UNIVERSIDAD AUTÓNOMA METROPOLITANA
UNIDAD IZTAPALAPA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

GENERACIÓN DE SUCESIONES CRIPTOGRÁFICAMENTE FUERTES

TESIS QUE PRESENTA

SANDRA DÍAZ SANTIAGO

PARA LA OBTENCIÓN DEL GRADO DE

MAESTRIA EN CIENCIAS

ASESOR: DR. HORACIO TAPIA RECILLAS

JULIO DE 2005

UNIVERSIDAD AUTÓNOMA METROPOLITANA
UNIDAD IZTAPALAPA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

GENERACIÓN DE SUCESIONES CRIPTOGRÁFICAMENTE FUERTES

TESIS QUE PRESENTA

SANDRA DÍAZ SANTIAGO

PARA LA OBTENCIÓN DEL GRADO DE

MAESTRIA EN CIENCIAS

ASESOR: DR. HORACIO TAPIA RECILLAS

JULIO DE 2005

Agradecimientos

Agradezco el gran apoyo y cariño de mi familia: Aris, Isabel, Ricardo, Alfonso; y muy en especial de mis padres: Brígida y Rubén, por enseñarme a concluir todo lo que se empieza.

Agradezco a mi asesor el Dr. Horacio Tapia Recillas, por su enorme paciencia en el desarrollo del presente trabajo.

Agradezco también el apoyo de mis amigos Jazmín, Carlos, Julia, Verónica, Ulises, Josué, Luis, Paula; y muy en especial a Andrés por estar siempre al lado mio, ayudándome y brindándome su poesía visual.

Agradezco a la Dra. Shirley Bromberg, por impulsarme para concluir mis estudios de maestría y al Dr. Noé Gutiérrez Herrera por leer y corregir la versión inicial de éste escrito.

Por último pero no por ello menos importantes, agradezco a las secretarias del Departamento de Matemáticas: Michelle, Bety, Silvia y Lulú por su amabilidad y ayuda.

Índice general

Agradecimientos I

Introducción V

Notación IX

1. Conceptos Básicos **1**
1.1. Antecedentes 1
1.2. Clases de complejidad: $\mathcal{P}, \mathcal{NP}$. 3
1.3. Algoritmos probabilísticos 5
1.4. Pseudoaleatoriedad 6
1.5. Pruebas estadísticas de aleatoriedad 7

2. Generador de Congruencia Lineal (GCL) **11**
2.1. Obteniendo la máxima longitud de período 13
2.2. Generador puramente multiplicativo 18
2.3. Predicción de los parámetros del GCL 20

3. Generador Blum-Blum-Shub **31**
3.1. Conceptos básicos 31
3.2. Construcción del generador 35
3.2.1. Un par de ejemplos 41
3.3. Longitud de período 42

4. Generador Blum-Micali **47**
4.1. Conceptos básicos 47
4.1.1. Preliminares matemáticos 47
4.1.2. Familias de circuitos no uniformes 49
4.2. Predicados no aproximables 52

4.3. Construcción de un GBPCF 56
4.4. Construcción del generador Blum-Micali 57

5. Comentarios adicionales 63
5.1. Otras formas de generar sucesiones 63
5.1.1. Sucesiones generadas por hardware 64
5.1.2. Funciones de un sólo sentido 64
5.1.3. Curvas elípticas 65
5.2. Otras aplicaciones 66
5.2.1. Ampliación de Espectro 66
5.2.2. LFSR 67

6. Conclusiones 71
6.1. Resumen 71
6.2. Conclusiones obtenidas 72

A. Resultados Clásicos 75

B. Código fuente 81
B.1. Generador de congruencia lineal 81
B.2. Generador Blum-Blum-Shub 87
B.3. Generador Blum-Micali 90

Bibliografía 95

Introducción

La palabra criptografía viene del griego *cryptos* que significa **escondido** y *graphos* que significa **escritura**, es decir criptografía significa *escritura secreta*. Según David Kahn [KAH67], la criptografía surgió en Egipto hace aproximadamente 4000 años, cuando un escriba al realizar una inscripción en la tumba de su señor, utilizó algunos jeroglíficos poco comúnes, no con la finalidad de hacer ininteligible el texto, sino simplemente para imprimirle cierto grado de dignidad y autoridad. Posteriormente, las modificaciones hechas por los escribas se volvieron más complejas hasta otorgarles caracter de secreto a sus textos. Progresivamente, con la finalidad de mantener oculta determinada clase de información, comienza a expandirse lentamente el uso de la criptografía en diversos lugares. De esta manera civilizaciones como la India, Mesopotamia, Babilonia, Grecia, Roma, y Arabia, entre otras utilizaron métodos ampliamente conocidos hoy día, como la sustitución y la permutación en sus distintas variantes.

A lo largo de la historia, distintas entidades entre las cuales destacan los reyes, la iglesia, y los diplomáticos, desarrollan sus propios mecanismos criptográficos o de cifrado. Al mismo tiempo, aparecen las primeras técnicas de criptoanálisis, el cuál trata de hallar el mensaje en claro a partir de un mensaje cifrado, sin conocer el método que fue utilizado para cifrarlo. Un periodo en el cual la criptografía y el criptoanálisis crecen ampliamente, ocurre durante las dos guerras mundiales. No sólo se inventan nuevas técnicas de cifrado, sino que inclusive se construyen dispositivos mecánicos para agilizar el cifrado y el descifrado de la información. En este período, es importante mencionar a personajes como Alan Turing, William Friedmann y por supuesto Claude Shannon, entre otros. [KAH67, SIN99].

Sin embargo la criptografía continúa utilizándose con un sólo objetivo: co-

municarse de manera segura a través de un canal inseguro. Además se utiliza exclusivamente lo que se conoce como criptografía de clave secreta, es decir tanto para cifrar como para descifrar el mensaje se utilizaba la misma clave, lo cual tiene el inconveniente de requerir una clave por cada par de entidades que desean comunicarse. Y además el hecho de que para acordar la llave se requería el uso de un canal seguro. Ante tales problemas W. Diffie y M. E. Hellmann se dieron a la tarea de pensar en alguna forma de solucionarlos y es así que alrededor de 1976, publican el artículo que revolucionaría la criptografía [DIF76], en el cual proponen el uso de dos claves distintas, una para cifrar y otra para descifrar. De esta manera, algunos objetivos que persigue la seguridad de la información son los siguientes:

1. *Confidencialidad:* Se mantiene la información en secreto.
2. *Integridad:* Permite averiguar si la información ha sido modificada.
3. *Autenticación:* Se determina quien ha enviado la información.
4. *No repudio:* Permite que una entidad, no pueda negar o repudiar un acuerdo previamente establecido.

Al mismo tiempo la criptografía se define como una ciencia más formal en la que se echa mano de ramas de las matemáticas tales como: teoría de números, álgebra, estadística y probabilidad, entre otras. Y con el objetivo de crear algoritmos eficientes, también se estudia la complejidad de los algoritmos, rama importante de la computación teórica.

Dentro de la criptografía, existen diversos algoritmos para proporcionar tales servicios, los cuales buscan ser más eficientes, más rápidos, más seguros, o simplemente ofrecer alternativas a los diferentes usuarios. Para que dichos algoritmos tengan alguna o varias de tales características, es necesario enfrentarse a distintos problemas. Así, por ejemplo, para incrementar la rapidez de un algoritmo que realiza operaciones sobre campos finitos, se requiere averiguar cómo hacer aritmética rápida sobre dichos campos. Cuando se intenta diseñar una función hash [MEN97], ésta debe cumplir con determinadas características para no comprometer la seguridad de la aplicación criptográfica.

Pero aún cuando no se desee crear un nuevo algoritmo o mejorarlo, sino simplemente implementarlo, surgen dificultades. Por ejemplo, si se quiere

implementar el RSA, uno de los algoritmos de cifrado de clave pública más usados actualmente [RIV78], al generar las claves, el primer paso del algoritmo dice: *Genere dos primos al azar p y q del mismo tamaño.* Entonces uno se pregunta ¿cómo verifico si un número es primo o no?, más aún, ¿cómo los genero al azar? La respuesta a la primera pregunta se resuelve estudiando las pruebas de primalidad que existen [KRA86, MEN97, STI95], las cuales se podrán aplicar al número candidato. La segunda pregunta, aunque aparentemente sencilla de responder, es la que atañe al presente trabajo.

Generar un número al azar podría reducirse a usar alguna función implementada en algún lenguaje de programación con dicha finalidad. Sin embargo, si uno desea evitarse problemas, debe preguntarse ciertas cosas como ¿qué significa el término *al azar*? ¿existe alguna característica que deban cumplir los números generados al azar? En efecto, una sucesión de números de la que se presume fue generada al azar, debe cumplir con ciertas cualidades, que se estudiarán más adelante. Cuando se trata de aplicaciones criptográficas la situación es todavía más seria, y se debe ser sumamente cuidadoso al generar sucesiones al azar, ya que la seguridad del sistema criptográfico dependerá en parte de la generación de cantidades aleatorias e impredecibles. Por ejemplo, el RSA basa su seguridad en el hecho de mantener en secreto los primos p y q mencionados anteriormente. Si no se tiene cuidado al generar las sucesiones de bits que representan a p y q, algún adversario podría hallar cuáles fueron las sucesiones utilizadas para ello, haciendo totalmente vulnerable al algoritmo de cifrado.

Existen dispositivos que funcionan como verdaderas fuentes de sucesiones aleatorias, pero la mayoría provienen de métodos físicos, los cuales tienden a ser muy caros o muy lentos. Para solucionar éstos problemas, se ha tratado de diseñar métodos para construir sucesiones pseudoaleatorias de una manera determinística; a partir de un valor inicial denominado *semilla*. Sin embargo, es importante que tales métodos sean estudiados concienzudamente, a través de las sucesiones generadas por ellos. Y más aún cuando su uso posterior sea en alguna aplicación criptográfica.

El presente trabajo tiene la finalidad de analizar detalladamente la construcción de dos generadores de sucesiones pseudoaleatorias para usos criptográficos los cuáles fueron introducidos por L. Blum, M. Blum, M. Shub y S. Micali a mediados de los 80's. Para exhibir la importancia que tiene el

no dejar al azar la generación de cantidades aleatorias, inicialmente se estudia el generador de congruencia lineal, el cual es absolutamente inseguro cuando de criptografía se trata. El trabajo está organizado en 6 capítulos, en el capítulo 1 se establecen las definiciones básicas de generadores, y otros conceptos necesarios para entender porqué algunos generadores son seguros para usos criptográficos y otros no. En el capítulo 2, se estudia el generador de congruencia lineal tanto su construcción como la forma de determinar sus paramétros a partir de una porción de la sucesión. Los capítulos 3 y 4 están basados en los trabajos de Blum, Blum, Shub [BLU86] y de Blum y Micali [BLU84] respectivamente. Puesto que existen otras técnicas para generar sucesiones pseudoaleatorias y éstas se usan en otras aplicaciones distintas de las criptográficas, en el capítulo 5, se trata de manera muy general ambos temas. Finalmente se incluyen dos apéndices en los cuales se definen algunos conceptos básicos de Teoría de Números y se agregan algunos programas utilizados durante la investigación.

Notación

$\lfloor x \rfloor$	*El más grande de los enteros, menor o igual que* x
(a, b)	*Máximo común divisor de* a *y* b
$[a, b]$	*Mínimo común múltiplo de* a *y* b
$x \vert y$	x *divide a* y
$\varphi(m)$	*Función Phi de Euler*
$\left(\frac{a}{n}\right)$	*Símbolo de Jacobi*
Q_n	*Conjunto de residuos cuadráticos módulo* n
J_n	*Conjunto de enteros módulo* n*, con símbolo de Jacobi igual a 1*
$\vert A \vert$	*Si* A *es un conjunto, denota la cardinalidad del mismo*
$\lambda(n)$	*Función lambda de Carmichael*
$\vert p \vert$	*Número de bits de* p *en expansión binaria*
$ord_n a$	*Orden de* a *módulo* n
$\mathbb{Z}$	*Anillo de los números enteros*
$\mathbb{Z}_n$	*Anillo de los enteros módulo* n
$a \parallel b$	a *concatenado con* b*, donde* a *y* b *son palabras binarias*

Capítulo 1

Conceptos Básicos

En el presente capítulo se introducirá la terminología y conceptos necesarios para entender la generación de números pseudoaleatorios y su importancia tanto en la criptografía como en otras áreas.

1.1. Antecedentes

Los números que se generan *aleatoriamente* son útiles en diversas situaciones, por ejemplo: simulación, análisis numérico, programación de computadoras, toma de decisiones y recreación entre otras.

Sin embargo ¿qué significa el término *aleatorio*?, esto es, ¿es posible decir si un número en particular, como el 5, es o no aleatorio? Es claro que no, por tal motivo es preferible hablar de *sucesiones de números aleatorios*, en donde cada número de la sucesión no tiene ninguna relación con el resto de los elementos de la misma. Informalmente, si cada elemento tiene la misma probabilidad de aparecer en la sucesión entonces se dice que ésta sucesión de números es aleatoria.

Pero a continuación aparece el problema de cómo obtener números "realmente" aleatorios. Generar sucesiones de tales números no es sencillo. Sin embargo a la fecha ha habido varios intentos por obtener sucesiones de números aleatorios, desde lanzar un dado hasta la construcción de dispositivos especiales. Por ejemplo según se menciona en [KNU81], la compañía RAND publicó un libro (1955) donde se listaba 1 millón de dígitos aleatorios, generados con el uso de un dispositivo especial. La forma en la que se generaban

tales números, era a través de procesos físicos, o de la mecánica cuántica entre otros. Pero, posteriormente se demostró que las sucesiones generadas de tal forma, no eran totalmente aleatorias.

Generar sucesiones aleatorias es bastante complicado, ya que éstas deben satisfacer ciertas propiedades y por eso, es deseable no *malgastarlas*. Por ello, se pensó en generar *sucesiones pseudoaleatorias*. Es decir, sucesiones cuyos elementos parecen generados al azar, pero que en realidad no lo son.

Además de las situaciones ya mencionadas en donde se usan sucesiones aleatorias, se tiene la aplicación en algoritmos criptográficos. Por ejemplo, en la generación de claves de mecanismos de cifrado tanto simétricos como asimétricos[1] [MEN97, STA03, STI95], se debe comenzar por generar una sucesión de 0's y 1's. Lo mismo ocurre al generar claves de sesión en protocolos de seguridad, como *Kerberos* [STE88, NEU94] y *, Secure Socket Layer (SSL)* [RFC99]. En tal circunstancia, se debe ser extremadamente cuidadoso, ya que escoger inadecuadamente la manera de generar sucesiones que van a ser utilizadas en algoritmos criptográficos, puede comprometer la seguridad de los mismos.

La cuestión de generar sucesiones aleatorias de acuerdo con [GOL99, GOL01], comenzó a estudiarla Claude E. Shannon, a la cual hace referencia en su famoso artículo *A mathematical theory of communication* [SHA48], en el que según su teoría, la aleatoriedad perfecta se logra en el caso extremo en el que no existe redundacia alguna en la información. Es decir, la aleatoriedad perfecta esta asociada a una distribución uniforme. Es claro que bajo esta teoría no es posible generar sucesiones que sean aleatorias, usando algoritmos determinísticos.

Posteriormente, el problema fue abordado por Solomonov [SOL64], Kolmogorov [KOL65] y Chaitin [CHA66] a principios de la década de los sesentas, quienes utilizan la teoría de la computabilidad y la noción de lenguaje universal. Se definía una cadena finita de bits $A = a_1 \cdots a_k$ como *Kolmogorov-aleatoria*, si la longitud de la entrada más corta para una máquina universal de Turing $\mathcal{T}$, la cual se detenía y cuya salida era la cadena A, es de longitud mayor o igual a k. Desafortunadamente, es un problema indecidible saber si una cadena A es o no Kolmogorov-aleatoria [LAG90].

[1] Los algoritmos de cifrado simétrico son aquellos que utilizan una única clave tanto para cifrar como para descifrar. A diferencia de los asimétricos, para los cuales se tienen 2 claves, una pública y otra privada, la primera de ellas se usa para cifrar y la segunda para descifrar.

Hacia 1980, Blum, Goldwasser, Micali y Yao [GOL01] analizan el problema de generar sucesiones, basándose en la teoría de la complejidad computacional. Bajo dicha teoría se afirma que una cadena de bits generada por medios determinísticos, es considerada aleatoria si ningún observador puede distinguir entre dicha cadena y una verdaderamente aleatoria, es decir, obtenida con una distribución uniforme. Esto es, dos objetos son considerados equivalentes si no es posible, con los recursos disponibles y de manera eficiente, hallar la diferencia entre ellos. Con este nuevo enfoque es posible entonces expandir una cadena de bits de longitud k, a una cadena de bits de longitud $l(k)$, con algún algoritmo determinístico. Esta será la forma de generar sucesiones pseudoaleatorias en el presente trabajo. A continuación se darán algunas definiciones más formales.

1.2. Clases de complejidad: $\mathcal{P}, \mathcal{NP}$.

Una cuestión importante al hablar de algoritmos es tratar de medir la eficiencia de alguno en particular, para ello se toma en cuenta la cantidad de recursos que se necesitarán, tales como el ancho de banda, la memoria y el tiempo. De particular importancia es éste último, pues en el caso de la criptografía, será crucial saber si el tiempo necesario para resolver ciertos problemas es demasiado, ya que justamente ese factor resulta crucial para definir la seguridad de un algoritmo de cifrado específico.

Al considerar la eficiencia para resolver un problema dado, se asumirá, a lo largo del presente trabajo, que un algoritmo se ejecuta en una máquina con un sólo procesador y que las instrucciones se ejecutan una tras otra, es decir, no hay operaciones concurrentes.

