogo al mostrado en el ecuación (4.25), con la siguiente distribución de probabilidad:

$$e = \begin{cases} 0 & \Pr[e=0] = \frac{1}{2} \\ \zeta & \Pr[e=\zeta] = \frac{1}{2^{\omega}} \end{cases} \tag{4.26}$$

el cual permitirá seleccionar uno de los representantes de congruencia, $\zeta \in \{\pm\alpha_u\}$.

4.5.4. Recuperación del entero equivalente

Al proceso que dada una expansión $\sum_{i=0}^{l-1} u_i\tau^i$ obtiene un número entero $k \in \mathbb{Z}_r$ tal que

$$\sum_{i=0}^{l-1} u_i\tau^i \leftarrow \omega\tau\text{-NAF}(k \operatorname{partmod} \delta) \tag{4.27}$$

se le conoce como la recuperación del entero equivalente. Este algoritmo es el inverso a la obtención de expansiones ωτ-NAF.

En [55], Lange propuso un algoritmo que recupera el entero equivalente de una expansión τ-NAF, utilizando $l-2$ multiplicaciones y sumas sobre el campo $\mathbb{Z}_r$. Lange también demostró que dado un punto generador, $P \in E_a(\mathbb{F}_{2^m})$ de orden primo r, y un endomorfismo τ, existe un entero único $s \in \mathbb{Z}_r$ tal que:

$$\tau(P) = [s]P \tag{4.28}$$

donde s se encuentra determinado al momento de seleccionar la extensión del campo donde la curva está definida, el valor de s puede ser obtenido mediante:

$$(\tau - s) = \gcd\left(\frac{\tau^m - 1}{\tau - 1}, \tau^2 - \mu\tau + 2\right) \tag{4.29}$$

En [14], se presentó un algoritmo que realiza la recuperación del entero equivalente utilizando la siguiente ecuación:

$$\tau^i = U_i\tau - 2U_{i-1} \quad i > 0 \tag{4.30}$$

la cual es utilizada para convertir cualquier potencia de τ a su forma canónica, es decir, como un elemento $d_1\tau + d_0$, con $d_0, d_1 \in \mathbb{Z}$. Sin embargo, este algoritmo sólo contempla expansiones τ-NAF, dejando como trabajo a futuro el caso general para cualquier valor de ω.

A continuación, formularemos un algoritmo para la recuperación del entero equivalente, partiendo de una expansión $\omega\tau$-NAF obtenida mediante el generador de la ecuación (4.26).

Dada una expansión $\omega\tau$-NAF de longitud l, donde los coeficientes $u_i \in \{\alpha_u\}_{u \in I}$, podemos denotar a los representantes de congruencia mediante $\alpha_u = \beta_u + \gamma_u\tau$, y entonces se tiene:

$$\sum_{i=0}^{l-1} u_i\tau^i = \sum_{i=0}^{l-1} ((\beta_u)_i + (\gamma_u)_i\tau)\,\tau^i \tag{4.31}$$

$$= \sum_{i=0}^{l-1} (\beta_u)_i\tau^i + \sum_{i=0}^{l-1} (\gamma_u)_i\tau^{i+1} \tag{4.32}$$

aplicando la ecuación (4.30), se pueden sustituir las potencias de τ:

$$\sum_{i=0}^{l-1} u_i\tau^i = \sum_{i=0}^{l-1} (\beta_u)_i(U_i\tau - 2U_{i-1}) + \sum_{i=0}^{l-1} (\gamma_u)_i(U_{i+1}\tau - 2U_i) \tag{4.33}$$

$$= \tau\sum_{i=0}^{l-1} [(\beta_u)_iU_i + (\gamma_u)_i(U_{i+1})] - 2\sum_{i=0}^{l-1} [(\beta_u)_iU_{i-1} + (\gamma_u)_iU_i] \tag{4.34}$$

esta última ecuación es la representación canónica de la expansión. Se puede notar que tiene la forma $d_0 + d_i\tau$ con:

$$d_1 = (\gamma_u)_0 + \sum_{i=1}^{l-1} [(\beta_u)_iU_i + (\gamma_u)_i(U_{i+1})] \tag{4.35}$$

$$d_0 = (\beta_u)_0 - 2\sum_{i=1}^{l-1} [(\beta_u)_iU_{i-1} + (\gamma_u)_i(U_i)] \tag{4.36}$$

Dado que existe un entero s tal que cumple con la ecuación (4.28), se puede obtener el entero equivalente al aplicar:

$$k = d_0 + d_1\tau = d_0 + d_1 s \pmod r \tag{4.37}$$

En el algoritmo 15 se muestran las operaciones necesarias para la recuperación del entero equivalente dada una expansión $\omega\tau$-NAF.

Algoritmo 15 Recuperación del entero equivalente dada una expansión $\omega\tau$-NAF aleatoria.

Constantes: s tal que $\tau(P) = [s]P$.
Entrada: ω y una expansión aleatoria $\sum_{i=0}^{l-1} u_i \tau^i$.
Salida: $k \in \mathbb{Z}_r$, tal que $\sum_{i=0}^{l-1} u_i \tau^i \leftarrow \omega\tau\text{-NAF}(k \operatorname{partmod} \delta)$

1: $d_0 \leftarrow (\gamma_u)_0$
2: $d_1 \leftarrow 0$
3: **for** $i \leftarrow 1$ to $l-1$ **do**
4: **if** $u_i \neq 0$ **then**
5: $(\beta, \gamma) \leftarrow ((\beta_u)_i, (\gamma_u)_i)$
6: $d_0 \leftarrow d_0 + \beta U_i + \gamma U_{i+1}$
7: $d_1 \leftarrow d_1 + \beta U_{i-1} + \gamma U_i$
8: **end if**
9: **end for**
10: $d_1 \leftarrow (\beta_u)_0 - 2d_1$
11: **return** $k \leftarrow d_0 + d_1 s \pmod{r}$

SECCIÓN 4.6

Multiplicación escalar en curvas de Koblitz

En esta sección se analizan los métodos para realizar el cómputo de la multiplicación escalar en las curvas de Koblitz basados en los algoritmos expuestos en la sección 4.2 en la página 53.

Este trabajo supone el escenario de punto desconocido, es decir, el punto a multiplicar se conoce hasta que se desarrolla la multiplicación escalar. A diferencia del escenario de punto conocido, en donde el punto a multiplicar ha sido fijado con anterioridad, por ejemplo, en la fase de generación de llaves de la firma digital ECDSA [9], el punto P es un parámetro público invariable.

4.6.1. Multiplicación escalar de izquierda a derecha en curvas de Koblitz

Este algoritmo se fundamenta en la multiplicación escalar de izquierda a derecha (algoritmo 8 de la página 54), la idea principal es evaluar eficientemente un polinomio de variable τ y con coeficientes en $\{\alpha_u\}$ mediante la regla de Horner.

Antes de mostrar el algoritmo se explica a detalle la fase de pre-cómputo sobre el punto P.

ω	u	$u \bmod \tau^\omega$ (ec. 4.23)	2τ-NAF(α_u)	α_u
3	1	1	(1)	1
	3	$-\mu\tau+1$	$(-1,0,-1)$	$-\mu\tau+1$
4	1	1	(1)	1
	3	$\mu\tau-3$	$(1,0,-1)$	τ^2-1
	5	$\mu\tau-1$	$(1,0,1)$	τ^2+1
	7	$\mu\tau+1$	$(-\mu,0,0,1)$	$-\mu\tau^3-1$
5	1	1	(1)	1
	3	$\mu\tau-3$	$(1,0,-1)$	τ^2-1
	5	$\mu\tau-1$	$(1,0,1)$	τ^2+1
	7	$\mu\tau+1$	$(-\mu,0,0,-1)$	$-\mu\tau^3-1$
	9	$2\mu\tau-3$	$(-\mu,0,-\mu,0,0,1)$	$-\mu\tau^3\alpha_5+1$
	11	$2\mu\tau-1$	$(-1,0,-1,0,-1)$	$-\tau^2\alpha_5-1$
	13	$2\mu\tau+1$	$(-1,0,-1,0,1)$	$-\tau^2\alpha_5+1$
	15	$-3\mu\tau+1$	$(1,0,0,0,-1)$	$\tau^2\alpha_5-\alpha_5$

Tabla 4.2: Representantes de principales de las clases de equivalencia impares $\{\alpha_u\}$ para $\omega \in \{3,4,5\}$.

Pre-cómputo sobre P. La fase de pre-cómputo consiste en calcular los múltiplos de P denotados como:

$$P_u = \{\alpha_u P\}_{u \in I} \tag{4.38}$$

Esta fase puede ser optimizada, dado que el conjunto $\{\alpha_u\}$ queda determinado al definir la curva, así como el parámetro ω de la expansión $\omega\tau$-NAF.

En la tabla 4.2 se muestran los representantes principales de las clases de equivalencia impares para $\omega \in \{3,4,5\}$. La tercer columna hace referencia al cálculo de los representantes principales utilizando la ecuación (4.23), sin embargo algunos de los coeficientes β_u, γ_u son iguales a 2 ó 3, y requieren el uso de doblados y sumas de punto, lo cual incrementa el costo computacional. Para evitar este problema se calcula la expansión 2τ-NAF de los elementos α_u dando como resultado coeficientes en el conjunto $\{-1,0,1\}$, listados en la cuarta columna. La última columna muestra cómo calcular los múltiplos P_u utilizando sólo una suma de punto por cada múltiplo.

Esta suma de puntos se realiza en coordenadas afines, lo cual implica hacer una inversión por cada representante, en este trabajo se hace uso de una sola inversión para calcular todos los múltiplos

ω	Operaciones	Memoria (bits)
3	2M, 3S, 1I	4m
4	9M, 9S, 1I	8m
5	23M, 19S, 2I	16m

Tabla 4.3: Costo del pre-cómputo para $\omega \in \{3,4,5\}$ en términos de multiplicaciones (M), elevaciones al cuadrado (S) e inversiones (I) de elementos en el campo $\mathbb{F}_{2^m}$.

necesarios haciendo uso de la inversión simultánea descrita en la sección 2.4.4. En el caso de $\omega = 5$ se requiere aplicar la inversión simultánea un par de ocasiones debido a la dependencia de datos en α_5.

Además se aprovecha que la mayoría de los representantes tiene la forma $Q \pm P$, para algún $Q \in E_a$, de esta manera el segundo punto a sumar es P o $-P$, por lo tanto el cálculo de operaciones tales como $Q+P$ y $Q-P$ puede realizarse compartiendo algunos resultados intermedios provenientes de la fórmula para sumar puntos. El costo del pre-cómputo se resume en la tabla 4.3; en la segunda columna detalla el espacio en memoria para guardar los múltiplos pre-computados.

Presentación del algoritmo. Siguiendo el esquema básico de doblado y suma se presenta la multiplicación escalar de izquierda a derecha en las curvas de Koblitz (algoritmo 16). Dado un escalar $k \in \mathbb{Z}_r$ y un punto $P \in E_a(\mathbb{F}_{2^m})$ de orden r, se desea encontrar $Q \in E_a(\mathbb{F}_{2^m})$ tal que $Q = [k]P$.

La suma de puntos en el ciclo principal se calcula utilizando la suma mixta, donde Q está en coordenadas proyectivas LD y los puntos P_u están en coordenadas afines.

4.6.2. Multiplicación escalar de derecha a izquierda en curvas de Koblitz

La versión de derecha a izquierda del algoritmo de multiplicación escalar, como su nombre lo indica, recorre a los coeficientes de la expansión $\omega\tau$-NAF desde la potencia menos significativa $(u_0\tau^0)$ hasta la más significativa $(u_{l-1}\tau^{l-1})$.

Existen cambios sutiles en el algoritmo de derecha a izquierda con respecto a la forma de procesar la multiplicación escalar: se elimina la fase de pre-cómputo, se requiere una fase de post-cómputo de puntos, se hace cambio sistema de coordenadas para los puntos y el procesamiento del cálculo de la expansión $\omega\tau$-NAF puede ser integrado en el ciclo principal. A continuación se describen explícitamente dichos cambios.