Para saber si un algoritmo es eficiente o no, éste debe ser analizado teniendo en cuenta el *tamaño de la entrada* y el *tiempo de ejecución* requerido para resolverlo. El *tamaño de la entrada* dependerá de las estructuras utilizadas, por ejemplo, si se utiliza un arreglo entonces el tamaño de la entrada será el número de elementos en el arreglo; pero si se lleva a cabo la multiplicación de dos números grandes, entonces el tamaño será el número de bits que posee cada uno de los números.

Ahora bien, el *tiempo de ejecución* de un algoritmo para una entrada específica es el *número de operaciones primitivas* o pasos para resolver el problema, y además se asume que el tiempo para realizar cada una de esas operaciones primitivas o pasos es constante, aunque en la realidad no es así,

sin embargo la diferencia es despreciable.

Al medir el tiempo de ejecución de esta manera y no empíricamente, es decir, llevando una estadística de cuanto se tarda un algoritmo específico, para n entradas con un procesador en particular, se obtiene una idea más clara del tiempo de ejecución, sin necesidad de probar el algoritmo con todas las entradas posibles y sin depender de la velocidad de un procesador. Ya que como bien se sabe dicha velocidad va aumentado con el paso del tiempo, haciendo que nuestras mediciones hechas de esa manera se vuelvan obsoletas. Se puede conocer con más detalle el análisis de algoritmos si se consulta [COR90].

Cuando se mide el tiempo de ejecución de la forma antes mencionada, es posible hablar del *orden o tasa de crecimiento* del tiempo de ejecución, para lo cual se utiliza la *notación asintótica*.

La *notación asintótica* es usada para describir el tiempo de ejecución de un algoritmo en términos de funciones cuyo dominio es el conjunto de los número naturales. Existen diversas notaciones $\Theta, \Omega, \omega, O, o$; sin embargo la más conocida de ellas es la notación O, que da una cota superior.

Definición 1.1 *Si $g : \mathbb{N} \to \mathbb{R}$, $O(g(n))$ denota al conjunto de funciones $f(n)$ que "estan por debajo" de $g(n)$ para toda $n \geq n_0$. Más formalmente*

$$O(g(n)) = \{f(n) | \exists\, c, n_0 \text{ positivos tales que } 0 \leq f(n) \leq c(g(n))\ \forall\, n \geq n_0\}$$

Por ejemplo, si al analizar un algoritmo se obtiene que para una entrada de tamaño n, el tiempo de ejecución en el peor de los casos, es un polinomio de la forma an^2+bn+c, se considerá unicamente el término de mayor grado y se podrá afirmar que la *complejidad del algoritmo* es $O(n^2)$, queriendo decir con ello que para cualquier otra entrada, la función correspondiente al tiempo de ejecución permanecerá siempre por debajo de la función cn^2, considerando las constantes c y n_0 como en la definición.

La mayoría de los algoritmos con entrada n tiene un tiempo de ejecución que cae dentro del orden $O(n^k)$ para $k \geq 0$, en el peor de los casos. A dichos algoritmos se les denomina *algoritmos con tiempo polinomial*, y al conjunto de problemas que son resueltos usando algoritmos con tiempo polinomial se les denota como $\mathcal{P}$. Sin embargo, no todos los problemas pueden ser resueltos en tiempo polinomial. Algunos de ellos pueden resolverse por algoritmos que requieren tiempo superpolinomial, es decir con $O(2^n), O(n!), O(n^n)$, a tales problemas se les denomina *intratables*, y el conjunto de los mismos se denota por $\mathcal{NP}$. Más aún, existen algunos problemas cuyo estatus es desconocido,

es decir, no se sabe si se pueden resolver o no en tiempo superpolinomial, a los cuales se les conoce como *NP-completos*. [COR90]

1.3. Algoritmos probabilísticos

En los párrafos anteriores se ha hablado exclusivamente de algoritmos *determinísticos*, es decir, aquellos que si reciben la misma entrada, dan exactamente los mismos resultados, puesto que ejecutan el mismo conjunto de instrucciones una y otra vez. Como hemos mencionado anteriormente, este tipo de algoritmos, tardan demasiado tiempo para resolver algunos problemas. Como una alternativa aparecen los algoritmos *probabilísticos*. Estos últimos a diferencia de los anteriores, realizan elecciones al azar en ciertos puntos de su ejecución, por lo que para una misma entrada, la salida puede ser diferente. Tal es el caso de los algoritmos para determinar si un número entero es o no primo. A continuación se mencionarán algunos conceptos básicos relacionados con este tipo de algoritmos, los cuales se utilizarán en capítulos posteriores, al hablar de los generadores criptográficamente fuertes.

Definición 1.2 *Sea L un problema abstracto de decisión, I una instancia y A un algoritmo probabilístico para resolver L entonces:*

1. *A tiene error lateral-0 si* $\mathsf{Pr[A\ responda\ SI\ |\ respuesta\ sea\ SI]} = 1$ *y* $\mathsf{Pr[A\ responda\ SI\ |\ respuesta\ sea\ NO]} = 0$.
2. *A tiene error lateral-1 si* $\mathsf{Pr[A\ responda\ SI\ |\ respuesta\ sea\ SI]} \geq \frac{1}{2}$ *y* $\mathsf{Pr[A\ responda\ SI\ |\ respuesta\ sea\ NO]} = 0$.
3. *A tiene error lateral-2 si* $\mathsf{Pr[A\ responda\ SI\ |\ respuesta\ sea\ SI]} \geq \frac{2}{3}$ *y* $\mathsf{Pr[A\ responda\ SI\ |\ respuesta\ sea\ NO]} \geq \frac{1}{3}$.

Los algoritmos probabilísticos también se clasifican de acuerdo a su complejidad de la siguiente forma. La clase de complejidad $\mathcal{ZPP}$ es el conjunto de los algoritmos del tipo *error lateral-0*, cuyo tiempo de ejecución es polinomial. La clase de complejidad $\mathcal{RPP}$ es el conjunto de algoritmos del tipo *error lateral-1*, que se ejecutan en el peor caso en tiempo polinomial. La clase de complejidad $\mathcal{BPP}$ es el conjunto de los algoritmos del tipo *error lateral-2*, que se ejecutan en el peor caso en tiempo polinomial.

1.4. Pseudoaleatoriedad

En ésta sección, se definirá en qué consiste un generador de bits aleatorios y un generador de bits pseudoaleatorios. También se mencionan algunas propiedades relacionadas con estos últimos, los cuales serán de suma importancia en el desarrollo del presente trabajo.

Definición 1.3 *Un* **generador de bits aleatorios** *es un algoritmo o dispositivo cuya salida es una sucesión estadísticamente independiente e impredecible de dígitos binarios, denominada sucesión aleatoria de bits.*

Definición 1.4 *Sean k, l, enteros positivos tales que $l \gg k$. Un generador de bits pseudoaleatorios (GBPA), es una función*

$$f : (\mathbb{Z}_2)^k \to (\mathbb{Z}_2)^l$$

con las siguientes características:

1. *f puede ser calculada en tiempo polinomial,*
2. *$s_0 \in (\mathbb{Z}_2)^k$ es una sucesión aleatoria de bits, a la cual se le denomina* **semilla**,
3. *$f(s_0) \in (\mathbb{Z}_2)^l$ denominada sucesión pseudoaleatoria de bits. A tal sucesión no es posible distinguirla de una sucesión aleatoria.*

El hecho de que una sucesión pase las pruebas estadísticas (que se explicarán en la sección 1.5), es una condición *necesaria* pero no *suficiente* para que un generador sea seguro, entendiéndose por seguro el hecho de que dada una determinada porción de la sucesión sea computacionalmente imposible predecir el siguiente bit. Esto es, un generador es seguro si al analizar las sucesiones obtenidas por aquel, éstas satisfacen el criterio de ser impredecibles. Por lo tanto para que un generador pueda ser usado en algún algoritmo criptográfico, sus sucesiones deberán ser impredecibles. El generador de congruencia lineal que se estudiará con más detalle en el siguiente capítulo, no cumple con este último criterio.

Definición 1.5 *Un GBPA se dice que pasa todas las* **pruebas estadísticas polinomiales** *si ningún algoritmo en tiempo polinomial puede distinguir entre una sucesión de salida de un GBPA y una sucesión realmente aleatoria, ambas con el mismo número de elementos, con una probabilidad significativamente mayor que $\frac{1}{2}$.*

Definición 1.6 *Un GBPA se dice que pasa la* **prueba del siguiente bit** *si no existe un algoritmo de tiempo polinomial, el cual, al tener como entrada una sucesión s de los primeros l bits pueda predecir el bit $l+1$ de s con una probabilidad mayor que $\frac{1}{2}$.*

Hecho 1.1 *Un GBPA pasa la prueba del siguiente bit, si y sólo si pasa todas las pruebas estadísticas polinomiales. [BLU84]*

Definición 1.7 *Un GBPA que pasa la prueba del siguiente bit se denomina un* **generador de bits pseudoaleatorios criptográficamente fuerte (GBPCF)**.

Cabe mencionar que éste último tipo de generadores, que son los que nos interesan, se dice que pasan la prueba del siguiente bit, asumiendo la intratabilidad de algunos problemas de la teoría de números como son el *problema del logaritmo discreto, el problema de la residuosidad cuadrática y el problema de la factorización.*

1.5. Pruebas estadísticas de aleatoriedad

En esta sección se mencionan algunas de las pruebas estadísticas más usadas, que se aplican a las sucesiones de bits generadas en forma determinística [KNU81, MEN97, MAU92], dichas pruebas ayudan a detectar cierto tipo de "debilidades" en un generador y si éste se comporta como debiera hacerlo una verdadera fuente de cantidades aleatorias. Por ejemplo, se verifica si en una sucesión existe aproximadamente el mismo número de ceros que de unos, ya que se supone que así ocurre con una sucesión realmente aleatoria.

Si la sucesión no pasa alguna de las pruebas de aleatoriedad, el generador es rechazado por no cumplir una de las características de las sucesiones pseudoaleatorias. Por el contrario, si el generador pasa las pruebas estadísticas ya existentes, entonces es "aceptado" como aleatorio, entendiéndose el término "aceptado", como "no rechazado", pues las pruebas sólo funcionan como evidencia probabilística de que las sucesiones obtenidas de dicho generador se comportan como las verdaderamente aleatorias. Sin embargo, esto no garantiza totalmente que en efecto sea aleatorio.

Las siguientes pruebas estadísticas fueron extraídas de [MEN97]. Sin embargo, el *National Institute of Standards and Technology (NIST)*, cuya página

web aparece en las referencias, a últimas fechas, ha diseñado un conjunto de programas que permiten realizar éstas pruebas y algunas otras, a las sucesiones obtenidas por cualquier GBPA.

En las pruebas estadísticas que se muestran a continuación, se supone que se tiene una sucesión $s = s_0, s_1, s_2, \ldots, s_{n-1}$ de longitud n.

Prueba de frecuencia o prueba del monobit

El propósito de esta prueba es determinar si el número de 0's y 1's es el mismo aproximadamente, como se esperaría en una sucesión realmente aleatoria. Las literales n_0, n_1 denotan el número de 0's y 1's respectivamente. La estadística usada es:

$$X_1 = \frac{(n_0 - n_1)^2}{n}$$

Esta estadística sigue aproximadamente una distribución χ^2 con un grado de libertad, si $n \geq 10$.

Prueba serial

Se considera que en una sucesión aleatoria, el número de ocurrencias de parejas 00, 01, 10 y 11 debe ser aproximadamente el mismo. Por lo tanto, para realizar esta prueba se cuenta el número de dichas ocurrencias en la sucesión dada. La estadística usada es:

$$X = \frac{4}{n-1}(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2) - \frac{2}{n}(n_0^2 + n_1^2) + 1$$

donde $n_{00}, n_{01}, n_{10}, n_{11}$, corresponden al número de ocurrencias de cada par. Esta estadística sigue aproximadamente una distribución χ^2 con 2 grados de libertad si $n \geq 21$.

Prueba del póker

La idea en la cual está basada ésta prueba, es similar a las dos anteriores, sólo que en vez de contar el número de 1's y de 0's, como en la primera de ellas, o el número de ocurrrencias de las parejas como en la segunda, se cuentan las ocurrencias de palabras de tamaño m, donde m es un entero tal que:

$$\lfloor n/m \rfloor \geq 5 \cdot (2^m) \text{ y } k = n/m$$

y la prueba determina si las diferentes palabras ocurren con la misma frecuencia. La estadística usada es:

$$X = \frac{2^m}{k}(\sum_{i=1}^{2^m} n_i^2) - k$$

donde n_i es el número de palabras que representan al entero i en bits. La estadística sigue aproximadamente una distribución χ^2 con $2^m - 1$ grados de libertad.

Prueba de rachas (runs test)

El nombre de la prueba se debe a que a un bloque de 0's o de 1's consecutivos se le conoce como *racha (run)*. Si se tiene el primer caso, entonces a dicho bloque se le llama *hueco*, en el segundo caso se le denomina *bloque*. La prueba consiste en contar el número de huecos, G_i [2], o de bloques B_i, de longitud i, donde i es un entero tal que $1 \leq i \leq k$, y k es el mayor entero que corresponde a la longitud del bloque más grande hallado en la muestra de tamaño n. El objetivo de ésta prueba es comparar el número de bloques o huecos existentes con la cantidad esperada dada por la siguiente expresión:

$$e_i = (n - i + 3)/2^{i+2}$$

La estadística usada es:

$$\sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}$$

la cual sigue aproximadamente una distribución χ^2 con $2k - 2$ grados de libertad.

Prueba de autocorrelación

El propósito es observar si existe relación entre la sucesión dada de tamaño n y la sucesión corrida d posiciones. Esta operación se realiza usando la operación XOR, para determinar en cuántas posiciones difieren ambas sucesiones. Esto se lleva a cabo de la siguiente forma:

[2] G_i, se debe a que el vocablo inglés *gap*, que se puede traducir como *hueco, brecha o zanja*

$$A(d) = \sum_{i=0}^{n-d-1}(s_i \oplus s_{i+d})$$

donde s_i es el i-ésimo bit de la sucesión y la estadística usada es:

$$X = 2(A(d) - \frac{n-d}{2})/\sqrt{n-d}$$

la cual sigue aproximadamente una distribución normal.

Prueba universal de Maurer

Maurer [MAU92] propone en esta prueba verificar la posibilidad de comprimir información, la cual debe ser mínima en una sucesión pseudoaleatoria. Es decir, normalmente es posible comprimir debido a que existe mucha redundancia en la información, dicha redundancia no debe existir o debe ser muy pequeña en una sucesión pseudoaleatoria. Para aplicar la prueba, se siguen los siguientes pasos:

- Se escoge el parámetro L entero, tal que $6 \leq L \leq 16$.
- La sucesión dada se subdivide en bloques de tamaño L no traslapados.
- Posteriormente se calcula el número de éstos bloques, el cual se denota como $Q+K$, donde Q se elige de tal forma que cada una de las palabras de longitud L, ocurra por lo menos una vez en los primeros Q bloques. Maurer propone que $Q = 10 \cdot 2^L$.
- Se construye una tabla T, tomando el entero j que representa el bloque en cuestión b_i y se asigna la posición en $T[j]$.
- Los K bloques restantes son usados para calcular la estadística de la siguiente forma: para cada i, $Q+1 \leq i \leq Q+K$, sea $A_i = i - T[b_i]$; A_i es el número de posiciones desde la última ocurrencia del bloque b_i. Entonces

$$X = \frac{1}{K}\sum_{i=Q+1}^{Q+K} \log A_i$$

Cabe mencionar que Maurer propone que K debe ser al menos igual a $1000 \cdot 2^L$. Aplicando el teorema del límite central a la estadística X para normalizarla se obtiene $Z = (X - \mu)/\sigma$, donde σ^2 es la varianza y μ es la media.

Capítulo 2

Generador de Congruencia Lineal (GCL)

Como ya se ha mencionado anteriormente, uno de los generadores de sucesiones pseudoaleatorias ampliamente usado es el *generador de congruencia lineal* [KNU81]. En el presente capítulo se dará una explicación acerca de su funcionamiento, y de cómo elegir los mejores parámetros. También se ha mencionado que este tipo de generadores no es seguro para usos criptográficos, hecho que se explicará al final del presente capítulo.

Definición 2.1 *Un generador de congruencia lineal (GCL), se construye con los siguientes parámetros. Sean $m, a, c, x_0 \in \mathbb{Z}$, donde*

$$\begin{array}{rll} m & \textit{el módulo} & m > 0 \\ a & \textit{el multiplicador} & 0 < a < m \\ c & \textit{el incremento} & 0 \leq c < m \\ x_0 & \textit{semilla} & 0 \leq x_0 < m \end{array}$$

y la sucesión de números pseudoaleatorios viene dada por

$$x_{n+1} \equiv ax_n + c \bmod m$$

Ejemplo.

Si se escogen los siguientes parámetros: $a = 5, c = 3, m = 8$ y como semilla $x_0 = 0$, entonces la sucesión generada es

$$0,3,2,5,4,7,6,1,0,3,2, \ldots$$

Como se puede observar, los elementos de la sucesión comienzan a repetirse a partir de algún punto de la misma. El inicio de la sucesión estará dado por la semilla y los siguientes elementos por la relación de recurrencia. Es decir, en la sucesión exisistirá un conjunto de elementos que se repiten, lo cual conduce a dar la siguiente definición.

Definición 2.2 *Al conjunto de elementos que se repiten en una sucesión se le denomina período y su cardinalidad es denominada longitud.*

En el ejemplo anterior, el período viene dado por los elementos: 0, 3, 2, 5, 4, 7, 6, 1 y la longitud del período correspondiente es 8. De una forma intuitiva es posible apreciar, que uno de los objetivos al generar sucesiones pseudoaleatorias de esta manera, es tener un período de una longitud tan grande como sea posible. Es fácil ver que en el caso del generador de congruencia lineal, la máxima longitud de período que se puede conseguir está dada por el módulo m. Para conseguir una longitud de período m es necesario escoger adecuadamente los parámetros a, c y m. En la siguiente sección, se explicará cómo elegir esos parámetros para conseguir una sucesión con longitud de período m.

Una expresión que será útil más adelante es la que aparece en la siguiente afirmación.

Proposición 2.1 *La expresión*

$$x_{n+k} = (a^k x_n + (a^k - 1)c/b) \text{ mód } m, \quad k \geq 0, \quad n \geq 0, \tag{2.1}$$

donde a y c son los parámetros del GCL, y $b = a - 1$, define una generalización del generador de congruencia lineal, y expresa el $(n+k)$-ésimo término directamente en función del n-ésimo término.

Demostración.

Se hará inducción sobre k. Si $k = 1$ entonces:

$$\begin{aligned} x_{n+1} &= ax_n + \frac{a-1}{a-1} c \text{ mód } m \\ &= ax_n + c \text{ mód } m \end{aligned}$$

Para $k - 1$ se cumple que

$$x_{n+k-1} = a^{k-1} x_n + (a^{k-1} - 1)c/b \text{ mód } m$$

Se mostrará que la expresión dada, también es válida para k. Así

$$\begin{aligned} x_{n+k} &= a^k x_n + (a^k - 1)c/b \bmod m \\ &= a^k x_n + (a^{k-1} + a^{k-2} + \cdots + 1)c \bmod m \\ &= a^k x_n + (a^{k-1} + a^{k-2} + \cdots + a)c + c \bmod m \\ &= a^k x_n + a(a^{k-2} + a^{k-3} + \cdots + 1)c + c \bmod m \\ &= a(a^{k-1} x_n + (a^{k-2} + a^{k-3} + \cdots + 1)c) + c \bmod m \\ &= a(a^{k-1} x_n + (a^{k-1} - 1)c/b) + c \bmod m \\ &= a x_{n+k-1} + c \bmod m \end{aligned}$$

□

2.1. Obteniendo la máxima longitud de período

A continuación se explicará la forma en la que se debe elegir a los parámetros a, c y m del generador de congruencia lineal para obtener una sucesión con un período de longitud máxima [KNU81].

Teorema 2.1 *La sucesión de congruencia lineal definida por m, a, c y x_0 tiene longitud de período m si y sólo si*

i) $(c, m) = 1$*;*

ii) $a \equiv 1 \bmod p$ *para cada primo* p *tal que* $p|m$*;*

iii) si $4|m$ *entonces* $a \equiv 1 \bmod 4$*.*

Para demostrar más fácilmente este resultado, se necesitará un par de lemas.

Lema 2.1 *Sea p un número primo y e un entero positivo, tales que $p^e > 2$. Si*

$$x \equiv 1 \bmod p^e \quad y \quad x \not\equiv 1 \bmod p^{e+1},$$

entonces

$$x^p \equiv 1 \bmod p^{e+1} \quad y \quad x^p \not\equiv 1 \bmod p^{e+2}$$

Demostración.
Tenemos que $x = 1 + qp^e$, para algún entero q que no sea múltiplo de p. Por el teorema del binomio:

$$\begin{aligned} x^p &= 1 + \sum_{k=1}^{p} \binom{p}{k} q^k p^{ke} \\ &= 1 + qp^{e+1}\left[1 + \sum_{k=2}^{p} \binom{p}{k} \frac{q^{k-1}p^{(k-1)e}}{p}\right] \end{aligned} \tag{2.2}$$

Si $1 < k < p$ el coeficiente binomial $\binom{p}{k}$ es divisible por p, y por lo tanto,

$$\frac{1}{p}\binom{p}{k} q^{k-1}p^{(k-1)e}$$

es divisible por $p^{(k-1)e}$. De la expresión 2.2 se obtiene que el último término $q^{p-1}p^{(p-1)e-1}$ es divisible por p puesto que $(p-1)e > 1$ cuando $p^e > 2$. Es decir, los términos dentro del paréntesis son enteros y divisibles por p, lo cual implica que $x^p = 1 + qp^{e+1}(1 + ps)$ para algún $s \in \mathbb{Z}$, es decir,

$$x^p \equiv 1 \text{ mód } p^{e+1} \qquad \text{y} \qquad x \not\equiv 1 \text{ mód } p^{e+2}$$

□

Proposición 2.2 *Sea* $x_n \equiv (ax_{n-1} + c) \text{ mód } m$. *Si* $d|m$ *y* $y_n \equiv x_n \text{ mód } d$ *entonces,*

$$y_{n+1} \equiv ay_n + c \text{ mód } d.$$

Demostración.
Si $d|m$ entonces $x_n \equiv ax_{n-1} + c \text{ mód } d$. También se cumple que $x_{n+1} \equiv ax_n + c \text{ mód } d$. Dado que $y_n \equiv x_n \text{ mód } d$ se tiene que

$$\begin{aligned} y_{n+1} &\equiv x_{n+1} \text{ mód } d \\ &\equiv ax_n + c \text{ mód } d \\ &\equiv ay_n + c \text{ mód } d \end{aligned}$$

□

Lema 2.2 *Si la descomposición de m en factores primos es:*

$$m = p_1^{e_1} \dots p_t^{e_t}$$

entonces la longitud λ del período de la sucesión de congruencia lineal determinada por (x_0, a, c, m) es el mínimo común múltiplo de las longitudes λ_j de los periodos de las sucesiones de congruencia lineal: $(x_0 \text{ mód } p_j^{e_j}, a \text{ mód } p_j^{e_j}, c \text{ mód } p_j^{e_j}, p_j^{e_j})$, $1 \leq j \leq t$.

Demostración.

La prueba se hará por inducción en t. Es suficiente probar que si m_1 y m_2 son primos relativos, la longitud λ de la sucesión de congruencia lineal determinada por:

$$(x_0, a, c, m_1 m_2)$$

es el mínimo común múltiplo de las longitudes de los períodos λ_1 y λ_2 de las sucesiones determinadas por:

$$\begin{array}{c}(x_0 \text{ mód } m_1, a \text{ mód } m_1, c \text{ mód } m_1, m_1)\\(x_0 \text{ mód } m_2, a \text{ mód } m_2, c \text{ mód } m_2, m_2)\end{array}$$

Denotaremos a las sucesiones anteriores por x_n, y_n y z_n, respectivamente. Por la proposición 2.2, es claro que

$$y_n \equiv x_n \text{ mód } m_1 \text{ y } z_n \equiv x_n \text{ mód } m_2 \;\; \forall\, n \geq 0,$$

y por el teorema A.1, del apéndice A, se sigue que

$$x_n \equiv x_k \text{ mód } m_1 m_2 \quad \text{si y sólo si} \quad \begin{array}{l} y_n \equiv y_k \text{ mód } m_1 \text{ y} \\ z_n \equiv z_k \text{ mód } m_2 \end{array} \tag{2.3}$$

Si $\lambda' = [\lambda_1, \lambda_2]$, se desea probar que

$$\lambda' = \lambda$$

recordando que λ es el periodo de la sucesión con módulo $m_1 m_2$. Puesto que $x_n \equiv x_{n+\lambda}$ mód $m_1 m_2$ para todo n suficientemente grande, tenemos que

$$y_n \equiv y_{n+\lambda} \text{ mód } m_1$$

de donde se concluye que λ es múltiplo de λ_1. Análogamente se tiene que:

$$z_n \equiv z_{n+\lambda} \text{ mód } m_2$$

por lo que λ es múltiplo también de λ_2. De ambas afirmaciones se sigue que $\lambda' \leq \lambda$. Más aún, sabemos que

$$y_n \equiv y_{n+\lambda'} \text{ mód } m_1 \text{ y } z_n \equiv z_{n+\lambda'} \text{ mód } m_2$$

para todo n convenientemente grande. Por consiguiente por la expresión 2.3

$$x_n \equiv x_{n+\lambda'} \text{ mód } m_1 m_2$$

entonces $\lambda \leq \lambda'$, y por lo tanto $\lambda = \lambda'$.

□

Ejemplo.

Considérese la 4-tupla $(0, 21, 3, 40)$, i.e., $x_n = 21x_{n-1} + 3 \text{ mód } 40$ que genera la sucesión:

3, 26, 29, 12, 15, 38, 1, 24, 27, 10, 13, 36, 39, 22, 25, 8, 11, 34, 37, 20,
23, 6, 9, 32, 35, 18, 21, 4, 7, 30, 33, 16, 19, 2, 5, 28, 31, 14, 17, 0, 3 ...

cuya longitud de periodo es 40. Dado que la descomposición de m en factores primos es $m = 2^3 \cdot 5$, tenemos las siguientes sucesiones:

$$\begin{array}{lcl} (0 \text{ mód } 2^3, 21 \text{ mód } 2^3, 3 \text{ mód } 2^3, 2^3) & = & (0, 5, 3, 8) \\ (0 \text{ mód } 5, 21 \text{ mód } 5, 3 \text{ mód } 5, 5) & = & (0, 1, 3, 5) \end{array}$$

La sucesión generada por la primera expresión es: $3, 2, 5, 4, 7, 6, 1, 0, 3, \ldots$, es decir, tiene período 8. La segunda expresión genera la sucesión: $3, 1, 4, 2, 0, 3, \ldots$, cuya longitud de período es 5. El mínimo cómun múltiplo de 8 y 5 es 40, que en efecto es la longitud de período de la sucesión original.

Del lema anterior, se deduce que para que la longitud de período sea $\lambda = p_1^{e_1} \ldots p_t^{e_t}$, la longitud de período λ_j, de cada una de las sucesiones $(x_0 \text{ mód } p_j^{e_j}, a \text{ mód } p_j^{e_j}, c \text{ mód } p_j^{e_j}, p_j^{e_j})$ debe ser $p_j^{e_j}$, para $1 \leq j \leq t$. De esta manera, $[\lambda_1, \ldots, \lambda_t] = p_1^{e_1} \ldots p_t^{e_t}$. Por tanto, se puede suponer que $m = p^e$, donde p es primo y e es un entero positivo.

Con la ayuda de los lemas anteriores se dará ahora la demostración del teorema 2.1. Se puede ver fácilmente que las afirmaciones *ii)* y *iii)* de tal teorema se cumplen, si $a = 1$. Para demostrar la afirmación *i)*, obsérvese que cuando $a = 1$, la sucesión se convierte en:

$$x_n = x_0 + nc \text{ mód } p^e$$

Si $(c, p^e) = d > 1$ entonces $d = p^t$, con $t \leq e$. Por lo tanto, la sucesión está dada por

$$x_n = x_0 + np^t s \text{ mód } p^e$$

y de ahí que x_0 aparece como elemento de la sucesión antes de que $n = p^e$.

Ahora considérese el caso cuando $a > 1$. Puesto que ningún entero ocurre dos veces en el período de la sucesión, el período podrá ser de longitud m si y sólo si cada uno de los posibles enteros, $0 \leq x < m$ aparece en la sucesión. Así, la longitud de período será m si al considerar $x_0 = 0$, la sucesión resultante es de longitud m. Entonces usando la expresión para hallar el término $n + k$ de la proposición 2.1 tenemos,

$$x_n = \left(\frac{a^n - 1}{a - 1}\right) c \text{ mód } m$$

Si c no es primo relativo con m, entonces nunca podrá ocurrir $x_n = 1$, de ahí que la afirmación $i)$ del teorema 2.1 es necesaria. Por consiguiente el teorema 2.1 se reduce al siguiente lema:

Lema 2.3 *Supóngase que* $1 < a < p^e$ *y* $1 < e \in \mathbb{Z}$, *donde* p *es primo. Si* λ *es el entero positivo más pequeño para el cual* $(a^\lambda - 1)/(a - 1) \equiv 0 \text{ mód } p^e$, *entonces*

$$\lambda = p^e \textit{ si y sólo si } \begin{cases} a \equiv 1 \text{ mód } p & \textit{si } p > 2, \\ a \equiv 1 \text{ mód } 4 & \textit{si } p = 2 \end{cases}$$

Demostración.

Supóngase que $\lambda = p^e$, $p > 2$ y que $a \not\equiv 1 \text{ mód } p$. En tal caso, la expresión: $(a^\lambda - 1)/(a - 1) \equiv 0 \text{ mód } p^e$ sería verdadera solamente si $(a^\lambda - 1) \equiv 0 \text{ mód } p^e$, lo cual implica que $a^{p^e} \equiv 1 \text{ mód } p^e$ y por lo tanto $a^{p^e} \equiv 1 \text{ mód } p$. Sin embargo por el teorema de Fermat, (ver apéndice A.3), se sabe que $a^p \equiv a \text{ mód } p$. Si se aplica dicho teorema e veces, se concluirá que $a^{p^e} \equiv a \text{ mód } p$ y se obtiene una contradicción, por el hecho de suponer que $a \not\equiv 1 \text{ mód } p$.

Supongase ahora que $p = 2$ y que $\lambda = 2^e$. Si $a \not\equiv 1 \text{ mód } 4$, entonces se tendrían tres posibilidades

1. $a \equiv 0 \text{ mód } 4$,

2. $a \equiv 2$ mód 4

3. $a \equiv 3$ mód 4

Si $a \equiv 0$ mód 4 entonces es fácil ver que $a \equiv 1$ mód 2, por lo tanto $a - 1 \equiv 1$ mód 2 y $a^{2^e} - 1 \equiv 1$ mód 2. En consecuencia $(a^{2^e} - 1)/(a-1) \not\equiv 0$ mód 2^e. Un argumento similar, muestra que $a \equiv 2$ mód 4 tampoco es factible.

Por último, si $a \equiv 3$ mód 4, se ha demostrado en la proposición A.1) que $(a^{2^{e-1}} - 1)/(a-1) \equiv 0$ mód 2^e, por lo que $a \not\equiv 3$ mód 4.

La justificación anterior muestra que en general, siempre que $\lambda = p^e$ es necesario que $a = 1 + qp^f$, donde $p^f > 2$ y q no es múltiplo de p.

Recíprocamente, si $a \equiv 1$ mód 4 o $a \equiv 1$ mód p se probará que $\lambda = p^e$. Para ello se aplica sucesivamente el lema 2.1, así se tiene que

$$a^{p^g} \equiv 1 \text{ mód } p^{f+g}, \quad a^{p^g} \not\equiv 1 \text{ mód } p^{f+g+1},$$

para toda $g \geq 0$, y por tanto

$$\begin{aligned} (a^{p^g} - 1)/(a-1) &\equiv 0 \text{ mód } p^g, \\ (a^{p^g} - 1)/(a-1) &\not\equiv 0 \text{ mód } p^{g+1}. \end{aligned}$$

En particular $(a^{p^e} - 1)/(a-1) \equiv 0$ mód p^e. De esta forma, la sucesión $(0, a, 1, p^e)$ viene dada por $x_n = (a^{p^e} - 1)/(a-1) \equiv 0$ mód p^e; por tanto tiene período de longitud λ, esto es, $x_n = 0$ si y sólo si n es múltiplo de λ, por ello p^e es un múltiplo de λ. Esto puede suceder solamente si $\lambda = p^g$, para alguna g, y las relaciones arriba mencionadas implican que $\lambda = p^e$ completando la prueba del teorema 2.1.

□

2.2. Generador puramente multiplicativo

Por último se considerará a los *generadores puramente multiplicativos*, denominados así cuando $c = 0$. Aunque el proceso de generación es un poco más rápido, el teorema 2.1 indica que no será posible alcanzar un período de longitud máxima. Es sencillo ver que la relación de recurrencia para este tipo de generadores viene dada por:

$$x_{n+1} \equiv a x_n \text{ mód } m,$$

Observése que el valor $x_n = 0$ no deberá aparecer nunca, pues de lo contrario la sucesión degeneraría a cero. Además, si $d|m$ y x_n es un múltiplo de d, los siguientes elementos en la sucesión: $x_{n+1}, x_{n+2}, \ldots$ serán múltiplos de d. En conclusión, si $c = 0$ entonces $(x_n, m) = 1$ para toda n, y esto implica que el período tendrá longitud máxima $\varphi(m)$.

A continuación, será necesario definir las condiciones que debe cumplir el multiplicador, a, para conseguir una sucesión con período de longitud máxima. De acuerdo al lema 2.2, el período de una sucesión depende totalmente de las sucesiones con $m = p^e$. Considerando dicho resultado, se tiene que

$$x_n \equiv a^n x_0 \text{ mód } p^e.$$

Es sencillo ver que si a es un múltiplo de p, tal sucesión tendrá período de longitud 1. En efecto cuando $n = e$ entonces $x_e = p^e q$ mód p^e, lo cual implica que $x_e = 0$ y por tanto la sucesión degeneraría a cero. Dado que a no debe ser un múltiplo de p entonces $(a, p) = 1$. Así, el período es el entero más pequeño λ para el cual $x_0 \equiv a^\lambda x_0$ mód p^e. Si $(x_0, p^e) = p^f$, la condición anterior es equivalente a

$$a^\lambda \equiv 1 \text{ mód } p^{e-f}.$$

Por el teorema de Euler (ver teorema A.4), se tiene que $a^{\varphi(p^{e-f})} \equiv 1$ mód p^{e-f}; de ahí que λ es un divisor de

$$\varphi(p^{e-f}) = p^{e-f-1}(p-1)$$

por el teorema A.6. Si $(a, m) = 1$, el entero más pequeño para el cual $a^\lambda \equiv 1$ mód m, es conocido como *el orden de a módulo m*, denotado por $ord_m a$. Cualquier valor de a que tenga el orden máximo módulo m, se denomina *elemento primitivo módulo m*.