Algoritmo 16 Multiplicación escalar de izquierda a derecha en curvas de Koblitz.

Entrada: ω, $k \in \mathbb{Z}_r$, $P \in E_a(\mathbb{F}_{2^m})$ de orden r.

Salida: $Q \in E_a(\mathbb{F}_{2^m})$ tal que $Q = [k]P$.

1: $\rho \leftarrow k$ partmod δ {Vía algoritmo 12.}
2: $\sum_{i=0}^{l-1} u_i \tau^i \leftarrow \omega\tau\text{NAF}(\rho)$ {Vía algoritmo 14.}
3: $P_u = \{\alpha_u P\}, \forall u \in I$
4: $Q \leftarrow O$
5: **for** $i = l-1$ **downto** 0 **do**
6: $\quad Q \leftarrow \tau Q$
7: $\quad$ **if** $u_i = \alpha_j$ **then**
8: $\quad\quad Q \leftarrow Q + P_j$
9: $\quad$ **else if** $u_i = -\alpha_j$ **then**
10: $\quad\quad Q \leftarrow Q - P_j$
11: $\quad$ **end if**
12: **end for**
13: **return** Q

Eliminación de fase de pre-cómputo. Este algoritmo no requiere el uso de los múltiplos P_u y su pre-cómputo, dado que en cada iteración se suma a Q el valor de R, el cual previamente fue inicializado con el valor de P. Posteriormente se realiza la actualización de $R \leftarrow \tau R$.

Fase de post-cómputo. Al utilizar expansiones binarias, en la iteración i-ésima, el punto R siempre contiene múltiplos $[2^i]P$, sin embargo en expansiones ω-NAF, donde los coeficientes $k_i \in \{0, 1, 3, 5, 7, \dots\}$, se deberá tener un conjunto $R_j = \{R_1, R_3, R_5, \dots\}$ de $2^{\omega-2}$ puntos; y en cada iteración actualizarlos mediante un doblado de puntos sobre cada elemento en el conjunto.

A pesar de que el doblado de puntos es reemplazado por el endomorfismo de Frobenius, en el caso de las expansiones $\omega\tau$-NAF, el cómputo no se realiza de esta forma, dado que en cada iteración se actualizarían $2^{\omega-2}$ puntos siendo que solamente uno de ellos es utilizado.

A fin de realizar un cómputo más eficiente, la actualización se realiza en un solo punto $R \leftarrow \tau R$ y se crea un conjunto Q_j de $2^{\omega-2}$ acumuladores. Así, en cada elemento se realizará la suma correspondiente con la clase de equivalencia del coeficiente i-ésimo.

La fase de post-cómputo en la multiplicación escalar de derecha a izquierda consiste en realizar el siguiente cómputo:

$$Q \leftarrow \sum_{u \in I} \alpha_u Q_u = [\alpha_1]Q_1 + [\alpha_3]Q_3 + [\alpha_5]Q_5 + \cdots + [\alpha_{2^{\omega-1}-1}]Q_{2^{\omega-1}-1} \tag{4.39}$$

ω	Operaciones
3	26M, 13S
4	79M, 45S
5	200M, 117S

Tabla 4.4: Costo del esfuerzo computacional en multiplicaciones (M) y elevaciones al cuadrado (S) del post-cómputo al evaluar la ecuación (4.39) para $\omega \in \{3,4,5\}$.

Es importante notar que los puntos en el conjunto Q_j son puntos desconocidos al momento de ser sumados, por lo tanto no se pueden aprovechar los resultados intermedios de las fórmulas de suma de puntos como en el caso del pre-cómputo. Para hacer esta suma de manera eficiente se agruparon los términos de acuerdo a las potencias de τ. La tabla 4.4 lista, en términos de multiplicaciones (M) y elevaciones al cuadrado (S), el esfuerzo computacional para calcular la fase de post-cómputo.

Coordenadas utilizadas. Para el caso de las sumas en el ciclo principal se utiliza la suma mixta de puntos, dado que los múltiplos de P son calculados en coordenadas afines, logrando con esto un ahorro de alrededor m elevaciones al cuadrado con respecto al algoritmo 16 de la página 69, al prescindir de la coordenada Z. A diferencia del cálculo del post-cómputo, donde todos los puntos están en coordenadas proyectivas LD.

Integración del cálculo de la expansión $\omega\tau$-NAF. Una característica notable de este algoritmo es que los coeficientes de la expansión $\omega\tau$-NAF son analizados en el mismo orden en que fueron generados por el algoritmo 14, es decir, desde u_0 hasta u_{l-1}.

Este orden permite intercalar el algoritmo de generación de la expansión $\omega\tau$-NAF con el ciclo principal de la multiplicación escalar. Este sutil cambio hace que la ejecución de la multiplicación escalar se acelere al combinar operaciones de ambos algoritmos.

El algoritmo 17 describe la multiplicación escalar de derecha a izquierda en las curvas de Koblitz.

Resumen

A lo largo de este capítulo se introdujo la aritmética de curvas elípticas, el grupo abeliano formado por los puntos de una curva elíptica posee gran relevancia para el desarrollo de protocolos criptográficos.

Algoritmo 17 Multiplicación escalar de derecha a izquierda en curvas de Koblitz.

Constantes: μ, t_ω de la ecuación (4.22), $\{\alpha_u = \beta_u + \gamma_u\tau\}_{u\in I}$ de la ecuación (4.23).

Entrada: $\omega, k \in \mathbb{Z}_r, P \in E_a(\mathbb{F}_{2^m})$ de orden r.

Salida: $Q \in E_a(\mathbb{F}_{2^m})$ tal que $Q = [k]P$.

1: $Q_u \leftarrow O|_{u\in I}$
2: $R \leftarrow P$
3: $(\rho_0 + \rho_1\tau) \leftarrow k$ partmod δ {Vía algoritmo 12.}
4: **while** $\rho_0 \neq 0$ **or** $\rho_1 \neq 0$ **do**
5: **if** ρ_0 es impar **then**
6: $u \leftarrow \rho_0 + \rho_1 t_\omega \pmod{2^\omega}$
7: **if** $u > 0$ **then**
8: $\xi \leftarrow 1$
9: $Q_u \leftarrow Q_u + R$
10: **else**
11: $\xi \leftarrow -1$, $u \leftarrow -u$
12: $Q_u \leftarrow Q_u - R$
13: **end if**
14: $(\rho_0, \rho_1) \leftarrow (\rho_0 - \xi\beta_u, \rho_1 - \xi\gamma_u)$
15: **end if**
16: $R \leftarrow \tau R$
17: $(\rho_0, \rho_1) \leftarrow (\rho_1 + \mu\rho_0/2, -\rho_0/2)$
18: **end while**
19: $Q \leftarrow \sum_{u\in I} \alpha_u Q_u$ {Usando la ecuación 4.39.}
20: **return** Q

La operación fundamental en estos protocolos es la multiplicación escalar, para computarla requiere de un gran esfuerzo computacional de operaciones de campo. Se examinaron dos algoritmos básicos para el cómputo de la multiplicación escalar, ambos conocidos como algoritmos de doblado y suma, en su versión de izquierda a derecha y viceversa, de derecha a izquierda.

Con el propósito de realizar un cómputo más eficiente de la multiplicación escalar, se introducen las curvas elípticas de Koblitz, las cuales poseen un endomorfismo, llamado endomorfismo de Frobenius, el cual que permite reemplazar los doblados de punto por aplicaciones de esta función al momento de calcular la multiplicación escalar.

Para realizar este cómputo ágilmente es necesario recodificar el escalar a una expansión $\omega\tau$-NAF, es decir, hacer una representación del escalar en base τ y con la propiedad de no adyacencia (ω-NAF), esto con la finalidad de aprovechar el bajo costo de la aplicación del endomorfismo.

Al final del capítulo se plantearon los algoritmos básicos para el cómputo de la multiplicación escalar utilizando el endomorfismo de Frobenius sobre las curvas de Koblitz. Ambos son adaptaciones de los algoritmos de suma y doblado presentados al inicio del capítulo.

5

Implementación y resultados

Solamente aquel que contribuye al futuro
tiene derecho a juzgar el pasado.

Friedrich Nietzsche

A lo largo de este capítulo, se expone la implementación en una arquitectura multinúcleo de la multiplicación escalar en las curvas de Koblitz.
Se explica en mayor detalle la implementación de la aritmética del campo finito binario $\mathbb{F}_{2^m}$, y cómo ésta es utilizada como infraestructura básica para procesar la multiplicación escalar en su versión secuencial y en la versión multinúcleo.
Se detalla el ambiente de desarrollo de la implementación, así como también los parámetros seleccionados de las curvas elípticas, con el propósito de alcanzar 112, 128 y 192 bits de seguridad.
Finalmente, resumiremos los resultados obtenidos en esta implementación.

SECCIÓN 5.1

Elección de curvas y parámetros

Un aspecto fundamental al momento de seleccionar las curvas elípticas es el nivel de seguridad que éstas ofrecen.

En el caso de los criptosistemas simétricos, la seguridad se encuentra determinada por el tamaño de su llave. Un cifrador ideal con una llave secreta de n bits de tamaño, posee n bits de seguridad dado que el mejor algoritmo conocido (por fuerza bruta) emplea 2^n operaciones para recuperar la llave.

En el caso de los criptosistemas asimétricos basados en curvas elípticas, el mejor algoritmo para resolver el problema de logaritmo discreto en curvas elípticas tiene una complejidad de $O(\sqrt{n})$, donde n es la cardinalidad del grupo [31].

En [1] se recomiendan tamaños de llave privada, tamaños de grupo y curvas elípticas sobre diversas extensiones de campo, con el propósito de brindar cinco niveles de seguridad. La tabla 5.1 muestra en la primera columna el nivel de seguridad, en la segunda se muestra un algoritmo de cifrado de llave privada, la tercer columna denota el tamaño en bits de las llaves para el criptosistema RSA, la columna cuarta y quinta de los tamaños de grupo recomendados para criptosistemas asimétricos basados en curvas elípticas.

Siguiendo la recomendación del NIST, este trabajo está enfocado en desarrollar tres niveles de seguridad, correspondientes a 112, 128 y 192 bits de seguridad, por lo cual se eligieron las curvas K-233, K-283 y K-409, respectivamente, donde la K-m denota a una curva de Koblitz definida en el campo finito binario $\mathbb{F}_{2^m}$.

SECCIÓN 5.2

Operaciones del campo $\mathbb{F}_{2^m}$

Representación de elementos. Un elemento en $\mathbb{F}_{2^m}$ es representado en lenguaje C mediante un arreglo unidimensional de $\lceil \frac{m}{64} \rceil$ registros de 64 bits (`unsigned long int`), donde el bit menos significativo corresponde con la potencia x^0 del elemento del campo. Análogamente, un elemento en $\mathbb{F}_{2^m}$ puede ser almacenado en $\lceil \frac{m}{128} \rceil$ y $\lceil \frac{m}{256} \rceil$ registros de 128 y 256 bits cada uno, en el banco de registros `XMM` y `YMM`, respectivamente.

Suma. La suma de dos elementos en el campo $\mathbb{F}_{2^m}$ se realiza fácilmente al computar la operación `XOR`. En el caso de las instrucciones `SSE`, ésta puede ser ejecutada mediante la función: `_mm_xor_si128`, y en el caso del conjunto de instrucciones `AVX` mediante: `_mm256_xor_si256`.

Tamaño de llave simétrica	Algoritmo de llave privada	Algoritmo de llave pública (RSA)	Campo $\mathbb{F}_p$	Campo $\mathbb{F}_{2^m}$
80	SKIPJACK	$n = 1,024$	$\|p\| = 192$	$m = 163$
112	Triple-DES	$n = 2,048$	$\|p\| = 224$	$m = 233$
128	AES-128	$n = 3,072$	$\|p\| = 256$	$m = 283$
192	AES-192	$n = 8,192$	$\|p\| = 384$	$m = 409$
256	AES-256	$n = 15,360$	$\|p\| = 521$	$m = 571$

Tabla 5.1: Curvas elípticas recomendadas por el NIST para usos del Gobierno Federal de los EE.UU. descritas en [1].