Sea $\lambda(m)$ el orden de un elemento primitivo. Las observaciones anteriores muestran que $\lambda(p^e)|(p^{e-1}(p-1))$, hecho que ayuda a encontrar un valor preciso para $\lambda(p^e)$ en todos los casos:

$$\begin{array}{ll} \lambda(2) = 1,\ \lambda(4) = 2, & \\ \lambda(2^e) = 2^{e-2}, & e \geq 3 \\ \lambda(p^e) = p^{e-1}(p-1), & p > 2 \\ \lambda(p_1^{e_1} \ldots p_t^{e_t}) = [\lambda(p_1^{e_1}), \ldots \lambda(p_t^{e_t})] & \end{array}$$

A λ se le conoce como *mínimo exponente universal* o como *función lambda de Carmichael*. El análisis anterior se resume en el siguiente teorema.

Teorema 2.2 *Con la notación anterior, si al construir un generador de congruencia lineal, c=0, entonces la longitud máxima de período es $\lambda(m)$, con λ como se definió anteriormente. Este período se alcanza si*

i) $(x_0, m) = 1$;

ii) a *es un elemento primitivo módulo* m.

Una implementación del GCL, se incluye en la sección B.1 del apéndice B.

2.3. Predicción de los parámetros del GCL

En esta sección se describirá el procedimiento, para obtener los parámetros a, c y m del generador de congruencia lineal [KRA86, KRA90] asumiendo que se conoce o se tiene acceso a una parte de la sucesión así generada.

Inicialmente, es necesario definir algunas expresiones. Para todo entero positivo $x < m$, se definen las sucesiones infinitas:

$$\begin{array}{c} x_0, x_1, \ldots, x_i, \ldots \\ x'_0, x'_1, \ldots, x'_i, \ldots \end{array}$$

de la siguiente forma:

$$x_i \equiv \begin{cases} x & \text{si } i = 0 \\ ax_{i-1} + c \text{ mód } m & \text{si } i > 0 \end{cases}$$

y

$$x'_{i+1} = x_{i+1} - x_i, \text{ donde } i \geq 0.$$

Proposición 2.3 $x'_{i+1} \equiv ax'_i$ mód m *para todo* $i \geq 1$.

Demostración.
Puesto que $x'_{i+1} = x_{i+1} - x_i$ y $x_i \equiv x_{i-1} + c$ mód m se sigue que

$$\begin{aligned} x'_{i+1} &\equiv (ax_i + c) - (ax_{i-1} + c) \text{ mód } m \\ &\equiv a(x_i - x_{i-1}) \text{ mód } m \\ &\equiv ax'_i \text{ mód } m \end{aligned}$$

□

El siguiente objetivo inmediato es hallar un *algoritmo eficiente*, que reciba como entrada el segmento inicial de la sucesión infinita $x_0, x_1, \ldots, x_i, \ldots$ lo suficientemente largo y que proporcione como salida los enteros a', c', m' tales que $x_i \equiv a'x_{i-1} + c' \text{ mód } m'$ para todo i.

Lema 2.4 *El menor entero $i \geq 1$ tal que $g_i | x'_{i+1}$ es mayor o igual que $2 + \lfloor \log_2 m \rfloor$, donde $g_i = mcd(x'_1, \ldots, x'_i)$ para todo $i \geq 1$.*

Demostración.
Sea t el menor entero positivo $i \geq 1$ tal que $g_i | x'_{i+1}$. Sin pérdida de generalidad se puede suponer que $t \geq 3$. Es claro que para todo i,

$$g_1 = x'_1 \quad \text{y} \quad g_{i+1} = (g_i, x'_{i+1}).$$

Sin embargo, si g_i no divide a x'_{i+1} entonces

$$\begin{array}{rcll} x'_{i+1} & = & g_i q_1 + r_1, & 0 \leq r_1 < g_i \\ g_i & = & r_1 q_2 + r_2, & 0 \leq r_2 < r_2 \\ & \vdots & & \\ r_{n-2} & = & r_{n-1} q_{n-1} + r_n, & 0 \leq r_n < r_{n-1} \\ r_{n-1} & = & r_n q_n & \end{array}$$

Obsérvese que cada uno de los cocientes $q_1, q_2, \ldots, q_{n-1}$ es mayor o igual que 1, y $q_n \geq 2$, puesto que $r_n < r_{n-1}$. Por consiguiente $1 \geq r_n$ y $2r_n \leq r_{n-1} < \ldots < r_1 < g_i$. Además, en éste proceso se tiene que $r_n = g_{i+1}$, es decir, $2g_{i+1} \leq r_{n-1} < \ldots < r_1 < g_i$. De donde se concluye que $2g_{i+1} \leq g_i$, y así $g_{i+1} \leq g_i/2$. En consecuencia

$$g_{t-1} \leq \frac{g_{t-2}}{2}, \quad g_{t-2} \leq \frac{g_{t-3}}{2}, \quad \ldots, \quad g_2 \leq \frac{g_1}{2}.$$

De donde se sigue fácilmente que

$$g_{t-1} \leq \frac{g_1}{2^{t-2}} = \frac{x'_1}{2},$$

y por lo tanto

$$2^{t-2} \leq \frac{x'_1}{g_{t-1}} < m$$

lo cual completa la demostración.

□

Con la notación descrita anteriormente, se tiene el siguiente resultado.

Teorema 2.3 *(J. Plumstead [PLU82]) Existe un algoritmo eficiente, A, que al recibir como entrada la sucesión $x_0, x_1, \ldots, x_{t+1}$ (producida por el generador de congruencia lineal), donde t es el menor entero positivo $j \geq 1$ tal que $g_j | x'_{j+1}$, producirá como salida los enteros a', c' tales que para todo $i \geq 1$,*

$$x_i \equiv a' x'_{i-1} + c' \mod m.$$

Tal algoritmo se ejecuta en un tiempo polinomial $\log_2 m$. Más aún, se tiene que $t \leq \lfloor \log_2 m \rfloor$.

Demostración.

La cota superior en t es una consecuencia inmediata del lema 2.4. El algoritmo A que se propone, es el siguiente:

Entrada: $x_0, x_1, \ldots, x_{t+1}$.

1. Calcular $x'_i = x_i - x_{i-1}$, donde $1 \leq i \leq t+1$.
2. Calcular $d = (x'_1, \ldots, x'_{t+1})$.
3. Calcular $u_1, \ldots, u_{t+1}$ tales que

$$d = \sum_{i=1}^{t+1} u_i x'_i$$

Salida:

$$a' = \sum_{i=1}^{t+1} u_i \frac{x'_{i+1}}{d},\ c' = x_1 - a' x_0.$$

Ahora se tiene:

Proposición 2.4 *$a' x'_i \equiv x'_{i+1} \mod m\ \ \forall\, i \geq 1$.*

Demostración.

Si $g = (m, d)$ entonces

$$ad = a\sum_{i=1}^{t+1} u_i x_i' = \sum_{i=1}^{t+1} a u_i x_i' \equiv$$
$$\sum_{i=1}^{t+1} u_i x_{i+1}' \equiv d\sum_{i=1}^{t+1} u_i \frac{x_{i+1}'}{d} \equiv a'd \text{ mód } m$$

Por la definición de g se tiene que

$$\frac{ad}{g} \equiv \frac{a'd}{g} \text{ mód } \frac{m}{g}$$

además $\left(\frac{d}{g}, \frac{m}{g}\right) = 1$,

$$a \equiv a' \text{ mód } \frac{m}{g}$$

Si $h_i = (x_i', m)$ para todo $i \geq 1$, entonces existen $s, t \in \mathbb{Z}$ tales que $x_i's + mt = h_i$. Por la definición de g, $g|m$ y $g|d$, pero también $d|x_i'$, lo cual implica que $g|x_i'$. Entonces $x_i's + mt = g(rs) + g(qt) = g(rs + qt) = h_i$, es decir, $g|(x_i', m)$. De donde se sigue que, para todo $i \geq 1$,

$$a \equiv a' \text{ mód } \frac{m}{(x_i', m)}.$$

Como $ax_i' \equiv x_{i+1}'$ mód m, entonces a es una solución de la congruencia

$$ux_i' \equiv x_{i+1}' \text{ mód } m$$

Además, al resolver congruencias lineales, es sabido por el teorema A.7 que cada solución es de la forma:

$$a + \frac{mt}{(x_i', m)},\ t = 0, 1, \ldots (x_i', m) - 1$$

y por lo tanto, a' es una solución de la congruencia, con lo cual queda probada la proposición.

□

Continuando con la prueba del teorema 2.3, tenemos que

$$a'x_i + c' - x_{i+1} \equiv a'x_i + (x_1 - a'x_0) - x_{i+1} \equiv$$
$$a'(x_i - x_0) - (x_{i+1} - x_1) \equiv a' \sum_{k=1}^{i} x'_k - \sum_{k=1}^{i} x'_{k+1} \equiv$$
$$\sum_{k=1}^{i} (a'x'_k - x'_{k+1}) \equiv 0 \text{ mód } m$$

□

No se conoce una cota inferior en el número de x_i, que permita predecir un módulo m'. Por lo que la conclusión del teorema 2.3 es válida. Sin embargo, es posible predecir tal módulo si se tiene un segmento finito, lo suficientemente largo, de la sucesión $x_0, x_1, \ldots x_i, \ldots$ producida por el generador de congruencia lineal. Esto se puede llevar a cabo utilizando el procedimiento que se describe a continuación.

Primero, se usa el algoritmo descrito en la prueba del teorema 2.3, para calcular a', c'. Posteriormente, es necesario considerar el siguiente argumento. Se toma algún $M \in \mathbb{Z}$ como módulo y se define la sucesión:

$$x_0(M) = x_0,\ x_{i+1}(M) \equiv a'x_i(M) + c' \text{ mód } M$$

Proposición 2.5 *Para todo $i \geq 0$, y $M, M^* \in \mathbb{Z}$ tal que $M^*|M$ se tiene que $x_i(M) \equiv x_i(M^*)$ mód M^*.*

Demostración.

Tal afirmación se prueba fácilmente por inducción sobre i:

Si $i = 0$ entonces $x_0(M) = x_0$ mód M y $x_0(M^*) = x_0$ mód M^*. Pero puesto que $M|M^*$, se sigue que $x_0(M) \equiv x_0(M^*)$ mód M^*. Se supone que la afirmación es cierta para i, es decir,

$$x_i(M) \equiv x_i(M^*) \text{ mód } M^*$$

entonces para $i + 1$,

$$\begin{aligned} x_{i+1}(M) &\equiv a'x_i(M) + c' \text{ mód } M, \\ x_{i+1}(M^*) &\equiv a'x_i(M^*) + c' \text{ mód } M^*, \end{aligned}$$

de donde

$$x_{i+1}(M) - x_{i+1}(M^*) \equiv a'(x_i(M) - x_i(M^*)) \text{ mód } M^*$$

puesto que $M^*|M$. Además, $x_i(M) \equiv x_i(M^*)$ mód M^*, por lo tanto

$$x_{i+1}(M) \equiv x_{i+1}(M^*) \text{ mód } M^*.$$

□

El procedimiento para hallar el módulo m' es el siguiente: se escoge un módulo $M \in \mathbb{Z}$ y se obtiene la sucesión generada por

$$x_{i+1}(M) \equiv a'x_i(M) + c' \text{ mód } M$$

los elementos de tal sucesión y el fragmento de sucesión dado coincidirán hasta el k-ésimo término. Debido a que tal k está en relación con el módulo escogido, se denotará más formalmente por $k(M)$ al menor entero k tal que $x_{k+1}(M) \not\equiv x_{k+1}$ mód M, para algún módulo M. Posteriormente, se escogerá el siguiente módulo como sigue:

$$M^* = (M, x_{k(M)+1}(M) - x_{k(M)+1})$$

Puesto que $x_{k(M)+1}(M) \equiv x_{k(M)+1}$ mód M^*, se sigue de la proposición 2.5 que para cualquier módulo M, $k(M^*) \geq k(M) + 1$.

En general para encontrar m', es necesario hacer varios intentos, inicialmente se escogerá un módulo m_0 lo suficientemente grande, se generará la sucesión $x_{i+1}(m_0) = a'x_i(m_0)+c'$ mód m_0, se verá hasta que elemento coincide y posteriormente se calculará un nuevo módulo m_1, tal que $m_1 = (m_0, x_{k(m_0)+1}(m_0) - x_{k(m_0)+1})$, y se obtendrá una nueva sucesión $x_{i+1}(m_1) = a'x_i(m_1)+c'$ mód m_1, la cual coincidirá con el fragmento original, al menos en un elemento más. Dicho procedimiento seguirá a lo más s veces, donde s es el número de elementos del fragmento original.

Ejemplo.

Supóngase que sólo se tiene la siguiente porción de una sucesión y que para obtenerla se utilizó un generador de congruencia lineal:

$$x_0 = 0, \quad x_1 = 3, \quad x_2 = 11, \quad x_3 = 24, \quad x_4 = 17, \quad x_5 = 15$$

A partir de ella se intentará predecir los parámetros a, c y m y generar el resto de la sucesión.

Aplicando el algoritmo anterior, tendremos que obtener las $x'_i = x_i - x_{i-1}$ $1 \leq i \leq t+1$:

$$x'_1 = 3, \quad x'_2 = 8, \quad x'_3 = 13, \quad x'_4 = -7, \quad x'_5 = -2$$

Posteriormente se calcula el máximo común divisor, $d = (x'_1, \ldots, x'_t)$, el cual en este caso es 1. A continuación se calculan los coeficientes, u_1, u_2, u_3, u_4, u_5 del máximo común divisor, las cuales son:

$$u_1 = 3, \quad u_2 = -1, \quad u_3 = 0, \quad u_4 = 0, \quad u_5 = 0 \ .$$

De donde, es posible calcular el valor de los parámetros $a' = u_1 x'_2 + u_2 x'_3$ y $c' = x_1 - a' x_0$:

$$\begin{array}{rcl} a' & = & 3(8) + 13(-1) = 11, \\ c' & = & 3 - 0(11) = 3 \end{array}$$

Lo único que falta por hacer es predecir el módulo m.

Puesto que para hallar el módulo es necesario comenzar con un módulo al azar, es posible hacer antes ciertas suposiciones, que pueden ayudar a determinar dicho parámetro más fácilmente. Una de tales suposiciones es que, tal sucesión se generó pensando en tener el período más largo posible. En cuyo caso, tendríamos las siguientes condiciones (ver teorema 2.1):

1. $(3, m) = 1$
2. $a - 1 = pk, k \in \mathbb{Z}$ para cada primo p que divide a m

En este caso $a - 1 = 10 = 5 \cdot 2$, es decir que el módulo tiene que ser divisible por 2 o por 5. Así, es posible escoger $m' = 5^3 = 125$, que junto con los otros parámetros da lugar al siguiente GCL:

$$x_{i+1}(125) = 11x_i + 3 \text{ mód } 125$$

de donde se obtiene una sucesión, cuyos primeros elementos son:

$$0, 3, 36, 24, 17, 65, 93, 26, 39, 57, 5, 58, 16, 54, 97, 70, 23, 6, \ldots$$

El primer elemento en donde ambas sucesiones empiezan a diferir, es el tercero, de esta manera para construir el siguiente módulo, se calcula $x_2(125) - x_2 = 36 - 11 = 25$. Se escoge un nuevo módulo, que será el máximo común divisor del módulo anterior y la diferencia anterior, es decir $(125, 25) = 25$. Así, el nuevo generador es:

$$x_{i+1}(25) = 11x_i + 3 \text{ mód } 25$$

de donde se obtiene la siguiente sucesión[1]:

$$0, 3, 11, 24, 17, 15, 18, 1, 14, 7, 5, 8, 16, 4, 22, 20, 23, 6, 19, 12, 10, 13, 21, 9, 2, \ldots$$

cuyos seis primeros elementos coinciden completamente con los del fragmento de sucesión dado, lo cual indica que el proceso ha concluido y que los parámetros que se usaron para generar tal sucesión son:

$$a' = 11, c' = 11 \text{ y } m' = 25$$

Es importante señalar que los parámetros a' y c' que se obtienen como resultado del algoritmo, no necesariamente son iguales a los originales a y c, pero sí son congruentes módulo m'. Cuando se encuentra el módulo correspondiente, entonces será posible reducir dichos parámetros módulo m', y hallar los originales. Para ilustrar esta situación se muestra el siguiente ejemplo:

Ejemplo.

Considérese ahora el fragmento de sucesión:

$$1, 245, 258, 40, 130, 528, 156, 92, 336, 349, 131, \ldots$$

las x_i' son

$$\begin{array}{lllll} x_1' = 244, & x_2' = 13, & x_3' = -218, & x_4' = 90, & x_5' = 398, \\ x_6' = -372, & x_7' = -64, & x_8' = 244, & x_9' = 13, & x_{10}' = -218 \end{array}$$

Nuevamente el máximo común divisor de las x_i' es 1. La combinación lineal correspondiente es:

$$\begin{array}{lllll} u_1 = 4, & u_2 = -75, & u_3 = 0, & u_4 = 0, & u_5 = 0, \\ u_6 = 0, & u_7 = 0, & u_8 = 0, & u_9 = 0, & u_{10} = 0. \end{array}$$

[1]Para generar a ésta sucesión se utilizó el programa en C, crakgcl que aparece en el apéndice B, sección 1

Así, se encuentran a' y c'

$$\begin{aligned} a' &= 4(13) + (-75)(-218) = 16402 \\ c' &= 245 - 16402 = -16157 \end{aligned}$$

Supóngase que se escoge un módulo $m_0 = 53361$ y se procede a generar la sucesión dada por la relación:

$$x_{i+1}(m_0) = 16402x_i - 16402 \text{ mód } 53361$$

cuyos primeros elementos son:

$$1, 245, 258, 40, 52952, 52272, 51361, 50219, 48846, 47242, 45407, 43341, \ldots$$

Puesto que los elementos de esta nueva sucesión, no coinciden por completo con los del fragmento dado, es necesario calcular un nuevo módulo, m_1, como sigue. Dado que tales sucesiones difieren en el elemento 4, realizamos la diferencia $x_4(m_0) - x_4 = 52952 - 130 = 52822$, y se obtiene el máximo común divisor de la diferencia anterior y m_0, así tenemos:

$$m_1 = (53361, 52822) = 539$$

Se procede a obtener una nueva sucesión dada por:

$$x_{i+1}(m_1) = 16402x_i - 16402 \text{ mód } 539$$

si se reduce a a' y a c' módulo m_1 se tiene:

$$x_{i+1}(m_1) = 232x_i + 13 \text{ mód } 539$$

que genera a la sucesión:

$$\begin{aligned} &1, 245, 258, 40, 130, 528, 156, 92, 336, 349, \\ &131, 221, 80, 247, 183, 427, 440, 222, 312, \ldots \end{aligned}$$

puesto que todos los elementos del fragmento de sucesión y la anterior coinciden, el proceso termina con $a' = 232, c' = 13$ y $m' = m_1 = 539$. Obsérvese que en este caso, aunque a y c no son iguales a a' y c', sí son congruentes módulo m'.