Multiplicación. Para realizar la multiplicación de dos elementos en el campo $\mathbb{F}_{2^m}$ se calcula mediante una multiplicación polinomial seguida por una reducción polinomial $(\mathrm{mod}\, p(x))$.

Para construir un multiplicador polinomial de m bits, se decidió utilizar el algoritmo de Karatsuba, como ya se demostró en el algoritmo 2 de la sección 2.4.4 en la página 18. La idea principal consiste en la forma de realizar una multiplicación de dos polinomios de grado m, puesto que ésta puede ser calculada con tres multiplicaciones de polinomios de grado $\lceil \frac{m}{2} \rceil$ y sumar los resultados de cada multiplicación de acuerdo a la ecuación (2.26).

Este particionamiento se puede continuar recursivamente hasta nivel de bits, o bien, cuando se cuente con un multiplicador polinomial eficiente. He aquí en donde entra en juego el multiplicador polinomial integrado en el conjunto de instrucciones `AES-NI`.

En este trabajo se decidió realizar la multiplicación polinomial con ayuda de la instrucción `PCLMULQDQ`, descrita en la sección 3.4.5 de la página 37. Esta instrucción recibe como parámetros de entrada: la representación vectorial de dos polinomios de grado menor a 64, denotados como $a(x)$ y $b(x)$. Una vez que se ha procesado la multiplicación polinomial, la instrucción obtiene como resultado la representación vectorial de un polinomio de grado menor a 128, llamado $c(x)$, tal que $c(x) = a(x) \cdot b(x)$.

En la figura 5.1 se muestra la forma de particionar los polinomios de grado $m \in \{233, 283, 409\}$ para aplicar de forma recursiva el algoritmo de Karatsuba, la recursión se detiene cuando se llega a polinomios de grado menor a 64, en donde se aplica `PCLMULQDQ`.

La instrucción `PCLMULQDQ` puede ser invocada mediante la función `_mm_clmulepi64_si128`. El mnemónico de la instrucción es la abreviación de "multiplicación sin acarreo" (en inglés *carry-less multiplier*), puesto que es análoga a una multiplicación en el conjunto de los números enteros, sin embargo no toma en consideración los bits de acarreo al realizar la suma. Después de haber realizado la multiplicación polinomial, se procede a realizar la reducción polinomial para obtener

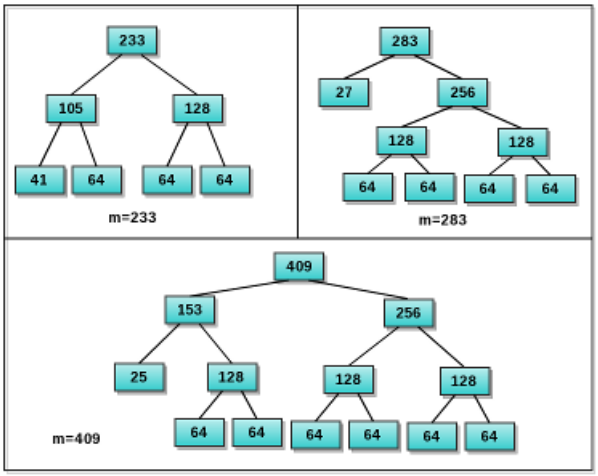


Figura 5.1: Descomposición recursiva de polinomios de grado $m \in \{233, 283, 409\}$.

un elemento en el campo $\mathbb{F}_{2^m}$.

Representación fraccionaria. En [10] se introduce la representación fraccionaria, la cual es una forma de representar a los elementos del campo $\mathbb{F}_{2^m}$, y a continuación se describe.

Sea $a(x) \in \mathbb{F}_{2^m}$, podemos representar al elemento $a(x)$ al separarlo en dos polinomios:

$$a_L(x) = \sum_{\substack{0 \le i < m, \\ 0 \le i \bmod 8 \le 3}} a_i x^i \tag{5.1}$$

$$a_H(x) = \sum_{\substack{0 \le i < m, \\ 4 \le i \bmod 8 \le 7}} a_i x^{i-4} \tag{5.2}$$

donde a_L almacena los 4 bits menos significativos de cada byte donde se almacena a. Mientras que a_H almacena los 4 bits más significativos de cada byte donde se almacena a. Usando esta representación, $a(x)$ puede ser escrita de la siguiente forma:

$$a(x) = a_H(x)x^4 + a_L(x) \tag{5.3}$$

Para separar los bits de la representación vectorial y obtener los vectores a_H y a_L, se utilizan máscaras de bits y corrimientos en registros. Esta representación agrupa términos de 4 bits, lo que permite realizar operaciones de campo de forma eficiente.

Elevación al cuadrado. Esta operación puede ser computada ágilmente mediante instrucciones SIMD, tal como se describe a continuación: dado un elemento $a(x) \in \mathbb{F}_{2^m}$ representado en forma fraccionaria, se desea encontrar un elemento $c(x) \in \mathbb{F}_{2^m}$ tal que:

$$c(x) = a(x)^2 = (a_H(x)x^4 + a_L(x))^2 = a_H(x)^2x^8 + a_L(x)^2, \tag{5.4}$$

lo cual implica que para efectuar el cuadrado de $a(x)$; sólo es necesario calcular el cuadrado de $a_L(x)$ y de $a_H(x)$ multiplicado por x^8, y al final, realizar la reducción polinomial $(\mathrm{mod}\, p(x))$ para obtener un elemento en el campo $\mathbb{F}_{2^m}$.

Para calcular $a_L(x)^2$ y $a_H(x)^2$, se deben insertar ceros entre cada coeficiente. Es preciso recordar que los coeficientes de la representación fraccionaria son conjuntos de 4 bits, entonces se puede construir una tabla de pre-cómputo T que contenga 2^4 entradas de 8 bits cada una, tal que la entrada de la tabla $T[b_3,b_2,b_1,b_0] = \{0,b_3,0,b_2,0,b_1,0,b_0\}$, donde $b_i|_{0\leq i<4}$ son los bits de entrada.

La función `_mm_shuffle_epi8` es capaz de realizar cualquier función que toma 4 bits de entrada y devuelve 8 bits de salida, mediante el uso de una tabla previamente almacenada. A este tipo de instrucción se le conoce como tabla de consulta en paralelo, (del inglés *parallel table lookup*), la cual fue analizada en [27]. Este nombre proviene del hecho, que la consulta a la tabla se realiza de forma paralela utilizando como entrada un vector de datos independientes.

Una vez que se obtienen los vectores $a_H(x)^2$ y $a_L(x)^2$, éstos deben volver a la representación en base polinomial. Para esto, se hace uso de las instrucciones `_mm_unpackhi(lo)_epi8` cuya función es intercalar los bytes más (menos) significativos de los vectores $a_H(x)^2$ y $a_L(x)^2$. Finalmente, se obtiene un polinomio de grado $2m-2$, al cual se aplica la reducción polinomial para obtener un elemento en el campo $\mathbb{F}_{2^m}$.

El algoritmo 18 enlistan las operaciones necesarias para el cálculo la elevación al cuadrado de un elemento en $\mathbb{F}_{2^m}$ usando la representación fraccionaria.

Raíz cuadrada. Usando la representación fraccionaria expuesta en la ecuación (5.3), es posible calcular la raíz cuadrada de un elemento $a(x) \in \mathbb{F}_{2^m}$ mediante:

$$\sqrt{a(x)} = \sqrt{a_H(x)x^4 + a_L(x)} = \sqrt{a_H(x)}x^2 + \sqrt{a_L(x)}, \tag{5.5}$$

es decir, basta con calcular las raíces cuadradas de $a_L(x)$ y de $a_H(x)$, multiplicar este último término por x^2 y finalmente realizar la reducción polinomial $(\mathrm{mod}\, p(x))$.

De acuerdo a la ecuación (2.29), la raíz cuadrada de un elemento $a(x) \in \mathbb{F}_{2^m}$ se puede calcular mediante:

$$\sqrt{a(x)} = \sqrt{x} \cdot (a_{L_{impar}}(x) + a_{H_{impar}}(x)x^2) + a_{L_{par}}(x) + a_{H_{par}}(x)x^2 \tag{5.6}$$

Algoritmo 18 Elevación al cuadrado en $\mathbb{F}_{2^m}$.

Almacenamiento: Utilizando $\lceil \frac{m}{128} \rceil$ registros de 128 bits, denotados como $\mathsf{a}[\cdot]$.

Entrada: $a(x) \in \mathbb{F}_{2^m}$.

Salida: $c(x) \in \mathbb{F}_{2^m}$ tal que $c(x) = a(x)^2$.

1: $sq \leftarrow$ `0x5554515045444140151411005040100`
2: $mask \leftarrow$ `0x0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F`
3: **for** $i = 0$ **to** $\lceil \frac{m}{128} \rceil - 1$ **do**
4: $\mathsf{a_L}[i] \leftarrow \mathsf{a}[i] \wedge mask$
5: $\mathsf{a_H}[i] \leftarrow (\mathsf{a}[i] \gg 4) \wedge mask$
6: $\mathsf{a_L}[i]^2 \leftarrow$ `_mm_shuffle_epi8`$(\mathsf{a_L}[i], sq)$
7: $\mathsf{a_H}[i]^2 \leftarrow$ `_mm_shuffle_epi8`$(\mathsf{a_H}[i], sq)$
8: $\mathsf{t}[2i] \leftarrow$ `_mm_unpacklo_epi8`$(\mathsf{a_L}[i]^2, \mathsf{a_H}[i]^2)$
9: $\mathsf{t}[2i+1] \leftarrow$ `_mm_unpackhi_epi8`$(\mathsf{a_L}[i]^2, \mathsf{a_H}[i]^2)$
10: **end for**
11: **return** $c(x) \leftarrow t(x) \bmod p(x)$ {Usando ecuación (2.27).}

Para realizar el cálculo de esta operación, se procede como en el caso de la elevación al cuadrado, es decir: se separan bits, se hacen corrimientos, se realizan consultas a tablas, y se intercalan los vectores resultantes. Para obtener más detalles sobre esta implementación consúltese [10].

Elevaciones al cuadrado consecutivas. Para realizar la inversión de un elemento en el grupo multiplicativo $\mathbb{F}^*_{2^m}$, es conveniente contar con una función que dado un elemento $a(x) \in \mathbb{F}^*_{2^m}$, calcule el elemento $a(x)^{2^k}$ para algún valor fijo $k \in \mathbb{Z}_m$, $k > 1$, es decir, que realice k elevaciones al cuadrado de forma consecutiva.

En [13] esta operación es implementada eficientemente utilizando grandes tablas de consulta, idea análoga a la que fuera propuesta en [6].

Dado un valor fijo de k, se construye de una tabla T que contenga $16\lceil \frac{m}{4} \rceil$ elementos de campo, tal que las entradas de la tabla se calculen mediante la siguiente ecuación:

$$T[j, i_0 + 2i_1 + 4i_2 + 8i_3] = (i_0 x^{4j} + i_1 x^{4j+1} + i_2 x^{4j+2} + i_3 x^{4j+3})^{2^k} \tag{5.7}$$

donde $i_0, i_1, i_2, i_3 \in \{0, 1\}$ y $0 \leq j < \lceil \frac{m}{4} \rceil$.

Una vez calculada y almacenada esta tabla en memoria, el cómputo de las elevaciones al cuadrado consecutivas se realiza mediante la siguiente ecuación:

$$a(x)^{2^k} = \sum_{j=0}^{\lceil \frac{m}{4} \rceil - 1} T[j, \lfloor a/2^{4j} \rfloor \bmod 2^4] \tag{5.8}$$

para lo cual es necesario realizar $\lceil \frac{m}{4} \rceil$ consultas a la tabla junto con una operación `XOR` por cada consulta para acumular el resultado.

SECCIÓN 5.3

Multiplicación escalar en paralelo

En esta sección se presenta la formulación algorítmica de la multiplicación escalar en su versión paralela.