De esta manera, se muestra que no es seguro usar un generador de congruencia lineal en aplicaciones criptográficas, dado que si se posee un fragmento

de alguna sucesión generada por éste, es posible predecir los parámetros, aún sin conocer la semilla, ni ningún otro dato del generador.

Más aún, en ocasiones se puede pensar, que es posible usar un generador de números pseudoaleatorios no muy seguro, como es el caso del generador de congruencia lineal, ya que al utilizarse en un algoritmo criptográfico, no queda al descubierto ningún dato del mismo. Sin embargo, se ha demostrado [BEL97] que tal afirmación es falsa ya que al combinarse el GCL con el algoritmo de firma digital: *Digital Signature Standard*, éste último se vuelve completamente vulnerable.

Las sucesiones obtenidas con este generador, se utilizan en las implementaciones de funciones random de distintos lenguajes de programación como C. También se usan en diversos algoritmos probabilísticos que solucionan cierto tipo de problemas, ya que si se optara por algoritmos determinísticos, éstos serían menos eficientes. Para saber más acerca de este tipo de algoritmos se puede consultar [MOT95].

Capítulo 3

Generador Blum-Blum-Shub

A continuación se mostrará uno de los generadores más importantes para usos criptográficos, el generador Blum-Blum-Shub [BLU86]. Dicho generador basa su seguridad en el problema de los residuos cuadráticos.

3.1. Conceptos básicos

En esta sección se enunciarán algunas definiciones y resultados de Teoría de Números que se utilizarán a lo largo del capítulo, para mayor referencia consultar [HAR83, ROS83].

Definición 3.1 *Sea n un número entero positivo, se define*

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n | mcd(a, n) = 1\}$$

como el grupo de unidades del anillo $\mathbb{Z}_n$ de clases residuales módulo n.

Definición 3.2 *Sea n un número entero positivo, se define*

$$Q_n = \{a \in \mathbb{Z}_n^* \ |x^2 \equiv a \bmod n \text{ tiene solución}\}$$

como el conjunto de residuos cuadráticos módulo n.

Definición 3.3 *Sea p un número primo impar y a un entero no divisible por p. El símbolo de Legendre $\left(\frac{a}{p}\right)$ está definido de la siguiente manera*

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{si } a \in Q_p \\ -1 & \text{en otro caso} \end{cases}$$

Definición 3.4 *Sea n un entero positivo impar cuya factorización es $n = p_1^{e_1} p_2^{e_2} \cdots p_m^{e_m}$ y a un entero tal que $(a, n) = 1$. Entonces el símbolo de Jacobi $\left(\frac{a}{n}\right)$ está dado por*

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1^{e_1} p_2^{e_2} \cdots p_m^{e_m}}\right) = \left(\frac{a}{p_1}\right)^{e_1} \left(\frac{a}{p_2}\right)^{e_2} \cdots \left(\frac{a}{p_m}\right)^{e_m}$$

donde $\left(\frac{a}{p_i}\right)$ con $1 \leq i \leq m$ es el símbolo de Legendre.

Es importante resaltar que si $a \in Q_n$ entonces $\left(\frac{a}{n}\right) = 1$, pero el recíproco no siempre se cumple, es decir, el que para algún $a \in \mathbb{Z}_n^*$ se cumpla que $\left(\frac{a}{n}\right) = 1$, no necesariamente significa que a sea un residuo cuadrático módulo n.

Ejemplo.
Sea $n = 35$ y $a = 3$, el conjunto de residuos cuadráticos $Q_{35} = \{1, 4, 9, 11, 16, 29\}$. Como se puede observar el 3 no es un residuo cuadrático, sin embargo el símbolo de Jacobi es igual a 1:

$$\left(\frac{3}{35}\right) = \left(\frac{3}{5}\right) \left(\frac{3}{7}\right) = (-1)(-1) = 1.$$

Algunas de las propiedades del *símbolo de Legendre* y del *símbolo de Jacobi* se listan en el apéndice del presente trabajo, ver teoremas A.9, A.10. Ahora bien, de particular interés para aplicaciones criptográficas es el caso de los residuos cuadráticos módulo p, donde p es un número primo, y también cuando $n = pq$, tal que p y q son primos impares distintos. Respecto a ellos es importante considerar la cardinalidad de dichos conjuntos.

Proposición 3.1 *Si p y q son números primos impares distintos entonces*

1. *Q_{pq}, es un subgrupo de $\mathbb{Z}_n^*$.*

2. *$|Q_p| = \frac{\varphi(p)}{2} = \frac{p-1}{2}$.*

3. *$|Q_{pq}| = |Q_p||Q_q| = \frac{(p-1)(q-1)}{4}$.*

Demostración.

Primeramente se probará que Q_{pq} es un subgrupo de $\mathbb{Z}_n^*$. Si $a, b \in Q_{pq}$, entonces $\left(\frac{a}{pq}\right) = \left(\frac{b}{pq}\right) = 1$ y por las propiedades del símbolo de Jacobi $\left(\frac{ab}{pq}\right) = 1$. Por consiguiente $ab \in Q_{pq}$ cumpliéndose así la cerradura. Existe en Q_{pq} inverso multiplicativo, a saber 1, puesto que $\left(\frac{1}{pq}\right) = 1$. Si $a \in Q_{pq}$, entonces la ecuación $x^2 \equiv a \bmod pq$ tiene 4 soluciones no congruentes módulo pq, digamos x_1, x_2, x_3 y x_4. Considerando x_1 y $a^{-1} \in \mathbb{Z}_n^*$, se tiene que:

$$\begin{aligned} x_1^2 &\equiv a \bmod n \\ a^{-1} \cdot x_1^2 &\equiv 1 \bmod n \\ (a^{-1} \cdot x_1)^2 &\equiv a^{-1} \bmod n \end{aligned}$$

De esta manera hemos construido una solución para la ecuación $x^2 \equiv a^{-1} \bmod n$, a saber $a^{-1}x_1$. Análogamente se pueden construir otras 3 soluciones no congruentes entre sí, a partir de x_2, x_3 y x_4. Es fácil ver que las soluciones construidas son no congruentes entre sí, puesto que x_1, x_2, x_3 y x_4 ya lo eran. Por lo tanto $a^{-1} \in Q_{pq}$. Dado que la operación es asociativa en $\mathbb{Z}_n^*$, también lo es en Q_{pq}. Queda así demostrado que Q_n es un subgrupo de $\mathbb{Z}_n^*$.

Para probar que la cardinalidad de Q_p es $\frac{p-1}{2}$, se tiene lo siguiente: para hallar a todos los residuos cuadráticos módulo p, se calculan los cuadrados módulo p de cada uno de los elementos de $\mathbb{Z}_p^*$. Por lo tanto tendremos $p-1$ cuadrados módulo p. Ahora bien, se sabe por el lema A.1, que la ecuación $x^2 \equiv a \bmod p$, o bien no tiene solución o tiene exactamente dos soluciones no congruentes entre sí módulo p. Por consiguiente es fácil ver que el número de residuos cuadráticos, es $\frac{p-1}{2}$.
Análogamente ocurre, para Q_{pq}, en este caso el número de cuadrados módulo pq a considerar es $(p-1)(q-1)$ y además si $x^2 \equiv a \bmod n$, o bien no tiene solucón o tiene exactamente 4 soluciones no congruente entre sí módulo pq, por lo que el número de residuos cuadráticos módulo pq es $\frac{\varphi(n)}{4} = \frac{(p-1)(q-1)}{4}$.

□

Definición 3.5 *Sea $n = pq$ y $a \in \mathbb{Z}_n^*$, se define el siguiente conjunto*

$$J_n = \{a \in \mathbb{Z}_n^* | \left(\frac{a}{n}\right) = 1\}$$

es decir, J_n lo conformarán todos aquellos elementos en $\mathbb{Z}_n^*$ cuyo símbolo de Jacobi es igual con uno.

Proposición 3.2 *Sea p un número primo y $a \in \mathbb{Z}_p^*$. Es fácil decidir si a es un residuo cuadrático módulo p como lo indica lo siguiente: $a \in Q_p$ si y sólo si $\left(\dfrac{a}{p}\right) = 1$ y el símbolo de Legendre puede ser calculado eficientemente [MEN97].*

Proposición 3.3 *Sea $n = pq$, donde p y q son números primos impares distintos. Si $a \in J_n$, entonces $a \in Q_n$ si y sólo si $\left(\dfrac{a}{p}\right) = 1$.*

Demostración.
Si $a \in Q_n$ entonces la congruencia $x^2 \equiv a \bmod n$ tiene solución, en consecuencia $x^2 \equiv a \bmod p$ también la tendrá, y por definición del símbolo de Legendre $\left(\frac{a}{p}\right) = 1$.

Recíprocamente, si $\left(\frac{a}{p}\right) = 1$ y $a \in J_n$ se tiene que

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right) = 1,$$

por lo tanto

$$\left(\frac{a}{q}\right) = 1$$

Es decir, las congruencias

$$\begin{aligned} x^2 &\equiv a \bmod p \\ x^2 &\equiv a \bmod q \end{aligned}$$

tienen solución. La congruencia $x^2 \equiv a \bmod p$ tiene exactamente dos soluciones no congruentes módulo p, digamos $x \equiv x_1 \bmod p$ y $x \equiv p - x_1 \bmod p$. Lo mismo ocurre para la congruencia $x^2 \equiv a \bmod q$, así $x \equiv x_2 \bmod q$ y $x \equiv q - x_2 \bmod q$.
Por el Teorema Chino del Residuo (ver teorema A.8), existen cuatro soluciones distintas para la congruencia $x^2 \equiv a \bmod n$, las cuales son únicas módulo pq y se obtienen al resolver los siguientes cuatro pares de congruencias simultáneas

$$\begin{array}{rclcrcl} x & \equiv & x_1 \text{ mód } p & & x & \equiv & p - x_1 \text{ mód } p \\ x & \equiv & x_2 \text{ mód } q & & x & \equiv & x_2 \text{ mód } q \end{array}$$

$$\begin{array}{rclcrcl} x & \equiv & x_1 \text{ mód } p & & x & \equiv & p - x_1 \text{ mód } p \\ x & \equiv & q - x_2 \text{ mód } q & & x & \equiv & q - x_2 \text{ mód } q \end{array}$$

□

Definición 3.6 *Si $m = 2^e p_1^{e_1} \cdots p_k^{e_k}$, donde $p_1, \ldots, p_k$ son primos distintos impares, entonces la función de Carmichael está definida como:*

$$\lambda(2^e) = \begin{cases} 2^{e-1} & si\ e = 1\ \ o\ e = 2, \\ 2^{e-2} & si\ e > 2, \end{cases}$$

y $\lambda(n) = [\lambda(2^e), p_1^{e_1-1}(p_1 - 1), \ldots, p_k^{e_k-1}(p_k - 1)]$.

3.2. Construcción del generador

En la presente sección se explicará la forma de construir el *generador Blum-Blum-Shub* [BLU86], y porqué puede ser utilizado para aplicaciones criptográficas, es decir, se podrá verificar que la sucesión generada de esta manera es impredecible.

El generador Blum-Blum-Shub basa su seguridad en un problema intratable de la Teoría de Números denominado **problema de los residuos cuadráticos**, el cual se puede enunciar como sigue.

Dado un número compuesto impar, $n \in \mathbb{Z}$ y $a \in J_n$, es computacionalmente imposible decidir si a es o no un residuo cuadrático módulo n, i.e., si $a \in Q_n$.

Ahora es claro que la proposición 3.2 indica que si la factorización de n es conocida, entonces el *problema de los residuos cuadráticos*, se puede resolver fácilmente calculando el símbolo de Legendre $\left(\frac{a}{p}\right) = 1$.

Definición 3.7 *Sean $n = pq$ donde p y q son dos primos distintos tales que $p \equiv q \equiv 3 \text{ mód } 4$ y $|p| = |q| = k/2$, para algún entero positivo k suficientemente grande. Elegir alguna $x_0 \in Q_n$ y definir*

$$x_{i+1} = x_i^2 \text{ mód } n,$$

y

$$f(x_0) = (z_1, z_2, \ldots, z_l)$$

donde $z_i = x_i \text{ mód } 2$, $1 \leq i \leq l$. *Entonces* f *es un generador de bits pseudoaleatorio conocido como el* **generador Blum-Blum-Shub**.

Es importante señalar que en la definición anterior se está abusando de la notación, ya que $|p|$ y $|q|$ denotan el número de bits en la expansión binaria de p y q.
La propiedad de que el generador Blum-Blum-Shub sea impredecible, se basa en la dificultad de resolver el problema de los residuos cuadráticos, mencionado líneas arriba. Formalmente los autores del artículo [BLU86] utilizan la definición dada líneas abajo para establecer dicha dificultad, en ella se indica que cualquier algoritmo probabilístico al intentar averiguar si un elemento de $\mathbb{Z}_n^*$ es o no un residuo cuadrático, se equivocará con una cierta probabilidad, para la cual se establece una cota inferior.

Se asume que dicho algoritmo recibe como entrada, la pareja (n, x), tales que $n = pq, p \equiv q \equiv 3 \text{ mód } 4$, $x \in J_n$ y y dará como salida un 1 ó un 0; 1 para cuando $x \in Q_n$, y 0 en caso contrario. De esta manera, se dice que el algoritmo se *equivoca* si da como salida un 0, cuando $x \in Q_n$; y un 1 cuando x no es residuo cuadrático.

Definición 3.8 *Sea* $\mathbf{P}[n, x]$ *cualquier procedimiento probabilístico de tiempo polinomial, que recibe como entrada la pareja* (n, x)*, tal que la representación binaria de* n *y* x *tiene* m *bits, donde* m *es suficientemente grande; y cuya salida es 0 ó 1. La probabilidad de que* $\mathbf{P}[n, x]$*, se equivoque al averiguar si un elemento de* J_n *es residuo cuadrático, es mayor que* $1/2 - 1/m^t$*, donde* t *es un entero positivo, en el siguiente sentido:*

$$\left(\frac{\sum_{x \in J_n} Pr(\mathbf{P}[n,x] \text{ se equivoque })}{\varphi(n)/2} \right) > 1/2 - 1/m^t.$$

La suposición anterior también puede leerse como que la probabilidad, de que un algoritmo probabilístico $\mathbf{P}$ logre su objetivo, es en el mejor de los casos ligeramente mayor a $1/2$.

Es fácil ver que si se conoce n entonces se puede generar eficientemente la sucesión $x_0, x_1, x_2 \ldots$ y por supuesto $b_0, b_1, b_2, \ldots$, comenzando por cualquier raíz x_0. Nótese que elevar al cuadrado módulo n es una función uno a uno en Q_n, como lo prueba el siguiente lema.

Lema 3.1 *Sea $n = pq$, tal que p y q son primos impares distintos y $p \equiv q \equiv 3 \mod 4$. Entonces la ecuación $x^2 \equiv a \mod n$ tiene exactamente una solución x_0, tal que $x_0 \in Q_n$, para toda $a \in Q_n$.*

Demostración.
Puesto que $n = pq$, la ecuación $x^2 \equiv a \mod n$ tiene exactamente cuatro soluciones no congruentes en $\mathbb{Z}_n$: $\pm y_1$ y $\pm y_2$. Ya que $p \equiv q \equiv 3 \mod 4$ entonces $n \equiv 1 \mod 4$. De ahí que por la propiedad 3 del símbolo de Jacobi, (ver teorema A.10)

$$\left(\frac{-1}{n}\right) = 1$$

Por consiguiente, $\left(\frac{-y_1}{n}\right) = \left(\frac{-1}{n}\right)\left(\frac{y_1}{n}\right) = \left(\frac{y_1}{n}\right)$. Lo mismo ocurre con $\pm y_2$: $\left(\frac{y_2}{n}\right) = \left(\frac{-y_2}{n}\right)$. Ahora bien, como y_1 y y_2 son soluciones, entonces $y_1^2 \equiv y_2^2 \mod n$ y por lo tanto $y_1^2 - y_2^2 \equiv 0 \mod n$. Si descomponemos el producto notable, tenemos que $n|(y_1+y_2)(y_1-y_2)$. En consecuencia $pq|(y_1+y_2)(y_1-y_2)$. Se probará que p divide a $y_1 + y_2$ o bien p divide a $y_1 - y_2$.