Cabe hacer la aclaración, que en esta sección los algoritmos están presentados en forma de columnas, donde las columnas representan el procesador o núcleo donde se ejecutan las instrucciones. Las barreras de sincronización indican explícitamente el punto de sincronización de los hilos de ejecución, es decir, en dicho punto, un hilo debe esperar a que los demás terminen, para poder ejecutar la instrucción que continúa después de la barrera de sincronización.

La primer parte de la multiplicación escalar puede ser dividida en dos partes, dado que el cálculo de la expansión $\omega\tau$-NAF y el cálculo del pre-cómputo son tareas totalmente independientes, lo cual permite que puedan ser ejecutadas simultáneamente.

A continuación se exponen dos técnicas de particionamiento que son aplicadas al ciclo principal que realiza las sumas de puntos.

5.3.1. $(\tau^{-1}|\tau)$-y-suma

Basado en [6], la multiplicación escalar puede ser paralelizada al representar al escalar k en términos del operador τ y τ^{-1}. Por lo tanto, dada una expansión $\omega\tau$-NAF de k:

$$k = \sum_{i=0}^{l-1} u_i \tau^i$$

donde l es la longitud de la expansión y $u_i \in \{\alpha_u\}$. La expresión anterior puede reescribirse como:

$$\begin{aligned} k &= u_0 + u_1\tau + u_2\tau^2 + \cdots + u_n\tau^n + u_{n+1}\tau^{-(m-n-1)} + \cdots + u_{l-1} \\ &= \sum_{i=0}^{n} u_i\tau^i + \sum_{n+1}^{l-1} u_i\tau^{-(m-i)} \end{aligned} \tag{5.9}$$

donde $0 < n < m$. En la última expresión puede observarse que la multiplicación escalar puede realizarse mediante dos sumas independientes: la primera utiliza el operador de Frobenius, y la segunda utiliza su inverso, τ^{-1}.

La variable n toma un valor cercano a $\lfloor \frac{m}{2} \rfloor$, dada la diferencia entre el costo del cómputo del endomorfismo τ y su inverso. El valor exacto de n será el que optimice el tiempo de ejecución de la multiplicación escalar.

El algoritmo 19 muestra la multiplicación escalar usando esta técnica de paralelización.

5.3.2. $(\tau|\tau)$-y-suma

Tomando como base el algoritmo de izquierda a derecha (algoritmo 16), dada una expansión $\omega\tau$-NAF de longitud l, ésta se puede dividir en dos partes de iguales, de forma que se procese la multiplicación escalar como sigue:

$$\begin{aligned}
\rho &\leftarrow k \text{ partmod } \delta \\
\sum_{i=0}^{l-1} u_i\tau^i &\leftarrow \omega\tau\text{-NAF}(\rho) \\
n &\leftarrow \left\lfloor \frac{m}{2} \right\rfloor \\
Q &\leftarrow \sum_{i=0}^{l-1} u_i\tau^i P \\
&= \sum_{i=0}^{n-1} u_i\tau^i P + \tau^n \sum_{i=n}^{l-1} u_i\tau^{i-n} P \\
&= Q_0 + \tau^n Q_1
\end{aligned}$$

De esta manera se obtienen resultados intermedios Q_0 y Q_1; a este último punto se le aplican n elevadas al cuadrado consecutivas, dado que n es un valor fijo se puede utilizar la ecuación (5.8). El algoritmo 20 describe este modo de particionamiento.

SECCIÓN 5.4

Resultados

En esta sección se muestran las arquitecturas y procesadores utilizados en el cómputo de la multiplicación escalar. Se exponen los resultados de la implementación en dos diferentes arquitecturas y se realiza una comparativa con otras implementaciones.

5.4.1. Arquitecturas

Westmere. La arquitectura Westmere de Intel® fue la primera en integrar el conjunto de instrucciones `AES-NI`; hecho que estimuló el desarrollado de este trabajo, al aplicar estas instrucciones en la criptografía de curvas elípticas.

Entre las características principales de esta arquitectura mecionamos que posee un predictor de ramificaciones, cuenta con tres niveles de memoria caché, integra la tecnología *Hyper-Threading* y la tecnología *Turbo Boost*.

Algoritmo 19 Multiplicación escalar en curvas de Koblitz usando $(\tau^{-1}|\tau)$-y-suma.

Constantes: $n \approx \lfloor \frac{m}{2} \rfloor$

Entrada: ω, $k \in \mathbb{Z}_r$, $P \in E_a(\mathbb{F}_{2^m})$ de orden r.

Salida: $Q \in E_a(\mathbb{F}_{2^m})$ tal que $Q = [k]P$.

1: $\rho \leftarrow k$ partmod δ {Alg. 12}

2: $\sum_{i=0}^{l-1} u_i \tau^i \leftarrow \omega\tau\text{NAF}(\rho)$ {Alg. 14}

3: $P_u = \{\alpha_u P\}, \forall u \in I$ {Ec. 4.38}

{----------barrera de sincronización----------}

4: $Q_0 \leftarrow O$

5: **for** $i = n$ **to** 0 **do**

6: $\quad Q_0 \leftarrow \tau Q_0$

7: $\quad$ **if** $u_i = \alpha_j$ **then**

8: $\quad\quad Q_0 \leftarrow Q_0 + P_j$

9: $\quad$ **else if** $u_i = -\alpha_j$ **then**

10: $\quad\quad Q_0 \leftarrow Q_0 - P_j$

11: $\quad$ **end if**

12: **end for**

13: $Q_1 \leftarrow O$

14: **for** $i = n+1$ **to** $l-1$ **do**

15: $\quad Q_1 \leftarrow \tau^{-1} Q_1$

16: $\quad$ **if** $u_i = \alpha_j$ **then**

17: $\quad\quad Q_1 \leftarrow Q_1 + P_j$

18: $\quad$ **else if** $u_i = -\alpha_j$ **then**

19: $\quad\quad Q_1 \leftarrow Q_1 - P_j$

20: $\quad$ **end if**

21: **end for**

{----------barrera de sincronización----------}

22: $Q \leftarrow Q_0 + Q_1$

23: **return** Q

Algoritmo 20 Multiplicación escalar en curvas de Koblitz usando $(\tau|\tau)$-y-suma.

Constantes: $n = \lfloor \frac{m}{2} \rfloor$

Entrada: ω, $k \in \mathbb{Z}_r$, $P \in E_a(\mathbb{F}_{2^m})$ de orden r.

Salida: $Q \in E_a(\mathbb{F}_{2^m})$ tal que $Q = [k]P$.

1: $\rho \leftarrow k$ partmod δ {Alg. 12}
2: $\sum_{i=0}^{l-1} u_i \tau^i \leftarrow \omega\tau\text{NAF}(\rho)$ {Alg. 14}
3: $P_u = \{\alpha_u P\}, \forall u \in I$ {Ec. 4.38}

{----------barrera de sincronización----------}

4: $Q_0 \leftarrow O$
5: **for** $i = n-1$ **to** 0 **do**
6: $\quad Q_0 \leftarrow \tau Q_0$
7: $\quad$ **if** $u_i = \alpha_j$ **then**
8: $\quad\quad Q_0 \leftarrow Q_0 + P_j$
9: $\quad$ **else if** $u_i = -\alpha_j$ **then**
10: $\quad\quad Q_0 \leftarrow Q_0 - P_j$
11: $\quad$ **end if**
12: **end for**

13: $Q_1 \leftarrow O$
14: **for** $i = l-1$ **to** n **do**
15: $\quad Q_1 \leftarrow \tau Q_1$
16: $\quad$ **if** $u_i = \alpha_j$ **then**
17: $\quad\quad Q_1 \leftarrow Q_1 + P_j$
18: $\quad$ **else if** $u_i = -\alpha_j$ **then**
19: $\quad\quad Q_1 \leftarrow Q_1 - P_j$
20: $\quad$ **end if**
21: **end for**

{----------barrera de sincronización----------}

22: $Q \leftarrow Q_0 + \tau^n Q_1$
23: **return** Q

La arquitectura posee procesadores con dos, cuatro y ocho núcleos en un mismo circuito integrado.

Sandy Bridge. Durante el desarrollo del presente trabajo de tesis, en enero de 2011, salió al mercado la nueva arquitectura llamada Sandy Bridge de Intel®. Extendiendo a la arquitectura Westmere, la arquitectura Sandy Bridge agrega nuevas funcionalidades, dentro de las que destacan: la optimización del predictor de ramificaciones y la reducción de la latencia de las instrucciones de acceso a memoria, ahora el procesador es capaz de ejecutar dos instrucciones en un ciclo de reloj.

En arquitecturas anteriores a la arquitectura Sandy Bridge, cuando se realizaba una operación binaria, el resultado de la operación se almacenaba en alguno de los registros de entrada, reescribiendo el valor de alguno de los operandos. De esta forma, las instrucciones en lenguaje ensamblador, sólo direccionan a dos registros (o localidades de memoria), por lo que este código se le conoce como código de dos operandos.

Una característica notable de la arquitecura Sandy Bridge es la capacidad de ejecutar código de tres operandos, es decir, el resultado de la operación puede ser almacenado en un tercer registro, sin que los registros de entrada sean modificados.

La arquitectura Sandy Bridge es la primera en incluir el conjunto de instrucciones `AVX`, la cual consta de un banco de 16 registros de 256 bits como se detalló en la sección 3.4.6 en la página 38.

5.4.2. Pruebas de rendimiento

Las pruebas de rendimiento fueron realizadas en dos procesadores que cuentan con las arquitecturas antes mencionadas. En la tabla 5.2 se enlistan las especificaciones técnicas de estos procesadores.

Tiempo de cómputo. El tiempo de cómputo de las operaciones puede ser medido tomando como unidad base los microsegundos, sin embargo, la medición del tiempo está directamente relacionada con la frecuencia del procesador. Al momento de comparar con otras implementaciones resulta más complicado, puesto que se debe tomar en cuenta tanto el tiempo como la frecuencia del procesador.

En este trabajo se determinó utilizar el número de ciclos de reloj como unidad de medida estándar. Existe una instrucción del lenguaje ensamblador que permite obtener este valor (`RDTSC`). Esta unidad permite comparar diferentes implementaciones, la cuales hasta cierto punto no son dependientes de la computadora donde se realicen las pruebas.

Reduciendo la aleatoreidad. Con el propósito de obtener tiempos precisos, se obtuvo la media aritmética de 10^4 mediciones tomadas.

Arquitectura	Westmere	Sandy Bridge
Serie del procesador	Core i3, Core i5, Core i7	Core i3, Core i5, Core i7 (Segunda generación)
Modelo del procesador	Core i5-660	Core i7-2600K
Memoria caché L3	4 MB	8 MB
Frecuencia nominal	3.33 GHz	3.40 GHz
Frecuencia máxima	3.60 GHz	3.80 GHz
Número de núcleos	2	4
Hyper-Threading	Si	Si
Turbo Boost	Si	Si
`SSE4.2` y anteriores	Si	Si
`AES-NI`	Si	Si
`AVX`	No	Si

Tabla 5.2: Especificaciones técnicas de los procesadores utilizados en este trabajo, consúltese [17] y [18], para mayor información sobre los procesadores Core i5-660 y Core i7-2600K, respectivamente.

Existen ciertos factores que pueden afectar la lectura de las mediciones, dos de los más importantes se describen a continuación:

- La tecnología *Hyper-Threading* descrita en 3.5.1 en la página 40, acelera la ejecución de instrucciones en los procesadores lógicos por cada núcleo de procesamiento.
- La tecnología *Turbo Boost* presente en los procesadores de Intel®, aumenta la frecuencia de reloj cuando el procesador está saturado y la disminuye cuando el procesador está ocioso.

Estas tecnologías producen inestabilidad en la toma de tiempos, tal y como se describe en el sitio de pruebas de rendimiento, eBACS [12], cuya recomendación es deshabilitar estas tecnologías.

Interpretación de las tablas. En las siguientes secciones se exponen los tiempos de cómputo obtenidos por las diferentes arquitecturas. Los tiempos se presentan mediante tablas con las siguientes columnas:

1. Muestra la operación matemática o criptográfica.
2. La extensión del campo binario sobre el que se trabaja.

a) GCC, el compilador utilizado.

b) ICC, el compilador utilizado.

c) op/M, denota la proporción de las operaciones con respecto a la multiplicación de campo utilizando el compilador ICC.

En el caso de la multiplicación escalar:

1. La columna ω denota el ancho de las expansiones $\omega\tau$-NAF como fue descrito en la página 60.
2. Los valores enlistados en las columnas deberán ser multiplicados por 10^3 para obtener el número exacto en ciclos de reloj.
3. En el cálculo de la raíz cuadrada en el campo $\mathbb{F}_{2^{283}}$, el término $\sqrt{x}$ es un polinomio con 68 coeficientes debido a que el polinomio irreducible, $p(x) = x^{283} + x^{12} + x^7 + x^5 + 1$, no es amigable a la extracción de raíces cuadradas, tal como se describe en [11]. Debido a esto, se omitió la implementación de la raíz cuadrada utilizando el polinomio irreducible propuesto por el NIST. Las entradas en la tabla con (–) son producto de esta deficiencia.

Tiempos obtenidos. A continuación se presentan los resultados de las pruebas de rendimiento para la arquitectura Westmere. En la tabla 5.3 se muestran los tiempos de cómputo de las operaciones del campo $\mathbb{F}_{2^m}$. En la tabla 5.4 se muestran los tiempos de cómputo de las operaciones aritméticas del grupo $E_a(\mathbb{F}_{2^m})$.

Las tablas 5.5 y 5.6 muestran el tiempo de cómputo para el cálculo de la multiplicación escalar en curvas de Koblitz en su versión secuencial y paralela, respectivamente.

De la misma forma, se presentan los resultados de las pruebas de rendimiento para la arquitectura Sandy Bridge. En la tabla 5.7, se enlistan los tiempos de cómputo de las operaciones del campo $\mathbb{F}_{2^m}$. En la tabla 5.8 se muestran los tiempos de cómputo de las operaciones aritméticas del grupo $E_a(\mathbb{F}_{2^m})$.

Las tablas 5.9 y 5.10 muestran el tiempo de cómputo para el cálculo de la multiplicación escalar en curvas de Koblitz en su versión secuencial y paralela, respectivamente.

5.4.3. Comparación con otras implementaciones

Con el fin de evaluar el trabajo aquí desarrollado de una forma objetiva, haremos una comparativa con otros trabajos similares que procesan la multiplicación escalar.

Existen muchas implementaciones secuenciales de la multiplicación escalar sobre diferentes sistemas de curvas elípticas, haciendo que las comparaciones no sean del todo uniformes. Para evitar estas desigualdades las comparativas se realizarán en base al nivel de seguridad que proveen.

Operación de campo	$\mathbb{F}_{2^{233}}$			$\mathbb{F}_{2^{283}}$			$\mathbb{F}_{2^{409}}$		
	GCC	ICC	op/M	GCC	ICC	op/M	GCC	ICC	op/M
Multiplicación	120	122	1.00	225	212	1.00	319	325	1.00
Raíz cuadrada	59	55	0.45	–	–	–	54	48	0.15
Elevar al cuadrado	28	28	0.23	66	54	0.25	40	44	0.14
a^{2^k}	174	165	1.35	287	292	1.37	462	460	1.42
Inversión	2,764	2,577	21.12	5,509	4,885	23.04	9,381	9,789	30.12

Tabla 5.3: Ciclos de reloj obtenidos en la arquitectura Westmere para las operaciones del campo $\mathbb{F}_{2^m}$, con $m = \{233, 283, 409\}$.

Operación de punto	$E_a(\mathbb{F}_{2^{233}})$			$E_a(\mathbb{F}_{2^{283}})$			$E_a(\mathbb{F}_{2^{409}})$		
	GCC	ICC	op/M	GCC	ICC	op/M	GCC	ICC	op/M
Frobenius (x, y)	56	56	0.46	132	108	0.51	80	80	0.25
Frobenius (x, y, z)	84	84	0.69	198	162	0.76	120	120	0.37
Doblado	625	626	5.13	1,279	1,150	5.42	1,484	1,508	4.64
Suma (mixta)	1,167	1,142	9.36	1,311	1,160	5.47	2,852	2,888	8.88
Suma (general)	1,834	1,817	14.89	3,454	3,143	14.82	4,543	4,606	14.17

Tabla 5.4: Ciclos de reloj obtenidos en la arquitectura Westmere para las operaciones del grupo $E_a(\mathbb{F}_{2^m})$, con $m = \{233, 283, 409\}$.

ω	Método	K-233		K-283		K-409	
		GCC	ICC	GCC	ICC	GCC	ICC
3	Izquierda a derecha	111	110	228	212	413	416
	Derecha a izquierda	98	98	204	192	381	389
4	Izquierda a derecha	97	95	199	184	353	355
	Derecha a izquierda	90	89	183	172	332	339
5	Izquierda a derecha	92	90	188	173	326	328
	Derecha a izquierda	95	93	188	175	321	332

Tabla 5.5: Ciclos de reloj obtenidos en la arquitectura Westmere para el cálculo de la multiplicación escalar en curvas de Koblitz K-233, K-283 y K-409 (versión secuencial).

ω	Método	K-233		K-283		K-409	
		GCC	ICC	GCC	ICC	GCC	ICC
3	(τ,τ)-y-suma	73	74	146	139	248	248
	(τ^{-1},τ)-y-suma	80	78	–	–	253	248
4	(τ,τ)-y-suma	68	65	128	121	216	214
	(τ^{-1},τ)-y-suma	73	69	–	–	218	214
5	(τ,τ)-y-suma	63	58	120	110	197	191
	(τ^{-1},τ)-y-suma	68	63	–	–	197	194

Tabla 5.6: Ciclos de reloj obtenidos en la arquitectura Westmere para el cálculo de la multiplicación escalar en curvas de Koblitz K-233, K-283 y K-409 (versión en paralelo).

Operación de campo	$\mathbb{F}_{2^{233}}$			$\mathbb{F}_{2^{283}}$			$\mathbb{F}_{2^{409}}$		
	GCC	ICC	op/M	GCC	ICC	op/M	GCC	ICC	op/M
Multiplicación	100	100	1.00	168	169	1.00	270	273	1.00
Raíz cuadrada	63	56	0.56	–	–	–	52	49	0.18
Elevar al cuadrado	22	24	0.24	41	43	0.25	35	34	0.12
a^{2^k}	133	112	1.12	307	269	1.59	419	392	1.43
Inversión	2,215	2,110	21.10	4,263	4,090	24.20	7,256	7,858	28.78

Tabla 5.7: Ciclos de reloj obtenidos en la arquitectura Sandy Bridge para las operaciones del campo $\mathbb{F}_{2^m}$, con $m = \{233, 283, 409\}$.

Operación de punto	$E_a(\mathbb{F}_{2^{233}})$			$E_a(\mathbb{F}_{2^{283}})$			$E_a(\mathbb{F}_{2^{409}})$		
	GCC	ICC	op/M	GCC	ICC	op/M	GCC	ICC	op/M
Frobenius (x,y)	44	48	0.48	82	86	0.51	70	68	0.68
Frobenius (x,y,z)	66	72	0.72	123	129	0.76	105	102	1.02
Doblado	540	540	5.40	907	920	5.44	1,283	1,291	4.73
Suma (mixta)	959	953	9.53	953	939	5.56	2,395	2,413	8.84
Suma (general)	1,530	1,517	15.17	2,497	2,504	14.81	3,822	3,848	14.10

Tabla 5.8: Ciclos de reloj obtenidos en la arquitectura Sandy Bridge para las operaciones del grupo $E_a(\mathbb{F}_{2^m})$, con $m = \{233, 283, 409\}$.

ω	Método	K-233		K-283		K-409	
		GCC	ICC	GCC	ICC	GCC	ICC
3	Izquierda a derecha	83.8	84.0	169	170	325.9	325.5
	Derecha a izquierda	75.0	74.9	152	154	299.8	301.5
4	Izquierda a derecha	72.9	73.1	148	148	277.0	277.1
	Derecha a izquierda	69.3	67.8	136	138	261.5	262.4
5	Izquierda a derecha	69.4	69.9	139	140	255.6	256.2
	Derecha a izquierda	73.0	72.4	139	141	255.7	257.1

Tabla 5.9: Ciclos de reloj obtenidos en la arquitectura Sandy Bridge para el cálculo de la multiplicación escalar en curvas de Koblitz K-233, K-283 y K-409 (versión secuencial).

ω	Método	K-233		K-283		K-409	
		GCC	ICC	GCC	ICC	GCC	ICC
3	(τ,τ)-y-suma	59.5	58.6	106	108	195.0	191.6
	(τ^{-1},τ)-y-suma	64.6	63.3	–	–	196.3	194.5
4	(τ,τ)-y-suma	53.3	50.8	94	93	167.5	165.2
	(τ^{-1},τ)-y-suma	57.9	55.7	–	–	168.2	166.3
5	(τ,τ)-y-suma	48.2	46.5	90	86	154.1	148.8
	(τ^{-1},τ)-y-suma	54.9	51.0	–	–	154.0	150.3

Tabla 5.10: Ciclos de reloj obtenidos en la arquitectura Sandy Bridge para el cálculo de la multiplicación escalar en curvas de Koblitz K-233, K-283 y K-409 (versión en paralelo).

En la tabla 5.11 se comparan algunas implementaciones secuenciales de la multiplicación escalar en diferentes sistemas de curvas elípticas, todas éstas ofrecen 128 bits de seguridad.

Los tiempos obtenidos han mejorado algunas implementaciones en dispositivos reconfigurables. A modo de ejemplo: el cómputo de la multiplicación escalar en la curva K-163 utilizando un dispositivo Stratix II, reportado en [47], procesa en 4.91 μs la multiplicación escalar a 80 bits de seguridad, es decir, por cada bit de seguridad emplea 61.375 ns, mientras que resultado de este trabajo, se procesa la multiplicación escalar a 112 bits de seguridad, en la curva K-233, en 13.67 μs, obteniendo un rendimiento de 122.054 ns por cada bit de seguridad; en comparativa, la implementación realizada en un dispositivo reconfigurable es apenas 1.98 veces más rápida que utilizando una computadora convencional que cuente con la arquitectura Sandy Bridge.

En la tabla 5.12 se muestran algunas implementaciones en dispositivos reconfigurables y su comparación con los resultados de este trabajo.

5.4.4. Análisis de resultados

Haremos una descripción de los resultados significativos de este trabajo de investigación. Comenzando con las observaciones tecnológicas, se encuentra que el cálculo de la operación para elevar al cuadrado y la extracción de raíces cuadradas se ven beneficiadas mediante el uso de instrucciones SIMD al consumir sólo unos cuantos ciclos de reloj.

En [10] se implementó la multiplicación escalar en la curva K-283, obteniendo 386×10^3 ciclos de reloj, prescindiendo del multiplicador sin acarreo. Resultado de esta investigación, el multiplicador de campo fue desarrollado utilizando la instrucción `PCLMULQDQ`, logrando acelerar el cómputo de la multiplicación escalar en un 60 % ante una implementación carente de esta operación.

Los resultados obtenidos muestran que el cálculo de la multiplicación escalar puede ser paralelizado eficientemente, logrando ahorros del 31.4 %, 36.7 % y 41.7 % para 112, 128 y 192 bits de seguridad, respectivamente, con respecto a la versión secuencial.

Bajo el contexto definido en la sección 3.1, la aceleración obtenida por el algoritmo paralelo fue de 1.45, 1.58 y 1.71 para los tres niveles de seguridad.

Como se puede observar, se obtuvo mayor aceleración en el campo $\mathbb{F}_{409}$. Esto indica que la aceleración obtenida depende también del tamaño de la entrada del programa, confirmando la predicción teórica de la Ley de Gustafson (veáse sección 3.1.2).

Desde el punto de vista algorítmico, se encontró que la operación que realiza elevadas al cuadrado consecutivas, mediante tablas de consulta, incrementó el rendimiento de la inversión de elementos en el grupo multiplicativo $\mathbb{F}^*_{2^m}$. Esta operación es bastante rápida, puesto que sólo toma el equivalente a 1.12 multiplicaciones en el campo $\mathbb{F}_{2^{233}}$.