Si $p|(y_1 + y_2)$ entonces $y_1 \equiv -y_2 \mod p$ y si además $p|(y_1 - y_2)$ entonces $y_1 \equiv y_2 \mod p$, y por lo tanto $y_2 \equiv -y_2 \mod p$. De ésta última afirmación y utilizando las propiedades del símbolo de Jacobi se concluye que

$$\left(\frac{y_2}{p}\right) = -\left(\frac{y_2}{p}\right)$$

y además

$$\left(\frac{y_2}{n}\right) = -\left(\frac{y_2}{n}\right)$$

lo cual conduce a una contradicción. Esto es, p divide exclusivamente a uno de los factores $y_1 + y_2$ o $y_1 - y_2$ y q divide al otro. Supongase que $p|(y_1 + y_2)$ y que $q|(y_1 - y_2)$, entonces

$$y_1 + y_2 = ps \text{ y } y_1 - y_2 = qt$$

o equivalentemente

$$y_1 \equiv -y_2 \mod p \text{ y } y_1 \equiv y_2 \mod q$$

Nuevamente, usando las propiedades del símbolo de Jacobi y del símbolo de Legendre se tiene que

$$\left(\frac{y_1}{p}\right) = \left(\frac{-y_2}{p}\right) \quad \text{y} \quad \left(\frac{y_1}{q}\right) = \left(\frac{y_2}{q}\right)$$

Por consiguiente,

$$\left(\frac{y_1}{n}\right) = \left(\frac{y_1}{p}\right)\left(\frac{y_1}{q}\right) = \left(\frac{-y_2}{p}\right)\left(\frac{y_2}{q}\right) = -\left(\frac{y_2}{n}\right)$$

De donde se concluye que $\left(\frac{y_1}{n}\right) \neq \left(\frac{y_2}{n}\right)$, i.e.,

$$\left(\frac{y_1}{n}\right) = \left(\frac{-y_1}{n}\right) \neq \left(\frac{y_2}{n}\right) = \left(\frac{-y_2}{n}\right)$$

A continuación se descartarán las raíces cuyo símbolo de Jacobi es -1. Quedan dos raíces cuyo símbolo de Jacobi es 1, una de ellas será también un residuo cuadrático. Suponga que se descartan las raíces $\pm y_2$. Entonces quedan $\pm y_1$. Para saber cual de ellas es un residuo cuadrático, debe cumplirse que el símbolo de Legendre sea igual a 1 respecto a p y q. Puesto que

$$\left(\frac{y_1}{n}\right) = 1$$

entonces:

$$\left(\frac{y_1}{p}\right) = \left(\frac{y_1}{q}\right)$$

Si

$$\left(\frac{y_1}{p}\right) = \left(\frac{y_1}{q}\right) = -1$$

por las propiedades del símbolo de Legendre y dado que $p \equiv q \equiv 3 \text{ mód } 4$ se tiene:

$$\left(\frac{-y_1}{p}\right) = \left(\frac{-y_1}{q}\right) = 1$$

En conclusión, la ecuación $x^2 \equiv a \text{ mód } n$, cuando n cumple las condiciones del lema, tiene exclusivamente una raíz que es también un residuo cuadrático módulo n.

□

A continuación se analizarán algunas otras propiedades de las sucesiones, que permitirán ver el caracter de impredecible de las mismas.

Proposición 3.4 *Solamente se podrá generar la sucesión en sentido inverso, i.e., $x_0, x_{-1}, x_{-2}, \ldots$ si y sólo si se conocen los factores p y q de n.*

Demostración.
Supóngase que se puede generar la sucesión hacia atrás eficientemente. Para factorizar n, se escoge al azar alguna $x \in \mathbb{Z}_n^*$, tal que $\left(\frac{x}{n}\right) = -1$. Se toma $x_0 \equiv x^2 \bmod n$ y se calcula x_{-1}. Por las características de las raíces cuadradas, es fácil ver que al hallar el máximo común divisor de $x+x_{-1}$ y n, se tendrá alguno de los factores de n, esto es, $(x + x_{-1}, n) = p$ o a q.
Recíprocamente, supóngase que se conocen los factores de n: p y q, entonces el algoritmo que se presenta en el siguiente teorema, será capaz de generar la sucesión en sentido inverso.

□

Teorema 3.1 *Existe un algoritmo determinístico eficiente A, el cual recibe como entradas: n, p y q, y cualquier $x_0 \in \mathbb{Z}_n^*$, y calcula como salida, el único $x_{-1} \in Q_n$ tal que $x_0 \equiv x_{-1}^2 \bmod n$.*

Demostración.
Formalmente el algoritmo A se describe como sigue:
Entrada: p y q tales que $p \equiv q \equiv 3 \bmod 4$, $x_0 \in Q_n$, donde $n = pq$.

1. Calcular x_p tal que $x_p^2 \equiv x_0 \bmod p$, lo cual se puede hacer fácilmente utilizando la afirmación A.2. De la misma forma, calcular x_q. Nótese que tanto x_p como x_q cumplen con $\left(\frac{x_p}{p}\right) = 1$ y $\left(\frac{x_q}{q}\right) = 1$.

2. Utilizar el algoritmo extendido de Euclides para hallar los enteros u y v tales que $pu + qv = 1$.

3. La solución buscada es $x_n = x_p qv + x_q pu$.

Salida: $x_{-1} \in Q_n$, tal que $x_{-1}^2 \equiv x_0 \bmod n$.

Para ver que x_n es solución de la ecuación $x^2 \equiv x_0 \bmod n$, primero se observa que $x_n \equiv x_q pu \bmod q$ y que $x_n \equiv x_p qv \bmod p$. Pero también $pu \equiv 1 \bmod q$ y $qv \equiv 1 \bmod p$. Por lo tanto $x_n \equiv x_q \bmod q$ y $x_n \equiv x_p \bmod p$. En consecuencia,

$$x_n^2 \equiv x_q^2 \equiv x_0 \text{ mód } q, \quad x_n^2 \equiv x_p^2 \equiv x_0 \text{ mód } p$$

Es decir, $x_n^2 \equiv x_0$ mód q y $x_n^2 \equiv x_0$ mód p, por el corolario A.1 se concluye que $x_n^2 \equiv x_0$ mód n. Ahora bien, que $x_n \in Q_n$ se deduce del hecho de que $x_n \equiv x_p$ mód p y $x_n \equiv x_q$ mód q, aplicando la propiedad 2 del teorema A.9 se concluye que $\left(\frac{x_n}{p}\right) = \left(\frac{x_n}{q}\right) = 1$ y por lo tanto $x_n \in Q_n$.

□

Proposición 3.5 *Si se conocen los factores primos p y q de n y alguna $x_0 \in Q_n$, entonces se podrá tener una ventaja ϵ al encontrar la paridad de x_{-1}.*

Antes de probar la afirmación anterior, se dará la siguiente definición y un par de lemas.

Definición 3.9 *Un algoritmo probabilístico polinomial $\mathbf{P}[n, x_0]$ tiene una ventaja ϵ, con $0 < \epsilon \leq 1/2$ sobre n, para averiguar la paridad de x_{-1} si*

$$\frac{\sum_{x_0 \in Q_n} Pr(\mathbf{P}[n, x_0] = \textit{ paridad}(x_{-1}))}{\varphi(n)/4} \geq \frac{1}{2} + \epsilon$$

El definición anterior nos indica que un algoritmo probabilístico tendrá cierta ventaja si, en promedio, sus respuestas son correctas más del 50 % de las veces.

Lema 3.2 *Un algoritmo probabilístico para obtener la paridad, con ventaja ϵ, puede convertirse fácilmente en un algoritmo probabilístico con ventaja ϵ, para saber si un elemento en $\mathbb{Z}_n^*$ es residuo cuadrático.*

Demostración.
Supóngase que se desea averiguar si elemento $x \in J_n$ es residuo cuadrático. Para hacerlo se realiza la siguiente operación: $x_0 \equiv x^2$ mód n. La otra raíz cuadrada x_1 tal que $x_0 \equiv x_1^2$ mód n, es $n - x$. Como se observa en la demostración del lema 3.1, también $x_1 \in J_n$. Puesto que además $p \equiv q \equiv 3$ mód 4, entonces x y x_1 tienen paridad opuesta. Aquella cuya paridad sea la misma que la obtenida por el algoritmo mencionado en el lema 3.2, será residuo cuadrático módulo n.

□

Lema 3.3 *La ventaja ϵ, para adivinar si un entero módulo m es residuo cuadrático se puede ampliar a una ventaja $1/2-\epsilon$, uniforme y eficientemente.*

El lema anterior, significa que una vez que se tiene un algoritmo probabilístico con ventaja ϵ para averiguar si se tiene un residuo cuadrático, entonces dicha ventaja ϵ es posible ampliarla a una ventaja $1/2-\epsilon$, es decir, que la probabilidad de que el algoritmo acierte aumenta, ya que ahora dicha probablidad será mayor o igual que $1-\epsilon$.
Ahora sí es posible probar la proposición 3.5.
Demostración. (de la proposición 3.5).
Supóngase que dada una n de la forma previamente establecida, y que no se conocen sus factores primos p y q. Y que se tiene un algoritmo probabilístico $\mathbf{P}$ que tiene una ventaja $1/m^t$ al adivinar la paridad. Después es posible convertir dicho algoritmo en un algoritmo $\mathbf{P'}$ para determinar si un entero módulo m es residuo cuadrático, cuya ventaja se puede ampliar a $1/2-1/m^{t'}$. Esto querría decir que entonces se puede saber si un elemento $x \in J_n$ es residuo cuadrático con una probabilidad, en el peor de los casos de 1/2. Sin embargo, la suposición acerca de la dificultad para resolver el problema de los residuos cuadráticos, indica que cualquier algoritmo probabilístico, atinará correctamente, con una probabilidad, en el mejor de los casos, ligeramente mayor de 1/2. Por lo tanto, el asumir que existe un algoritmo que sin conocer los factores primos de n, encuentre correctamente si un entero módulo n es un residuo cuadrático con una probabilidad de al menos 1/2 es una contradicción.

□

3.2.1. Un par de ejemplos

A continuación se muestra un par de ejemplos, los cuales fueron construidos con el programa en Java cuyo código aparece en el apéndice. Es importante mencionar que aunque aquí se ha incluido un ejemplo pequeño, el programa puede realizar sucesiones donde la longitud de p y q puede ser mayor.

Ejemplo.
Se escogieron $p = 179, q = 223, n = 39917, x_0 = 188$, generando la sucesión que a continuación se muestra:

```
0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1,
0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0,
0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0,
0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1,
1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0,
0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1,
0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1,
0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0,
0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1,
1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1,
0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1,
0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1,
0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1,
0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1,
0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1,
0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0,
1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0,
1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1,
1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, ...
```

Ejemplo.
Se tomaron como parámetros $p = 139, q = 131, n = 18209, x_0 = 225$, obteniendo con ellos la sucesión:

```
1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1,
1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0,
0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1,
0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1,
0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1,
0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0,
0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, ...
```

3.3. Longitud de período

Las sucesiones resultantes del generador Blum-Blum-Shub, al igual que aquellas obtenidas por el generador de congruencia lineal, estudiado en el

capítulo 1, tienen una longitud de período. Tal longitud está relacionada con el *mínimo exponente universal* también denominada *función lambda de Carmichael* , que se mencionó en el mismo capítulo, y cuya definición se puede consultar en el apéndice.

Teorema 3.2 *Sea* $n = pq$*, tal que* $p \equiv q \equiv 3 \text{ mód } 4$*. Si* $x_0 \in Q_n$ *entonces* $\pi | \lambda(\lambda(n))$*, donde* $\pi = \pi(x_0)$ *es el período de la sucesión generada usando residuos cuadráticos.*

Demostración.
Como se definió en el capítulo 1, el orden de x mód n es el entero más pequeño s para el cual $x^s \equiv 1$ mód n y se denota por $ord_n x$. Se afirma que si $x \in J_n$, donde J_n es el conjunto de aquellos elementos en $\mathbb{Z}_n^*$ cuyo símbolo de Jacobi es 1, entonces $ord_n x$ es impar.
Si $s = ord_n x_i$ y $t = ord_n x_{i+1}$, se probará que $s = t$. Como $x_{i+1} \equiv x_i^2$ mód n, $x_i^s \equiv 1$ mód n y $x_{i+1}^t \equiv 1$ mód n, entonces

$$x_{i+1}^s \equiv x_i^{2s} \equiv 1 \text{ mód } n$$

de donde se concluye que $t|s$. Puesto que la sucesión tiene periodo π, el elemento x_i volverá a aparecer en ella: $x_i, x_{i+1}, \ldots x_0, x_i, \ldots$, es decir, $x_i \equiv x_{i+1}^{2^\pi}$ mód n, así

$$x_i^t \equiv x_{i+1}^{2^\pi t} \equiv 1 \text{ mód } n$$

por lo tanto $s|t$. En consecuencia $ord_n x_i = ord_n x_{i+1}$.
Ahora bien, supóngase que $ord_n x_i$ es par, esto es, $2^u | ord_n x_i$ y $2^{u+1} \not| ord_n x_i$, o equivalentemente $ord_n x_i = 2^u m$ con $(2, m) = 1$, de ahí que $x_i^{2^u m} \equiv 1$ mód n. Por la definición de la sucesión:

$$x_{i+1} \equiv x_i^2 \text{ mód } n$$

y por lo tanto

$$x_{i+1}^{2^{u-1}m} \equiv x_i^{2^u m} \equiv 1 \text{ mód } n$$

como $ord_n x_i = ord_n x_{i+1}$ de la relación anterior se sigue que $2^u m | 2^{u-1} m$, lo cual es una contradicción. Y por lo tanto el $ord_n x_i$ es impar.

Por la extensión de Carmichael al teorema de Euler, ver corolario A.2:

$$2^{\lambda(ord_n x_0)} \equiv 1 \text{ mód } ord_n x_0$$

Pero puesto que π es el periodo de la sucesión, y por lo tanto el entero más pequeño para el cual se cumple que $x_0 \equiv x_0^{2^\pi} \text{ mód } n$, en consecuencia, $2^\pi \equiv 1 \text{ mód } ord_n x_0$ y $\pi | \lambda(ord_n x_0)$. Además, ya que $x_0^{ord_n x_0} \equiv 1 \text{ mód } n$ y $x_0^{\lambda(n)} \equiv 1 \text{ mód } n$ entonces $ord_n x_0 | \lambda(n)$, $x_0 \in \mathbb{Z}_n^*$, por lo cual $\lambda(ord_n x_0) | \lambda(\lambda(n))$.

□

A continuación se indicarán las condiciones bajo las cuales es posible saber con exactitud el periodo de la sucesión.

Teorema 3.3 *Sea $n = pq$, $p \equiv q \equiv 3 \text{ mód } 4$, $x_0 \in Q_n$ y $\pi(x_0)$ el periodo de la sucesión. Si*

1. *n es tal que $ord_{\lambda(n)/2}(2) = \lambda(\lambda(n))$, y*
2. *$x_0 \in Q_n$ es tal que $ord_n(x_0) = \lambda(n)/2$.*

entonces $\lambda(\lambda(n)) | \pi(x_0)$.

Demostración.
Se sabe que $x_i \equiv x_0^{2^i} \text{ mód } n$. Puesto que π es el periodo de la sucesión, se tiene que

$$x_\pi \equiv x_0^{2^\pi} \equiv x_0 \text{ mód } n$$

o equivalentemente

$$x_0^{2^\pi - 1} \equiv 1 \text{ mód } n$$

Ahora bien, por la hipótesis 2 se tiene que $x_0^{\lambda(n)/2} \equiv 1 \text{ mód } n$. Por lo tanto $\dfrac{\lambda(n)}{2} | 2^\pi - 1$, es decir,

$$2^\pi \equiv 1 \text{ mód } \lambda(n)/2$$

Además por la hipótesis 1, $2^{\lambda(\lambda(n))} \equiv 1 \text{ mód } \lambda(n)/2$. Por consiguiente

$$\lambda(\lambda(n)) | \pi.$$

□

Las definiciones y el teorema que se dan a continuación proporcionan las características bajo las cuales es posible satisfacer la condición 1 del teorema anterior.

Definición 3.10 *Si p es un primo tal que $p = 2p_1 + 1$ y $p_1 = 2p_2 + 1$, con p_1 y p_2 primos impares, entonces se dice que p es un primo "especial".*

Definición 3.11 *Un entero $n = pq$, es "especial" si p y q son primos especiales distintos tales que $p \equiv q \equiv 3 \text{ mód } 4$.*

Obsérvese que cuando n es especial, ocurre que tanto p como q son mayores o iguales que 7.

Teorema 3.4 *Si n es especial y $2 \in Q_{p_1}$ o bien $2 \in Q_{q_1}$ entonces $ord_{\lambda(n)/2} = \lambda(\lambda(n))$.*

Demostración.
Puesto que n es un número especial por la definición de $\lambda(n)$ se tiene que $\lambda(n) = [p-1, q-1] = [2p_1, 2q_1] = 2p_1q_1$. También, $\lambda(\lambda(n)) = [2, p_1 - 1, q_1 - 1] = [2p_2, 2q_2] = 2p_2q_2$. Ahora bien, obsérvese que $\lambda(\lambda(n)/2) = 2p_2q_2$, es decir, $\lambda(\lambda(n)) = \lambda(\lambda(n)/2)$. Es fácil ver que $(2, \lambda(n)/2) = 1$ y por la extensión de Carmichael al teorema de Euler, ver colorario A.2, se tiene:

$$2^{\lambda(\lambda(n)/2)} \equiv 1 \text{ mód } \lambda(n)/2$$

pero como $\lambda(\lambda(n)) = \lambda(\lambda(n)/2)$, entonces

$$2^{\lambda(\lambda(n))} \equiv 1 \text{ mód } \lambda(n)/2$$

por consiguiente $ord_n(2)|\lambda(\lambda(n))$, esto es, $ord_{\lambda(n)/2}(2)|2p_2q_2$. Supóngase que $ord_{\lambda(n)/2}(2) \neq 2p_2q_2$, por consiguiente puede ocurrir alguno de los siguientes casos

1. $ord_{\lambda(n)/2}(2)|2p_2$ o bien $ord_{\lambda(n)/2}(2)|2q_2$
2. $ord_{\lambda(n)/2}(2) = p_2q_2$.