Debido al uso del multiplicador sin acarreo, la razón (op/*M*) de la multiplicación de campo

con respecto a las demás operaciones de campo fue reducida, lo cual significa que operaciones que antes se consideraban desdeñables (como en el caso de las elevadas al cuadrado) ahora representan porcentajes significativos del tiempo de cómputo de la multiplicación de campo.

La multiplicación escalar en su versión secuencial fue implementada mediante el algoritmo de izquierda a derecha y el algoritmo de derecha a izquierda, los cuales son métodos similares excepto por el orden en que analizan los coeficientes de la expansión $\omega\tau$-NAF. Sin embargo, el método de derecha a izquierda ofrece un mejor rendimiento, ya que el cálculo de la expansión $\omega\tau$-NAF puede ser entrelazado con el ciclo principal. Como resultado de este entrelazado de operaciones, se pudo explotar el paralelismo a nivel de instrucción al utilizar diferentes recursos del procesador, lo que favorece a los programas secuenciales.

Resumen

En este capítulo se argumentó la elección de las curvas de Koblitz K-233, K-283 y K-409, las cuales ofrecen 112, 128 y 192 bits de seguridad respectivamente, de acuerdo a la recomendación del NIST.

Se detallaron los algoritmos para el desarrollo de las operaciones básicas del campo $\mathbb{F}_{2^m}$. La elevación al cuadrado y la extracción de raíces cuadradas fueron implementadas eficientemente utilizando la representación fraccionaria, esta representación favorece el uso del conjunto de funciones *Intrinsics*.

A lo largo de este capítulo, se examinó la construcción mediante el algoritmo de Karatsuba de la multiplicación de campo, utilizando el multiplicador sin acarreo provisto en el conjunto de instrucciones `AES-NI`.

Se introduce la operación que calcula elevaciones al cuadrado consecutivas, para realizar esta operación eficientemente, se utilizan tablas que almacenan elementos de campo, haciendo que esta operación sea computada fácilmente. Sin embargo, el tamaño de las tablas es considerable si se desea implementar en dispositivos restringidos en memoria.

La implementación multinúcleo de la multiplicación escalar fue realizada mediante dos técnicas de paralelización: $(\tau|\tau)$-y-suma y $(\tau^{-1}|\tau)$-y-suma.

La primera aproximación hace uso de la función que realiza elevaciones al cuadrado consecutivas para un valor fijo, esta operación puede ser calculada eficientemente si se cuenta con la memoria suficiente para almacenar las tablas de pre-cómputo.

La segunda aproximación está basada en aplicaciones de los operadores τ y τ^{-1}. Debido a la diferencia entre el tiempo de cómputo de estos operadores, el particionamiento no se realiza

exactamente por la mitad.

Finalmente, se exponen los resultados de las pruebas de rendimiento sobre las arquitecturas Westmere y Sandy Bridge para el cálculo de la multiplicación escalar en curvas de Koblitz. Con el fin de verificar los resultados obtenidos, se realizó una comparativa con otras implementaciones que ofrecen 128 bits de seguridad.

Curva elíptica	Campo	10^3 ciclos de reloj	Referencia
K-283	$\mathbb{F}_{2^{283}}$	386.0	[10]
curve25519	$\mathbb{F}_{2^{255}-19}$	307.0	[32]
surf127eps	$\mathbb{F}_{2^{127}-735}$	279.0	[32]
K-283	$\mathbb{F}_{2^{283}}$	136.0	Este trabajo
Jac128gls4	$\mathbb{F}_{(2^{128}-40557)^2}$	122.0	[43]

Tabla 5.11: Implementaciones de la multiplicación escalar en diversas curvas elípticas a 128 bits de seguridad.

Implementación	Curva elíptica de Koblitz	Nivel de seguridad	Plataforma	Tiempo (μs)
Järvinen, Skyttä [46]	K-163	80	Stratix II, 23,346 ALMs	28.95
			Stratix II, 13,472 ALMs	20.28
Järvinen, Skyttä [47]			Stratix II, 26,148 ALMs	4.91
Lutz y Hasan [60]			Virtex E, 10,017 LUTs	75
Ahmadi *et al.* * [6]	K-233	112	Virtex 2, 15,916 Slices	7.22
Este trabajo			i7-2600K @3.4GHz †	13.67
Este trabajo	K-283	128	i7-2600K @3.4GHz †	27.94
	K-409	192		43.76

* El área y tiempo invertido en calcular la expansión $\omega\tau$ NAF no fue incluído.

† Implementación multinúcleo.

Tabla 5.12: Comparación entre implementaciones utilizando dispositivos reconfigurables y los resultados de este trabajo.

6

Conclusiones

Estamos en este mundo para convivir en armonía. Quienes lo saben no luchan entre sí.

Siddhártha Gautama

Este capítulo expone el análisis de resultados obtenidos. Se da al lector una perspectiva de las contribuciones y las metas alcanzadas del trabajo de investigación realizado. Por último son evaluadas las deducciones por parte del autor para el desarrollo de futuras investigaciones.

SECCIÓN 6.1

Análisis

Uso de instrucciones SIMD. Se analizaron los conjuntos extendidos de instrucciones, entre las que destacan las instrucciones SIMD, contenidas en las arquitecturas de computadoras actuales.

Junto con la inclusión de nuevas instrucciones en las arquitecturas, se añadieron también registros de mayor tamaño que los nativos de la arquitectura. Estos registros fueron destinados a la ejecución de instrucciones SIMD.

Las instrucciones SIMD explotan el paralelismo a nivel de datos, lo cual fue de gran utilidad al calcular operaciones sobre valores almacenados en vectores de datos. Tal es el caso de la representación de los elementos de los campos finitos binarios. En donde, las operaciones aritméticas, tales como la suma, la raíz cuadrada y la elevación al cuadrado son ejemplos notorios del beneficio de este enfoque de programación.

Uso del multiplicador sin acarreo. La inclusión del conjunto de instrucciones `AES-NI` estableció un hito en las implementaciones con fines criptográficos. Puesto que ahora, el algoritmo de cifrado AES es posible realizarlo utilizando instrucciones incluidas en el procesador.

Así como también, en el caso de la instrucción `PCLMULQDQ`, la cual fue utilizada exitosamente para el cálculo de la multiplicación polinomial en el campo binario.

La contribución de este trabajo consiste en la aceleración del cómputo de la multiplicación escalar. En esta implementación, se logró realizar la multiplicación de dos elementos, en el campo $\mathbb{F}_{2^{233}}$, en tan sólo 100 ciclos de reloj. Esto es 2.4 veces más rápido que el mejor multiplicador de campo existente, presentado en [10], al inicio de este trabajo de tesis.

Multiplicación escalar en curvas elípticas de Koblitz. Con el objetivo de acelerar el cómputo de la multiplicación escalar en curvas de Koblitz, fue indispensable indagar cuáles son los mejores algoritmos que realicen esta operación de forma eficiente. Los algoritmos propuestos por Solinas en [73; 74] fueron la mejor opción, ya que aprovechan el endomorfismo de Frobenius presente en las curvas.

La implementación de la reducción parcial fue beneficiada por el uso de instrucciones vectoriales, puesto que involucra operaciones simples como desplazamientos de bits, sumas y restas de números enteros. La operación que redondea los datos de punto flotante fue emulada mediante números enteros, evitando de esta forma que se involucren operaciones de punto flotante.

En el algoritmo de multiplicación escalar, utilizando las expansiones $\omega\tau$-NAF de longitud l, se realizan en promedio $\frac{l}{\omega+1}$ sumas de punto. Si se incrementa el parámetro ω se acelera el cómputo

del ciclo principal de la multiplicación escalar. Sin embargo, esto aumenta el costo de calcular los representantes de congruencia. Resultado de esta implementación, se encontró que $\omega = 5$ presenta los mejores tiempos de cómputo para el cálculo de la multiplicación escalar.

La aceleración obtenida en la multiplicación escalar debido al reemplazo de los doblados de puntos por aplicaciones del endomorfismo de Frobenius es bastante significativa. En este trabajo, se observó que la proporción entre realizar un doblado de puntos y aplicar el endormorfismo de Frobenius es de alrededor 11 veces más rápida en los campos $\mathbb{F}_{2^{233}}$ y $\mathbb{F}_{2^{283}}$, y 7 veces más rápida en el campo $\mathbb{F}_{2^{409}}$. Esto comprueba un ahorro significativo en el cálculo de la multiplicación escalar al utilizar las curvas de Koblitz.

Multiplicación escalar de derecha a izquierda. En el caso particular de la implementación de la multiplicación escalar utilizando el algoritmo de derecha a izquierda, se confirmó el siguiente hecho: la ejecución del cálculo de la expansión $\omega\tau$-NAF puede ser entrelazada con el ciclo principal de la multiplicación escalar, lo cual reduce el número de ciclos de reloj que toma la ejecución de la multiplicación escalar. Este fenómeno ocurre al mezclar instrucciones que utilizan diferentes recursos del procesador, tales como instrucciones de aritmética entera y las instrucciones vectorizadas, ambas son mutuamente independientes.

Un análisis más detallado condujo a evaluar el paralelismo a nivel de instrucción, dado que las arquitecturas modernas poseen mecanismos que favorecen la ejecución en paralelo de instrucciones independientes de un programa secuencial. Entre los que se encuentran: el modelo de tubería, el predictor de ramificaciones y la ejecución fuera de orden.

La existencia de dependencias de datos entre instrucciones es un factor clave en el desarrollo de programas eficientes. Es, por lo tanto, recomendable evitar tanto como sea posible caer en esta situación, con el fin de evitar cuellos de botella en la ejecución del programa.

Paralelización de la multiplicación escalar. En este trabajo se tomó como punto de partida la propuesta de Ahmadi *et. al.* presentada en [6], la cual realiza el cómputo de la multiplicación escalar en paralelo, (algoritmo $(\tau^{-1}|\tau)$-y-suma). Se toma ventaja de la poca diferencia entre el cómputo de los operadores τ y τ^{-1}.

Por otro lado, como parte de esta investigación, se retomó el algoritmo $(\tau|\tau)$-y-suma, el cual no era hasta entonces posible realizarlo de forma eficiente, dado que el operador τ^n, con $n \in \mathbb{Z}$, resultaba muy costoso.

Una de las aportaciones de este trabajo fue la adecuación de este operador presentado en [13], el cual calcula τ^n utilizando tablas de consulta. En esta implementación, se obtuvo que el cálculo de τ^n, para un valor n fijo, se puede computar en aproximadamente 1.5 multiplicaciones de cam-

po. El resultado de esta adecuación, hizo factible el uso de esta técnica en la paralelización de la multiplicación escalar en curvas de Koblitz.

Programación en procesadores multinúcleo. Las arquitecturas de computadoras que cuentan con múltiples unidades de procesamiento, o núcleos, son atractivas para la ejecución de programas en paralelo. En esta implementación, los algoritmos paralelos que calculan la multiplicación escalar fueron implementados obteniendo factores de aceleración cercanos a 2.

Se pudo demostrar el beneficio de los procesadores multinúcleo, al acelerar la ejecución de la multiplicación escalar, lo que permite realizar protocolos criptográficos utilizando una computadora de escritorio, una computadora portátil o un servidor que cuente con un procesador multinúcleo.

SECCIÓN 6.2

Trabajo a futuro

En esta sección se exponen algunos apsectos que han quedado como trabajo a futuro luego del resultado de esta investigación.

- La búsqueda de una curva elíptica de Koblitz que ofrezca 128 bits de seguridad, cuya extensión de campo sea menor a 256 bits. Si se cuenta con dicha curva, se puede predecir que los tiempos de cómputo de la aritmética de curvas elípticas estarían apenas por encima de los tiempos obtenidos para la curva K-233.
- La paralelzación de la multiplicación escalar en K-283 mediante aplicaciones del operador inverso de Frobenius, es viable si se cuenta con un polinomio irreducible que sea ameno a la extracción de raíces cuadradas, lo cual disminuye el costo de la reducción polinomial.
- Es posible reducir el tiempo de la multiplicación escalar utilizando la idea del algoritmo $(\tau|\tau)$-y-suma, al realizar el cómputo en un sólo núcleo de procesamiento. Al prodecer de esta forma es posible realizar el cómputo de la multiplicación escalar compartiendo las aplicaciones del operador τ utilizando el método de *Interleaving* expuesto en la sección 3.3.3 de [40].
- Realizar un particionamiento en cuatro núcleos de procesamiento, de tal forma que la aceleración de la multiplicación escalar sea cercana a un factor de 4. Esto tomará ventaja cuando el tiempo de creación y sincronización de hilos sea mejorado por las nuevas arquitecturas.

Referencias

[1] Recommended elliptic curves for federal government use, 1999. Available in http://csrc.nist.gov/encryption.

[2] *GCC Online Documentation*, November 2011. http://gcc.gnu.org/onlinedocs/.

[3] *Intel® Advanced Vector Extensions Programming Reference*, June 2011. http://software.intel.com/file/36945.

[4] Leonard M. Adleman and Ming-Deh A. Huang. Function field sieve method for discrete logarithms over finite fields. *Information and Computation*, 151(1-2):5–16, 1999.

[5] Tilak Agerwala and John Cocke. High performance reduced instruction set processors. Technical Report 12434, IBM Thomas watson Research Center, Yorktown Heights, New York, March 1987.

[6] Omran Ahmadi, Darrel Hankerson, and Francisco Rodríguez-Henríquez. Parallel formulations of scalar multiplication on koblitz curves. *Journal of Universal Computer Science*, 14(3):481–504, feb 2008. http://www.jucs.org/jucs_14_3/parallel_formulations_of_scalar.

[7] Juan Manuel Cruz Alcaraz and Francisco Rodríguez-Henríquez. Multiplicación escalar en curvas de koblitz: Arquitectura en hardware reconfigurable. Master's thesis, Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, November 2005.

[8] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[9] ANSI. *ANSI X9.62:2005: Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)*. American National Standards Institute, pub-ANSI:adr, 2005.

[10] Diego F. Aranha, Julio López, and Darrel Hankerson. Efficient software implementation of binary field arithmetic using vector instruction sets. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT*, volume 6212 of *Lecture Notes in Computer Science*, pages 144–161. Springer, 2010.

[11] Roberto Avanzi. Another look at square roots (and other less common operations) in fields of even characteristic. In Carlisle Adams, Ali Miri, and Michael Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 138–154. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-77360-3_10.

[12] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`, accesed 20 November 2011.

[13] Joppe W. Bos, Thorsten Kleinjung, Ruben Niederhagen, and Peter Schwabe. ECC2K-130 on cell CPUs. In Daniel J. Bernstein and Tanja Lange, editors, *AFRICACRYPT*, volume 6055 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2010.

[14] B. Brumley and K. Järvinen. Koblitz curves and integer equivalents of Frobenius expansions. In *Selected Areas in Cryptography*, pages 126–137. Springer, 2007.

[15] Alice Chan. Intel® Compilers. Online resource. `http://software.intel.com/en-us/articles/intel-compilers`.

[16] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[17] Intel Corporation. Product Specifications Processor Corei5 660K. `http://ark.intel.com/products/43550/Intel-Core-i5-660-Processor-(4M-Cache-3_33-GHz)`.

[18] Intel Corporation. Product Specifications. Processor Corei7 2600K. `http://ark.intel.com/products/52214/Intel-Core-i7-2600K-Processor-(8M-Cache-3_40-GHz)`.

[19] Intel Corporation. Intel Hyper-Threading Technology, Technical User's Guide, January 2003. `http://cache-www.intel.com/cd/00/00/01/77/17705_htt_user_guide.pdf`.

[20] Intel Corporation. Prescott new instructions software developer's guide, January 2004. http://software.intel.com/file/21824.

[21] Intel Corporation. Intel architecture software developer's manual, volume 1: Basic architecture, 2007. http://developer.intel.com/design/index.htm.

[22] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[23] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(5):644–654, 1976.

[24] FIPS. Advanced encryption standard (AES), November 2001.

[25] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New frontiers in performance improvement and energy efficiency. White paper. `http://software.intel.com/`.

[26] Joseph A. Fisher. Very long instruction word architectures and the eli-512. *SIGARCH Comput. Archit. News*, 11:140–150, June 1983.

[27] A. Murat Fiskiran and Ruby B. Lee. Fast parallel table lookups to accelerate symmetric-key cryptography. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I - Volume 01*, pages 526–531, Washington, DC, USA, 2005. IEEE Computer Society.

[28] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[29] A. Fog. Instruction tables: List of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, March 2001.

[30] Kenny Fong, Darrel Hankerson, Julio Lopez, and Alfred Menezes. Field inversion and point halving revisited. *IEEE Transactions on Computers*, 53:1047–1059, 2003.

[31] Robert Gallant, Robert Lambert, and Scott Vanstone. Improving the parallelized pollard lambda search on anomalous binary curves. *Math. Comput.*, 69:1699–1705, October 2000.

[32] Pierrick Gaudry and Emmanuel Thomé. The mpfq library and implementing curve-based key exchanges, October 09 2007.

[33] Fayez Gebali. *Algorithms and Parallel Computing*. John Wiley & Sons, Inc., 2011.

[34] Jorge Guajardo and Christof Paar. Itoh–Tsujii inversion in standard basis and its application in cryptography and codes. *Des. Codes Cryptogr*, pages 207–216, 2002.

[35] Shay Gueron. Intel's new aes instructions for enhanced performance and security. In Orr Dunkelman, editor, *Fast Software Encryption*, volume 5665 of *Lecture Notes in Computer Science*, pages 51–66. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-03317-9_4.

[36] Shay Gueron. Intel® advanced encryption standard (AES) instructions set, January 2010.

[37] Shay Gueron and Michael Kounavis. Efficient implementation of the galois counter mode using a carry-less multiplier and a fast reduction algorithm. *Information Processing Letters*, 110(14-15):549–553, 2010.

[38] Shay Gueron and Michael E. Kounavis. *Carry-less multiplication and its usage for computing the GCM mode*. white paper, Intel Corporation, 2008.

[39] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.

[40] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[41] Martin E. Hellman and Justin M. Reyneri. Fast computation of discrete logarithms in gf(q). In *Advances in Cryptology: Proceedings of CRYPTO '82*, pages 3–13, 1982.

[42] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[43] Zhi Hu, Patrick Longa, and Maozhi Xu. Implementing 4-dimensional glv method on gls elliptic curves with j-invariant 0. Cryptology ePrint Archive, Report 2011/315, 2011. `http://eprint.iacr.org/2011/315`.

[44] Intel. Using streaming SIMD extensions (SSE2) to perform big multiplications, July 2000. http://software.intel.com/file/24753.

[45] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in gf(2m) using normal bases. *Inf. Comput.*, 78:171–177, September 1988.

[46] K. Jarvinen and J. Skytta. On parallelization of high-speed processors for elliptic curve cryptography. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(9):1162–1175, sept. 2008.

[47] Kimmo Järvinen and Jorma Skyttä. Fast point multiplication on koblitz curves: Parallelization method and implementations. *Microprocess. Microsyst.*, 33:106–116, March 2009.

[48] Jr Joseph D. Wieber and Gary M. Zoppetti. How to use intrinsics, December 2008. `http://software.intel.com/en-us/articles/how-to-use-intrinsics`.

[49] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *SIGARCH Comput. Archit. News*, 17:272–282, April 1989.

[50] David Kahn. *The Codebreakers*. The Macmillan Company, New York, 1967. xvi + 1164 pages.

[51] Anatoly A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963. URL: `http://cr.yp.to/bib/entries.html#1963/karatsuba`.

[52] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[53] Neal Koblitz. *A course in number theory and cryptography*. Graduate Texts in Mathematics, Vol. 114. Springer-Verlag New York, Inc., New York, NY, USA, 2 edition, 1987.

[54] Neal Koblitz. CM-curves with good cryptographic properties. *Lecture Notes in Computer Science*, 576:279–287, 1991.

[55] Tanja Lange. Koblitz curve cryptosystems. *Finite Fields and Their Applications*, 11(2):200–229, 2005.

[56] Tanja Lange and Igor Shparlinski. Collisions in fast generation of ideal classes and points on hyperelliptic and elliptic curves. *Appl. Algebra Eng. Commun. Comput*, 15(5):329–337, 2005.

[57] MSDN Library. Compiler intrinsics. Online resource. `http://msdn.microsoft.com/en-us/library/26td21ds.aspx`.

[58] Chris Lomont. Introduction to x64 Assembly, August 2010. `http://software.intel.com/en-us/articles/introduction-to-x64-assembly`.

[59] Lopez and Dahab. Improved algorithms for elliptic curve arithmetic in GF(2^n). In *SAC: Annual International Workshop on Selected Areas in Cryptography*. LNCS, 1998.

[60] J. Lutz and A. Hasan. High performance fpga based elliptic curve cryptographic co-processor. In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 2, pages 486–492 Vol.2, april 2004.

[61] Meier and Staffelbach. Efficient multiplication on certain nonsupersingular elliptic curves. In *CRYPTO: Proceedings of Crypto*, 1992.

[62] Ralph C. Merkle. Secure communications over insecure channels. *Commun. ACM*, 21(4):294–299, April 1978.

[63] Victor S Miller. Use of elliptic curves in cryptography. In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[64] Peter L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, January 1987.

[65] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. Comput.*, 33:968–976, November 1984.

[66] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over and its cryptographic significance (corresp.). *Information Theory, IEEE Transactions on*, 24(1):106–110, jan 1978.

[67] J. M. Pollard. Monte carlo methods for index computation $(\mathrm{mod}\, p)$. *Mathematics of Computation*, 32(143):918–924, July 1978.

[68] B. Ramakrishna Rau and Joseph A Fisher. Instruction-level parallel processing: History, overview, and perspective. In B. R. Rau and J. A. Fisher, editors, *Instruction-Level Parallelism*, volume 235 of *The Kluwer International Series in Engineering and Computer Science*, pages 9–50. Springer US, 1993. 10.1007/978-1-4615-3200-2_3.

[69] R. L. Rivest, A. Shamir, and L. Adelman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[70] Daniel Shanks. Class number, a theory of factorization, and genera. In *1969 Number Theory Institute (Proc. Sympos. Pure Math., Vol. XX, State Univ. New York, Stony Brook, N.Y., 1969)*, pages 415–440. Providence, R.I., 1971.

[71] D. Sheinwald, J. Satran, P. Thaler, and V. Cavanna. Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations. RFC 3385 (Informational), September 2002.

[72] Joseph H. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate Texts in Mathematics, Vol. 106*. Springer, 2 edition, 2009.

[73] Jerome A. Solinas. An improved algorithm for arithmetic on a family of elliptic curves. In Burton S. Kaliski Jr., editor, *Advances in Cryptology—CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 357–371. Springer-Verlag, 17–21 August 1997.

[74] Jerome A. Solinas. Efficient arithmetic on koblitz curves. *Des. Codes Cryptography*, 19(2/3):195–249, 2000.

[75] Shreekant (Ticky) Thakkar and Tom Huff. Internet streaming SIMD extensions. *Intel Technology Journal*, 32:26–34, December 1999.

[76] Stefan Tillich and Johann Großschädl. Instruction set extensions for efficient aes implementation on 32-bit processors. In *In Cryptographic Hardware and Embedded Systems — CHES 2006*, pages 270–284. Springer Verlag, 2006.

[77] Stefan Tillich, Johann Großschädl, and Alexander Szekely. An instruction set extension for fast and memory-efficient aes implementation. In *COMMUNICATIONS AND MULTIMEDIA SECURITY — CMS 2005*, pages 11–21. Springer Verlag, 2005.

[78] Stefan Tillich and Christoph Herbst. Boosting aes performance on a tiny processor core. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA conference on Topics in cryptology*, CT-RSA'08, pages 170–186, Berlin, Heidelberg, 2008. Springer-Verlag.

[79] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 2003.