Si $ord_{\lambda(n)/2}(2)|2p_2$, entonces $2^{2p_2} \equiv 1 \text{ mód } \lambda(n)/2$, es decir

$$\begin{aligned} 2^{2p_2} &\equiv 1 \text{ mód } p_1q_1 \\ 2^{2p_2} &\equiv 1 \text{ mód } q_1 \end{aligned}$$

pero también $2^{2q_2} \equiv 1 \text{ mód } q_1$, por el teorema de Euler, ya que $2q_2 = q_1 - 1$. Entonces $2^{(2p_2, 2q_2)} \equiv 1 \text{ mód } q_1$, por consiguiente $2^2 \equiv 1 \text{ mód } q_1$, lo cual no puede ser ya que $q_1 \geq 7$. Similarmente ocurre para el caso en el que $ord_{\lambda(n)/2}(2)|2q_2$.

Ahora bien, si se supone que $ord_{\lambda(n)/2}(2) = p_2q_2$, entonces $2^{p_2q_2} \equiv 1 \bmod p_1q_1$ de donde $2^{p_2q_2} \equiv 1 \bmod q_1$. Puesto que q_2 es impar, $2^{q_2} \not\equiv -1 \bmod q_1$, y de ahí que $2^{(q_1-1)/2} \not\equiv -1 \bmod q_1$. Por lo tanto, 2 es un residuo cuadrático módulo q_1. Análogamente, 2 es un residuo cuadrático módulo p_1, lo cual no puede ser ya que 2 es un residuo cuadrático, pero sólo con a lo más uno de los primos p_1 o q_1.

Por lo tanto se tiene la afirmación del teorema.

□

El generador Blum-Blum-Shub, es recomendado en el RFC 1750 [RFC95], pero sólo en el caso en el que se requiere un nivel alto de seguridad, pues aunque no es muy rápido, el hecho de que su construcción esté basada en el problema de los residuos cuadráticos, garantiza que aunque se posea cierta porción de la sucesión, no es posible predecir el siguiente bit de la misma. Por ejemplo, para generar los primos del RSA, podría usarse, ya que estos no se generan tan a menudo.

Capítulo 4

Generador Blum-Micali

En el presente capítulo se describe la construcción y funcionamiento del generador de sucesiones pseudoaleatorias criptográficamente fuertes propuesto por Manuel Blum y Silvio Micali, quienes de hecho introdujeron dicho concepto. En su artículo [BLU84] definen qué es un *generador pseudoaleatorio criptográficamente fuerte*, establecen cuáles son las propiedades que debe tener, proporcionan un esquema general para construir generadores de este tipo y por último utilizando dicho esquema diseñan un generador específico basado en el problema del logaritmo discreto, que se revisará en la primera sección de este capítulo.

4.1. Conceptos básicos

En esta sección se introducen algunos conceptos de Teoría de Números, que son fundamentales para la construcción del generador. También se explica brevemente el *modelo computacional no uniforme* utilizado por los autores para medir la complejidad tanto del algoritmo que genera la sucesión, como para medir la eficiencia de un algoritmo que intentase predecir el siguiente bit de la sucesión. Dicho modelo está basado en circuitos booleanos.

4.1.1. Preliminares matemáticos

Definición 4.1 *Sea G un grupo cíclico de orden n, α un generador de G, y $\beta \in G$. El logaritmo discreto base α de β denotado $\log_\alpha \beta$, es el único entero x, $0 \leq x \leq n-1$, tal que $\beta = \alpha^x$.*

Definición 4.2 *El* **problema generalizado del logaritmo discreto** *(PGLD) es el siguiente: dado un grupo cíclico finito G de orden n, un generador α de G, y un elemento $\beta \in G$, encontrar el entero x, $0 \leq x \leq n-1$, tal que $\beta = \alpha^x$.*

Es muy común en criptografía tomar G como $\mathbb{Z}_p^*$, esto es, el grupo de las unidades del anillo $\mathbb{Z}_p$ donde p es un número primo. Otro ejemplo es el grupo aditivo conformado por el conjunto de los puntos de una curva elíptica cuando esta satisface determinadas condiciones.

Actualmente, existen varios algoritmos que resuelven el problema del logaritmo discreto para algunos casos. Sin embargo ninguno de ellos resuelve el PGLD eficientemente [MEN97]. Entre dichos algoritmos se encuentran los siguientes:

- *búsqueda exhaustiva,*
- *baby-step, giant-step,*
- *Pollard rho,*
- *Pohlig-Hellman y*
- *cálculo de índices.*

El algoritmo del baby step, giant step, fue descrito por Heiman [HEI93], el método de Pollard, fue propuesto por su autor en [POL75]. Pohlig y Hellman dieron a conocer el algoritmo que lleva su nombre en [POH78]. Acerca del cálculo de índices se tienen varias referencias en donde se pueden hallar distintas variantes [ADL79, COP86, HEL83]. Además de estas referencias básicas, Menezes menciona más al final del capítulo 3 de [MEN97].

Proposición 4.1 *Dado un generador $g \in \mathbb{Z}_p^*$, un elemento $\alpha \in \mathbb{Z}_p^*$ es un* **residuo cuadrático módulo p** *i.e. $\alpha \in Q_p$ si y sólo si $\alpha = g^{2s} \bmod p$ para algún entero $1 \leq s \leq (p-1)/2$. Tal representación de α es única. Más aún α tiene dos raíces cuadradas módulo p : $g^s \bmod p$ y $g^{s+(p-1)/2} \bmod p$.*

Demostración.
Puesto que $\alpha \in Q_p$, entonces existe $t \in \mathbb{Z}_p^*$ tal que $\alpha \equiv t^2 \bmod p$. Dado que $t \in \mathbb{Z}_p^*$, es posible expresarlo de la siguiente manera: $t = g^i$ para algún $1 \leq i \leq p-1$. Por tanto,

$$\alpha \equiv g^{2i} \text{ mód } p$$

Pero $2i$ también es un entero tal que $1 \leq 2i \leq (p-1)$. Es decir, $1 \leq i \leq (p-1)/2$.

Recíprocamente, sea $\alpha \equiv g^{2s}$ mód p para algún entero $1 \leq s \leq (p-1)/2$. Entonces,

$$\alpha \equiv (g^s)^2 \text{ mód } p$$

Esto es, α es un residuo cuadrático.

□

Definición 4.3 *Sea g un generador para $\mathbb{Z}_p^*$, $\alpha \in Q_p$ y $2s$ el único exponente tal que* $1 \leq 2s \leq p-1$. *Entonces g^s* mód p *será denominada la g-raíz cuadrada principal de α, y $g^{s+(p-1)/2}$* mód p *la g raíz cuadrada no principal de α.*

Sea g es un generador de $\mathbb{Z}_p^*$, nótese que dado un residuo cuadrático módulo p, digamos α, pero no el índice de α base g, es posible con un método eficiente probar si α es un residuo cuadrático, y de igual manera es posible hallar las dos raíces módulo p. Sin embargo el decidir cuál de ellas es la raíz cuadrada principal es un problema mucho más difícil de resolver. De hecho si se tuviera un mecanismo para averiguar cuál es la raíz cuadrada principal, entonces sería igualmente fácil, resolver el problema del logaritmo discreto.

Ejemplo.
Sea $p = 23$, y $g = 14$, es fácil saber, calculando el símbolo de Legendre, que 9 es un residuo cuadrático módulo 23. Más aún, es posible utilizar un algoritmo probabilístico para averiguar las raíces cuadradas, que son 3 y 20. Sin embargo, es difícil saber cual es la raíz cuadrada principal, dado que para ello se tendría que averiguar la x tal que $14^x \equiv 3$ mód 23 y la y tal que $14^y \equiv 20$ mód 23, para así poder determinar cual de los dos exponentes es mayor que 1 pero menor $(p-1)/2$. Es decir, sería necesario tener un algoritmo eficiente para resolver el problema del logaritmo discreto.

4.1.2. Familias de circuitos no uniformes

En el capítulo 1, se introdujo una forma de medir la eficiencia de los algoritmos, conocido como *modelo computacional uniforme*. A continuación se

describe el modelo no uniforme, basado en circuitos booleanos. En el trabajo de Blum y Micali [BLU84] se argumenta que dicho modelo es más fuerte que el modelo uniforme pues ayuda a visualizar que un adversario aún con todos los recursos existentes, no podría resolver un problema dado. Puesto que aquí sólo se darán los linemientos generales, puede revisarse [LAG90, KRA86, GOL01, MOT95], para mayor información al respecto.

Definición 4.4 *Un* **circuito booleano** *es un grafo dirigido acíclico con las siguientes características:*

1. *Un conjunto de n vértices, a los que se les denominará vértices de entrada con grado 0, puesto que a ellos no llega ninguna arista. A cada uno de los vértices se les etiquetará con las variables $x_1, x_2, \ldots x_n$. Cada una de tales x_i podrá tomar un valor de 0 ó 1, exclusivamente.*

2. *Un conjunto de vértices internos, denominados vértices de salida con grado 1, lo cual indica que a estos pueden llegar varias aristas, pero saldrá exclusivamente una arista. Cada uno estos vértices internos, se etiquetará con una función booleana: AND, OR, NOT, o si se prefiere $\wedge, \vee, \neg$. Al vértice marcado con NOT ó $\neg$, sólo podrá llegar una arista.*

3. *Existirá solamente un vértice con grado de salida 0, el cual será la salida del circuito.*

La salida del circuito será el resultado de haber calculado una función booleana $f(x_1, \ldots, x_n)$. Por ejemplo si a un vértice etiquetado con *AND* o $\wedge$, llegan aristas de los vértices etiquetados con x_1, x_3, x_5, entonces la salida será $x_1 \wedge x_3 \wedge x_5$. Análogamente sucede para nodos marcados con $\vee$ y $\neg$. La salida del circuito se lee en el nodo de salida.

Cabe mencionar que a los nodos o vértices también se les denomina *compuertas*, en algunos textos. El **tamaño del circuito** estará dado por el número de vértices o compuertas en él. Un **circuito de tamaño mínimo**, será aquel que tenga el número mínimo de vértices o compuertas.

Ejemplo.
El circuito que se muestra en la figura 4.1, calcula la función booleana:

$$f(x_1, \ldots, x_5) = (x_1 \wedge x_2 \wedge x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_5)$$

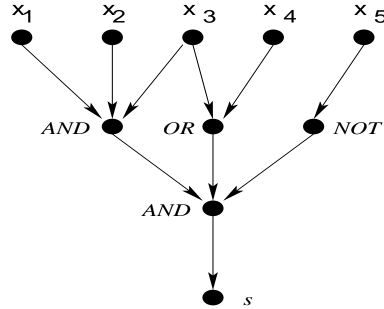

Figura 4.1: Ejemplo de un circuito booleano

Por ejemplo, si se le dan valores de 0 y 1 a las x_i's se tiene que $f(0,1,0,1,0) = 0 \wedge 1 \wedge 1 = 0$. Es evidente que un circuito booleano de esta naturaleza calcula una función $f : \{0,1\}^n \longrightarrow \{0,1\}$.

Además de los circuitos booleanos mencionados anteriormente, también existen los *circuitos aleatorios* [KRA86, MOT95]. Tales circuitos son muy similares a los anteriores, salvo que se tendrán m vértices con entrada de grado 0, a los cuales se les dará valores de 0 ó 1 al azar, a éstas se les denomina *entradas al azar* o también *testigos*. Se dice que un circuito aleatorio calcula la función f, con entradas $x_1, \ldots, x_n$ si cumple las siguientes características:

1. *Dadas las entradas* $x_1, \ldots, x_n$, *si* $f(x_1, \ldots, x_n) = 0$, *entonces la salida del circuito aleatorio también deberá ser 0, ignorando las entradas al azar.*

2. *Dadas las entradas* $x_1, \ldots, x_n$, *si* $f(x_1, \ldots, x_n) = 1$, *entonces la salida del circuito será 1 con un probabilidad de al menos 1/2.*

A lo largo de este capítulo se asumirá que los circuitos de los cuales se habla son aleatorios, aunque no se indique explícitamente. A continuación se dará una definición que permitirá medir la eficiencia del circuito en términos de su tamaño, es decir, del número de vértices que posee. Esto es similar a lo que se hace al medir la eficiencia de los algoritmos determinísticos (los cuales se mencionaron en el capítulo 1), en relación al tiempo que tardan en ejecutarse. Al igual que el tiempo de los algoritmos determinísticos se acotaba por arriba con alguna función f, siendo f por ejemplo un polinomio; al tamaño de los circuitos booleanos se le acotará por arriba con una función, que también puede ser un polinomio. Dicho polinomio está dado en términos

de una sola variable, y aunque no resulta importante cómo están dados los coeficientes, se puede asumir que son coeficientes reales y que el grado de dicho polinomio es mayor o igual que 1. Esto último se aplicará a lo largo de este capítulo para cualquier polinomio que se mencione.

Definición 4.5 *Una familia de circuitos de tamaño polinomial* $\mathcal{C} = \{C_1, C_2, \ldots, \}$ *se define como una sucesión infinita de circuitos booleanos tal que para todo* $n \in \mathbb{N}$*, el circuito* C_n *tiene* n *vértices de entrada y su número de vértices de* C_n *está acotado por arriba por algún polinomio* $p(n)$*.*

Los circuitos booleanos serán de suma importancia en el resto del capítulo, ya que a través de ellos, se probará que un generador es eficiente y que además es impredecible.

4.2. Predicados no aproximables

Puesto que la construcción de un generador criptográficamente fuerte, de acuerdo con Blum y Micali [BLU84], está basada en *predicados*; y dado que éstos deben cumplir determinadas cualidades para poder obtener un generador a ser usado en aplicaciones criptográficas, en esta sección se analizarán dichas cualidades. Se comenzará por dar la definición de predicado.

Un predicado B*, es una función* $B : \{0,1\}^n \longrightarrow \{0,1\}$*. Sin embargo en este caso, nos interesan los predicados que pertenecen al conjunto* $\mathcal{B} = \{B_i : D_i \rightarrow \{0,1\} | i \in S_n, n \in \mathbb{N}\}$ *donde*

- S_n *es un subconjunto de los enteros cuya representación en binario es con exactamente* n *bits, siendo siempre 1 el más significativo.*
- D_i *es a su vez el subconjunto de los enteros cuya representación en binario es con a lo más* i *bits, donde a su vez* $i \in S_n$*, sin importar si el bit más significativo es o no 0.*

Es importante resaltar que aunque un elemento de D_i no tiene la restricción de tener el bit más significativo igual a 1, su representación en binario será con exactamente n bits, donde los más significativos pueden ser 0's. Aunque ahora puede ser ambiguo qué propiedades, además de las ya mencionadas, puede tener un elemento de S_n o de D_i, más adelante esto se aclarará.

Definición 4.6 *Sea* $I_n = \{(i, x) | i \in S_n, x \in D_i\}$.

A cada elemento en I_n i.e. cada pareja (i, x), se le denomina una entrada de tamaño n. Obsérvese que I_n es el conjunto de todos los elementos de cada D_i. Esto será de utilidad para definir el dominio de un predicado booleano.

Definición 4.7 *Un conjunto de predicados* $\mathcal{B}$ *es* **accesible** *si existen dos constantes* c_1 *y* c_2 *y un algoritmo probabilístico* A *tal que al recibir* $n \in \mathbb{N}$ *como entrada, se detiene después de* n^{c_1} *pasos. La salida de dicho algoritmo será alguna de las siguientes dos opciones:*

1. *"?", o bien*
2. *Un elemento* $(i, x) \in I_n$

lo primero ocurrirá con una probabilidad $\frac{1}{2^{c_2}}$, *y lo segundo con una probabilidad uniforme.*

La definición anterior nos indica que para tener un predicado adecuado en la construcción de un generador criptográficamente fuerte, debe existir un algoritmo probabilístico que logre hallar los parámetros del tamaño requerido y en base a ellos definir al conjunto D_i y por lo tanto al predicado B. El algoritmo A debe ser eficiente, esto es, el hallar a tales parámetros no debe tomar demasiado tiempo, dicho de una manera más formal, tal algoritmo debe obtener resultados en un tiempo polinomial.

Definición 4.8 *Sea* $\mathcal{B}$ *un conjunto de predicados y* p *un polinomio. Sea* c_n^p *el tamaño del circuito mínimo* C, *que calcula correctamente* $B_i(x)$ *donde* $B_i \in \mathcal{B}$, *es decir,*

$$C[i, x] = B_i(x)$$

para al menos la fracción $\frac{1}{2} + \frac{1}{p(n)}$ *de las entradas* (i, x) *de tamaño* n. *Tal circuito se conoce como un* **circuito** $\mathbf{\frac{1}{p(n)}}$ **aproximable** *a* $\mathcal{B}$.

Definición 4.9 *Un conjunto de predicados* $\mathcal{B}$ *se dice que es* **no aproximable**, *si para todo polinomio* p, c_n^p *crece mucho más rápido que cualquier polinomio en* n.

El tipo de predicado utilizado en la construcción de generadores deber ser justamente, accesible y no aproximable. Esto significa que hallar los parámetros para su construcción debe hacerse de una forma rápida, y dado que el algoritmo es probabilístico, debe equivocarse un número muy pequeño de veces. Además de ello el predicado debe ser no aproximable, esto es, cualquier circuito que se tome para calcular $B(x)$, tendrá un tamaño demasiado grande. Otra manera de decirlo, sería que conociendo la pareja $(i, x) \in I_n$ es computacionalmente imposible, saber si $B(x) = 0$ o $B(x) = 1$.

A continuación, se realizarán un par de ejemplos, de predicados accesibles y no aproximables. Aunque no se demostrará que en efecto así es, deberá se supondrá que debería ser claro recordando algunos conceptos que también se abordaron en el capítulo anterior.