[80] Lawrence C. Washington and Wade Trappe. *Introduction to Cryptography: With Coding Theory*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2002.

A

Herramientas de programación

SECCIÓN A.1

Lenguajes de programación

En esta sección describiremos los lenguajes de programación utilizados en la implementación multinúcleo de la multiplicación escalar. Con el fin de realizar una implementación eficiente, la tendencia converge en utilizar un lenguaje de bajo nivel, el cual permita mayor control sobre el procesamiento de las instrucciones.

Ensamblador. El lenguaje ensamblador es considerado como un lenguaje de bajo nivel, debido a la poca abstracción entre este lenguaje y el lenguaje máquina, ocasionando que el lenguaje ensamblador esté estrechamente ligado a la arquitectura destino; por ende, es a menudo utilizado para acelerar el rendimiento de las partes críticas de un programa.

El nombre genérico `x64` es utilizado para referirse a las extensiones de 64 bits de las arquitecturas de Intel® y AMD®, las cuales reemplazaron a las arquitecturas de 32 bits precedentes. En [58] se describe un panorama general de la arquitectura y de las instrucciones que posee.

En esta implementación se utilizaron instrucciones en ensamblador de `x64` para realizar operaciones aritméticas con números enteros de magnitudes arbitrarias. En este caso, la magnitud en bits de los operados involucrados es mayor al tamaño de los registros existentes en la arquitectura.

Lenguaje de programación C. Es sumamente conocido que el lenguaje de programación C es ampliamente utilizado para el desarrollo de sistemas de operativos, controladores de dispositivos y

Conjunto de instrucciones	Archivo de encabezado
SSE	xmmintrin.h
SSE2	emmintrin.h
SSE3	pmmintrin.h
SSSE3	tmmintrin.h
SSE4.2	smmintrin.h
AES-NI	wmmintrin.h
AVX	immintrin.h

Tabla A.1: Archivos de encabezado para utilizar los conjuntos extendidos de instrucciones.

aplicaciones de propósito general. El lenguaje C tiene la capacidad de crear complejas estructuras de datos a partir de tipos básicos.

A diferencia de los lenguajes interpretados, C ofrece ejecuciones más rápidas, gracias a las optimizaciones de los compiladores existentes. Las implementaciones realizadas en el presente trabajo de investigación consiste en más del 70 % de código escrito en C.

Intrinsics. Las *Intrinsics* son funciones que encapsulan los conjuntos extendidos de instrucciones SSE de los procesadores. Estas funciones son compatibles con la sintaxis del lenguaje C, lo cual facilita su utilización al programar.

Cuando un programa codificado mediante funciones *Intrinsics* se compila, el compilador reconoce los llamados a las funciones *Intrinsics*, los cuales son reemplazados por las instrucciones en ensamblador correspondientes a los conjuntos de instrucciones SSE. El compilador realiza automáticamente la asignación de registros `XMM` o `YMM`, las operaciones SIMD involucradas y finalmente su almacenamiento en memoria.

Para hacer uso de *Intrinsics*, es necesario incluir la biblioteca de desarrollo que contiene las definiciones del conjunto de instrucciones, en la tabla A.1 se enlistan los archivos de encabezado del lenguaje C que agregan soporte de *Intrinsics*.

Existen tipos de datos compatibles con el lenguaje C que encapsulan los registros `XMM` y `YMM`, éstos están definidos en los archivos de encabezado de las instrucciones SSE. Los tipos de datos son:

- `__m128`: Registro con 4 datos de tipo punto flotante de 32 bits cada uno.
- `__m128d`: Registro con 2 datos de tipo punto flotante de doble precisión de 64 bits cada uno.

- `__m128i`: Registro con 1, 2, 4 o 8 datos de tipo entero de 128, 64, 32 o 16 bits respectivamente.
- `__m256`: Registro con 8 datos de tipo punto flotante de 32 bits cada uno.
- `__m256d`: Registro con 4 datos de tipo punto flotante de doble precisión de 64 bits cada uno.
- `__m256i`: Registro con 1, 2, 4, 8 o 16 datos de tipo entero de 256, 128, 64, 32 o 16 bits respectivamente.

El mnemónico de las funciones *Intrinsics* permite fácilmente recordar la instrucción y el tipo de dato que emplea. He aquí la estructura básica de ellas: `_mm_<nom>_<type>`. El prefijo `_mm` está siempre presente en las instrucciones. La segunda parte `<nom>` corresponde al nombre de la operación, la cual puede ser el nombre de una instrucción en ensamblador. La última parte, `<type>` indica el tipo de registro a utilizar:

- `epi`X: indica que la operación se realizará sobre múltiples datos de tipo entero de X bits cada uno.
- `pd`: indica que la operación se realizará en múltiples datos de tipo punto flotante de 32 bits cada uno.
- `pd`: indica que la operación se realizará en múltiples datos de tipo punto flotante de doble precisión de 64 bits cada uno.
- `si`X: indica que la operación se realizará en un dato de tipo entero de X bits de magnitud.

Para mayor información sobre las funciones *Intrinsics* consúltese [48; 57].

B

Compiladores

Los compiladores juegan un rol importante previo a la ejecución del programa, ya que son los encargados de traducir el código fuente a código máquina. A lo largo de los años, diversos compiladores e intérpretes de instrucciones han sido desarrollados. En este trabajo se utilizaron dos de los compiladores más utilizados contemporáneamente, tal es el caso de ICC y GCC, los cuales se describen a continuación:

ICC. El compilador desarrollado por Intel®, llamado Intel® C++ Compiler, disponible en [15], compila código en lenguaje C y C++ y es compatible con *Intrinsics*.

Provee tres niveles de optimización de código, entre las que se encuentran: la optimización entre procedimientos, el desenrollado de ciclos (*loop-unrolling*) y optimizaciones en el uso de la memoria caché. Cabe indicar que el tiempo de compilación aumenta a mayor grado de optimización deseado.

En este trabajo se utilizó la versión 12.0 de uso no-comercial. Es posible utilizar también otras herramientas que Intel® provee en su versión comercial, tales como analizadores de código e inspectores de ejecución, los cuales muestran el rendimiento de un programa así como sus secciones críticas.

GCC. El conjunto de compiladores creados por el proyecto de código abierto GNU, llamado `gcc` (del inglés *GNU C Compiler*), es capaz de producir código para procesadores con instrucciones SIMD. Soporta mediante bibliotecas adicionales, el uso de OpenMP.

Posee mecanismos para realizar optimizaciones de alto nivel y produce código específico de acuerdo a la arquitectura. La versión 4.6.1 fue utilizada para esta implementación y en [2] se puede encontrar la documentación completa de este compilador.

Durante el desarrollo del trabajo de investigación aparecieron nuevas versiones del compilador GCC, las cuales agregaban incrementalmente nuevas funcionalidades y optimizaciones. La versión 4.6 incluyó la capacidad de indicar al compilador la arquitectura donde se ejecutará el programa, de esta forma se producen las optimizaciones pertinentes.

Diferencias. El compilador GCC, por ser un programa de código abierto, es ampliamanete integrado en la mayoría de las distribuciones de Linux, esto ha producido que gran cantidad de desarrolladores se involucren en el proyecto, logrando así mejoras en el proceso de compilación y enlazado de programas.

No obstante, el compilador ICC produce código muy eficiente, puesto que toma ventaja de ser el diseñador y fabricante de la arquitectura. En algunas aplicaciones de cómputo de alto rendimiento, el compilador ICC produce programas que ahorran hasta en un 50 % el tiempo de ejecución comparado con compiladores convencionales.

En esta investigación se determinó utilizar ambos compiladores para realizar una comparativa del rendimiento de la implementación multinúcleo de la multiplicación escalar.

SECCIÓN B.1
OpenMP

OpenMP (del inglés *Open Multi-Processing*) es una interfaz de programación de código abierto que habilita la programación de múltiples procesos, los cuales se comunican mediante memoria compartida. En [16] se encuentran ampliamente descritos los detalles sobre la implementación de OpenMP.

Un lenguaje de programación en paralelo debe habilitar el soporte para tres aspectos básicos: especificar la ejecución secuencial y en paralelo del programa, permitir la comunicación entre los múltiples hilos de ejecución y finalmente poder expresar la sincronización entre los hilos.

El funcionamiento de OpenMP está basado en directivas que dan soporte al procesamiento en paralelo para lenguajes como C, C++ y Fortran. Existen dos beneficios principales del funcionamiento basado en directivas:

- Permite que el mismo código sea usado para el desarrollo de programas tanto en procesadores de un sólo núcleo, como en procesadores multinúcleo. Cuando se utiliza un sólo procesador las directivas son tratadas como comentarios y son ignoradas por el compilador.

- Habilita el desarrollo de programas en paralelo de forma incremental, es decir, dado un programa secuencial es posible adecuarlo mediante directivas que expresen la ejecución en para-

lelo.

La principal estructura utilizada en la paralelización de la multiplicación escalar es:

```
#pragma omp parallel {
   #pragma omp sections {
      #pragma omp section {
          funcion1()
      }
      #pragma omp section {
          funcion2()
      }
   }
}
```

La primer directiva `omp parallel` crea *N* hilos de ejecución, donde *N* es el número de núcleos de procesamiento que posee la computadora. Es posible especificar el número de hilos mediante la función: `omp_set_num_threads`.

La directiva `omp sections` distribuye el código enlistado en los fragmentos de código `omp section` a cada hilo de ejecución creado. Los bloques de código serán ejecutados en cada núcleo del procesador.

Al final se encuentra implícitamente una barrera de sincronización que espera a que todos los procesos hayan concluido. Se recomienda balancear la carga de procesamiento entre cada núcleo para evitar que los núcleos estén ociosos mientras los demás terminan su ejecución.

La implementación en paralelo usando la biblioteca de funciones OpenMP, permite acelerar el cómputo de la multiplicación escalar, logrando factores de aceleración cercanos a 2.

C

Lista de publicaciones

A continuación se enlistan las publicaciones por parte del autor como resultado de este trabajo de tesis:

Título	**Software Implementation of Binary Elliptic Curves: Impact of the Carry-Less Multiplier on Scalar Multiplication**
Autores	Jonathan Taverne[1] jonathan.taverne@etu.univ-lyon1.fr Armando Faz Hernández[2] armfaz@computacion.cs.cinvestav.mx Diego F. Aranha[3] dfaranha@ic.unicamp.br Francisco Rodríguez Henríquez[2] francisco@cs.cinvestav.mx Darrel Hankerson[4] hankedr@auburn.edu Julio López[3] jlopez@ic.unicamp.br
Afiliación de los autores	[1] Université de Lyon, Université Lyon1, ISFA, France [2] Computer Science Department, CINVESTAV-IPN, México [3] Institute of Computing, University of Campinas, Brazil [4] Auburn University, USA
DOI	10.1007978-3-642-23951-9_8
Conferencia	Cryptographic Hardware and Embedded Systems – CHES 2011 13th International Workshop, Nara, Japan, September 28 – October 1, 2011.
Editores	Bart Preneel y Tsuyoshi Takagi
Año	2011
ISBN	978-3-642-23950-2
Editorial	Springer Berlin / Heidelberg

Título	**Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction**
Autores	Jonathan Taverne[1] jonathan.taverne@etu.univ-lyon1.fr Armando Faz Hernández[2] armfaz@computacion.cs.cinvestav.mx Diego F. Aranha[3] dfaranha@ic.unicamp.br Francisco Rodríguez Henríquez[2] francisco@cs.cinvestav.mx Darrel Hankerson[4] hankedr@auburn.edu Julio López[3] jlopez@ic.unicamp.br
Afiliación de los autores	[1] Université de Lyon, Université Lyon1, ISFA, France [2] Computer Science Department, CINVESTAV-IPN, México [3] Institute of Computing, University of Campinas, Brazil [4] Auburn University, USA
DOI	10.1007/s13389-011-0017-8
Revista	Journal of Cryptographic Engineering
Volumen	Volumen 1, 2011
Colección	Computer Science
ISSN	2190-8508 (Impresa) 2190-8516 (En línea)
Editorial	Springer Berlin / Heidelberg

Índice alfabético