Ejemplo

Inicialmente se escoge una $n \in \mathbb{N}$, y posteriormente se definen S_n, D_i y B_i como sigue:

$$\begin{array}{lcl} S_n & = & \{i \in \mathbb{Z}^+ | i = pq, p \neq q, |p| = |q|, |i| = n\} \\ D_i & = & \{x \in \mathbb{Z}_i^* | i \in S_n, \left(\frac{x}{i}\right) = 1\} \\ B_i(x) & = & \begin{cases} 1 \text{ si } x \text{ es un residuo cuadrático módulo } i \\ 0 \text{ en otro caso} \end{cases} \end{array}$$

donde $\left(\frac{x}{i}\right)$ denota el símbolo de Jacobi (ver teorema A.10). Es importante señalar que se está abusando de la notación, dado que al denotar p y q, se está considerando la expansión binaria tanto de p como de q. Es fácil ver que el predicado B_i es accesible, dado que existen algoritmos probabilísticos con un tiempo de ejecución polinomial para hallar a los primos p y q. Y también existe un algoritmo eficiente para calcular el símbolo de Jacobi. [MEN97]
También es fácil ver que el predicado es no aproximable, dado que aún conociendo a i y a $x \in D_i$ es computacionalmente imposible averiguar si x es o no residuo cuadrático. Hecho que se puede establecer en términos de circuitos booleanos.
A continuación se mostrará cómo obtener B_i específicos, con la finalidad de ayudar a comprender la notación usada líneas arriba.

Ejemplo.

Tomando las definiciones anteriores, si $n = 6$ entonces $S_6 = \{35\}$, es decir, S_6 contiene a un único elemento que es el producto de dos primos distintos de la misma longitud, en este caso $p = 5$ y $q = 7$. Debido a que S_6 es un

conjunto con un solo elemento, lo único que resta por hacer es construir el dominio D_{35}, formalmente se tiene que:

$$D_{35} = \{1, 3, 4, 9, 11, 12, 13, 16, 17, 27, 29, 33\}$$

Ejemplo.
A continuación se construirá un ejemplo para la definición dada anteriormente, en la cual S_n tiene más de un elemento.

Si $n = 9$ entonces tenemos que S_9 serán los enteros que sean representables con 9 bits, donde el más significativo siempre es 1, y donde dichos enteros son el producto de dos primos que sean representables por el mismo número de bits. Los primos de igual longitud y cuyo producto es un entero de 9 bits son: 17, 19, 23, 29 y 31. Al calcular el producto de parejas de dichos números se tiene que:

$$S_9 = \{323, 391, 437, 493\}$$

de donde se escoge una i, supóngase que tal $i = 323$, entonces se obtiene el conjunto D_{323}[1] que se muestra a continuación:

$$\begin{gathered}D_{323} = \{1, 3, 4, 9, 10, 12, 14, 16, 22, 25, 26, 27, 29, 30, 31,\\ 35,36,37,40,41,42,43,46,47,48,49,55,56,64,65,66,71,\\ 75,77,78,79,81,83,87,88,90,91,93,97,100,101,104,105,\\ 106,107,108,109,111,113,115,116,118,120,121,122,123,\\ 124,126,129,134,137,138,140,141,143,144,146,147,148,\\ 149,157,160,161,164,165,167,168,169,172,173,178,181,\\ 184,188,191,192,193,195,196,198,206,211,213,220,224,\\ 225,227,229,231,234,237,239,241,243,249,250,251,253,\\ 254,256,260,261,262,263,264,265,269,270,271,273,278,\\ 279,284,290,291,295,299,300,302,303,305,308,310,312,\\ 315, 316, 317, 318, 321\}\end{gathered}$$

Para este conjunto se sabe i pero no cuáles son sus factores primos y si se toma cualquier elemento $x \in D_i$, por ejemplo 265, nos tomaría algún tiempo determinar si x es o no residuo cuadrático, es decir averiguar $B_i(x)$. Sin embargo si se escogiesen p y q de 1024 bits o de 2048 bits, no sería posible averiguar con éxito si $x \in D_i$ es residuo cuadrático, es decir, $B_i(x)$.

[1]Este conjunto se obtuvo con el programa que aparece en la sección B.3 del apéndice B.

4.3. Construcción de un GBPCF

A continuación se establecen las condiciones necesarias para construir un generador de bits pseudoaleatorios criptográficamente fuerte de acuerdo con el trabajo de Blum y Micali [BLU84].Para ello es necesario definir el término *generador de bits pseudoaleatorios criptográficamente fuerte, GBPCF.*
Es importante señalar que los polinomios que se mencionan en cada una de las definiciones y teoremas, tienen coeficientes en los reales y son de una sola variable.

Sean P y P_1 polinomios, $S = \{S_k\}$ una colección de conjuntos tal que S_k es el conjunto de sucesiones de bits de longitud $P(k)$. Un *colección de predicción* $\mathcal{C} = \{C_k^i\}$ es una colección de circuitos booleanos tal que cada circuito C_k^i tiene menos de $P_1(k)$ compuertas, con i entradas booleanas, tal que $i < P(k)$, y una salida booleana.

Ahora considere que las i entradas booleanas del circuito C_k^i son los primeros i bits de una sucesión s tomada al azar de S_k, dicho circuito generará como salida un bit b. La probabilidad de que dicho bit b, sea el bit $i+1$ en la sucesión s, será denotada por $p_{k,i}^{\mathcal{C}}$.

Se dice entonces que la colección S pasa la *prueba del siguiente bit* si para todas las posibles colecciones de predicción $\mathcal{C}$, todos los polinomios Q, todas las k suficientemente grandes y todas las $i < P(k)$ se tiene que:

$$p_{k,i}^{\mathcal{C}} < \frac{1}{2} + \frac{1}{Q(k)}$$

Definición 4.10 *Sea* $I = \{(i,x) \in I_n | n \in \mathbb{N}\}$

Definición 4.11 *Sea Q un polinomio, I un conjunto de cadenas binarias e $I_k \subseteq I$ el conjunto de cadenas binarias de longitud k. Sea A un algoritmo determinístico que recibe como entrada una semilla $x \in I_k$ y produce como salida una sucesión s_x con longitud $Q(k)$. Sea $S_k = \{s_x | x \in I_k\}$, es decir, el conjunto de sucesiones generadas por las semillas $x \in I_k$. El algoritmo A es un generador $Q - CFBP$, esto es, Q criptográficamente fuerte de bits pseudoaleatorios; si la colección $S = \{S_k\}$ pasa la prueba del siguiente bit.*

A continuación se enunciará un teorema propuesto y demostrado por Blum y Micali en [BLU84], el cuál nos indica cómo construir generadores criptográficamente fuertes.

Teorema 4.1 *Sean Q un polinomio, $\mathcal{B} = \{B_i : D_i \rightarrow \{0,1\} | i \in S_n, n \in \mathbb{N}\}$, un conjunto de predicados accesible e inaproximable e $I = \{(i,x) \in I_n | n \in \mathbb{N}\}$, el conjunto de todas las entradas relativas a $\mathcal{B}$. Dadas las siguientes funciones*

1. *$f : I \rightarrow D_i$ una función computable en tiempo polinomial,*
2. *$f_i : D_i \rightarrow D_i$ una permutación para toda $i \in S_n$,*
3. *$h : I \rightarrow B(f_i(x))$, un predicado computable en tiempo polinomial.*

Entonces es posible construir un generador Q-CFBP.

4.4. Construcción del generador Blum-Micali

Una vez planteado el esquema general para construir un GBPCF, en esta sección se explicará, la construcción del generador Blum-Micali dada en [BLU84] originalmente. Aunque también es posible hallar un enfoque ligeramente diferente en [KRA86]. La principal razón por la cual, tal generador podría ser usado en aplicaciones criptográficas es que el predicado es no aproximable, dado que en su construcción interviene el problema del logaritmo discreto, enunciado en la primera sección de este capítulo. Al final de esta sección se darán algunas razones por las que éste generador no es muy usado. Sin embargo sigue siendo de gran importancia teórica.

Definición 4.12 *Sea g un generador de $\mathbb{Z}_p^*$, $x \in \mathbb{Z}_p^*$. Se define el predicado $B_{p,g}(x)$ como sigue:*

$$B_{p,g}(x) = \begin{cases} 1 & \textit{si } x \textit{ es la raíz cuadrada principal mod } p \\ 0 & \textit{en otro caso} \end{cases}$$

Obsérvese en la definición anterior que si se conoce el exponente s tal que $x = g^s \bmod p$ es fácil evualuar $B_{p,g}$, solamente se necesita saber si $s \leq (p-1)/2$ o no. A continuación se mencionarán algunos resultados importantes de [BLU84]

Teorema 4.2 *Sea MB_Q un oráculo tal que para todos los primos p y para todos los generadores en $\mathbb{Z}_p^*$, puede hallar $B_{p,g}(x)$ con una probabilidad mayor o igual a $\frac{1}{2} + \frac{1}{Q(|p|)}$, es decir:*

$$Pr(MB_Q[p,g,x] = B_{p,g}) \geq \frac{1}{2} + \frac{1}{Q(|p|)},$$

donde $Q(|p|)$ es el resultado de evaluar el polinomio Q en la longitud de p, y $x \in \mathbb{Z}_p^*$. Entonces existe un algoritmo probabilístico con un oráculo MB_Q tal que para todos los primos p resuelve el problema del logaritmo discreto en un tiempo $\mathbf{poly}(|p|)$.

El oráculo al que se refiere el teorema anterior no es más que una subrutina en un algoritmo probabilístico para averiguar el logaritmo discreto.

En el artículo [BLU84] se da el siguiente teorema:

Teorema 4.3 *Bajo la intratabilidad del problema del logaritmo discreto es posible construir un generador de bits pseudoaletarios criptográficamente fuerte.*

A continuación se explicará la construcción del generador Blum-Micali, el cual es criptográficamente fuerte. Para ello es necesario definir cuáles son los conjuntos específicos S_n, D_i e I_n introducidos en la sección 4.2, a usarse en éste caso. En este ejemplo, se verá claramente que tanto S_n como D_i, se eligen cuidadosamente para que el generador tenga la propiedad de ser criptográficamente fuerte. Para definir al generador Blum-Micali se necesitan los siguientes conjuntos:

$$\begin{aligned} S_{2n} &= \{p \parallel g |\ \text{p primo, g generador de } \mathbb{Z}_p^*, |p| = n, |g| = n, n \in \mathbb{N}\}, \\ D_i &= \mathbb{Z}_p^*, \\ I_n &= \{(i,x) | i \in S_{2n}, x \in D_i\} \end{aligned}$$

donde $p \parallel g$ denota la concatenación de p y g, parámetros necesarios para definir al grupo $\mathbb{Z}_p^*$. Nuevamente abusando de la notación ya que se está considerando la representación binaria de p y g. También es importante señalar que para obtener tanto a p como a g, existe un algoritmos probabílistico con un tiempo de ejecución polinomial, el cual se mencionará en la demostración del teorema 4.4.

Definición 4.13 *El generador Blum-Micali se define como la permutación $f_i : \mathbb{Z}_p^* \to \mathbb{Z}_p^*$ para alguna i, donde $f_i(x_i) = g^{x_i} \bmod p$ y $B_i(x) = B_{p,g}(x)$. La sucesión está dada por: $s = B_{p,g}(x_0), B_{p,g}(x_1), B_{p,g}(x_2), \ldots, B_{p,g}(x_m)$*

Con la notación y definiciones anteriores se tiene el siguiente teorema.

Teorema 4.4 *El generador Blum-Micali es un generador criptográficamente fuerte.*

Demostración.
Para probar que el generador Blum-Micali es un generador criptográficamente fuerte, es necesario demostrar que el conjunto de predicados

$$\mathcal{B} = \{B_i : \mathbb{Z}_p^* \to \{0,1\} | i \in S_{2n}, n \in \mathbb{N}\}$$

es accesible y no aproximable. Para demostrar que $\mathcal{B}$ es accesible, es decir que existe un algoritmo probabilístico eficiente para obtener al azar un primo p y la tripleta (p, g, x), se considerarán los siguientes resultados:

Definición 4.14 *Un número primo seguro p es un primo de la forma $p = 2q + 1$ donde q es a su vez primo.*

Existe un algoritmo probabilístico eficiente para obtener un número primo y un generador de $\mathbb{Z}_p^*$ al mismo tiempo, el cual aparece en el capítulo 4, sección 6.1 de [MEN97] y se muestra a continuación:

Entrada: la longitud k requerida del primo,

1. Repetir

   a) Seleccionar al azar un primo q de longitud $k - 1$.

   b) Calcular $p = 2q + 1$ y probar si p es primo

   hasta que p sea primo.

2. Repetir

   a) Seleccionar un elemento $g \in \mathbb{Z}_p^*$

   b) Calcular $b_1 = g^2$ y $b_2 = g^q$

   hasta que b_1 y b_2 sean distintos de 1.

3. Regresar (p, g)

Salida: un primo p de longitud k y un generador $g \in \mathbb{Z}_p^*$.

A la fecha se conocen varios algoritmos probabilísticos eficientes para determinar si un número entero es o no primo [MEN97]. Por lo tanto, encontrar un primo q no será problema. Además es conocido que el algoritmo arriba mencionado selecciona una pareja con probabilidad uniforme.

Continuáremos ahora con la demostración del teorema 4.4. Procediendo por contradicción, supóngase ahora que $\mathcal{B}$ es un predicado aproximable, esto implicaría que existe un circuito C de tamaño polinomial de tal manera que averigua correctamente $B_{p,g}(x)$, con una probabilidad mayor o igual a $\frac{1}{2} + \frac{1}{P_1}$ para algún polinomio P_1, si recibe como entrada la tripleta (p, g, x). Sin embargo por el teorema 4.2 eso implicaría que existe un algoritmo probabilístico eficiente para resolver el problema del logaritmo discreto, llegando así a una contradicción.

□

A continuación se dará un par de ejemplos de cómo construir un generador Blum-Micali en particular, y de cómo generar la sucesión.

Ejemplo.
Sea $n = 4$, por lo que los primos posible son 17, 19, 23,29 y 31, ya que estos se pueden representar en palabras de 4 dígitos binarios siendo el más significativo 1. Observe que de estos 23 es un *primo seguro* puesto que $23 = 2(11)+1$, donde 11 a su vez es también primo, esta observación facilita la labor para encontrar un generador en $\mathbb{Z}_{23}^*$. A continuación se listan los generadores para cada uno de los $\mathbb{Z}_p^*$.

$$\begin{aligned}
G_{17} &= \{3, 5, 6, 7, 10, 11, 12, 14\} \\
G_{19} &= \{2, 3, 10, 13, 14, 15\} \\
G_{23} &= \{5, 7, 10, 11, 14, 15, 17, 19, 20, 21\} \\
G_{29} &= \{2, 3, 8, 10, 11, 14, 15, 18, 19, 21, 26, 27\} \\
G_{31} &= \{3, 11, 12, 13, 17, 21, 22, 24\}
\end{aligned}$$

Por lo tanto $S_{2n} = \{17 \parallel 3, 17 \parallel 5, 17 \parallel 6, \ldots 19 \parallel 2, \ldots, 31 \parallel 3 \ldots 31 \parallel 24\}$ o bien si se prefiere en palabras binarias $17 \sim 3$ se vería 10010011 observe que los bits más significativos del generador son ceros y tenemos una palabra binaria de longitud 8. Lo mismo ocurriría con el resto de los elementos del conjunto S_{2n}.

A continuación se escoge un elemento del conjunto anterior, por ejemplo $29 \sim 3$ junto con una $x \in \mathbb{Z}_{23}^*$, por ejemplo 1. Con esta x se puede comenzar a generar la sucesión, realizando la operación g^x mód $p = 3^1$ mód $29 = 3$, luego tomamos este nuevo resultado como x y repetimos la operación 3^3 mód $29 = 27$ y así sucesivamente. En la siguiente tabla se muestran todos los resultados, hasta antes de que se repita nuevamente el 1.

$x_i = g^{x_{i-1}}$	$B_{p,g}(x_i)$
3	1
27	0
10	1
5	1
11	1
15	0
26	0
13	1
19	0
18	0
6	1
4	1
23	0
8	1
7	1
12	1
16	0
20	0
25	0
14	1

De esta forma la sucesión se vería como sigue: $1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, \ldots$

En la sección B.3 del apéndice B, se incluye el código de un programa en Java que implementa este generador.
Es importante mencionar que el cálculo para obtener un bit, es demasiado costoso. Puesto que el exponente es grande, su aritmética es computacionalmente más costosa que la del generador Blum-Blum-Shub, en el que el exponente es 2. Por ello aunque el generador es criptográficamente fuerte,

casi no se usa, salvo en aplicaciones que requieren un alto nivel de seguridad y no demasiada eficiencia.

Capítulo 5

Comentarios adicionales

En los capítulos anteriores, se abordaron algunas formas de generar sucesiones, una de ellas no apta para aplicaciones criptográficas y dos más que sí lo son. Sin embargo, existen otros métodos de generar sucesiones, que son muy usados actualmente en protocolos de seguridad y en algoritmos criptográficos, los más importantes se mencionarán en la primera sección de este capítulo.

También es importante señalar que las sucesiones no solamente se utilizan en criptografía. Actualmente, las sucesiones tiene una amplia gama de aplicaciones, como por ejemplo en teoría de códigos, comunicación de banda ultra ancha (ultrawide band communication), *Acceso Múltiple por División de Código (CDMA)*, sistemas de posicionamiento global, entre otros. En la segunda sección se describe brevemente de que se tratan algunas de estos usos.

5.1. Otras formas de generar sucesiones

Los generadores de sucesiones que se han descrito en el presente trabajo, a pesar de brindar una amplia seguridad, no son muy usados, puesto que la aritmética es demasiado costosa, computacionalmente hablando. Por ello, los estándares para generar sucesiones como el *RFC 1750* [RFC95] recomiendan algunos otros métodos más eficientes. En esta sección se explican brevemente los más representativos de ellos.

Además, de los métodos que se recomiendan en los estándares, están diseñándose nuevas formas de generar sucesiones para usarse en seguridad

informática. Uno de tales métodos, es aquel basado en curvas elípticas, éste y algunos otros también se describen brevemente.

5.1.1. Sucesiones generadas por hardware

Como se mencionó en el capítulo 1, un generador de sucesiones pseudoaletorias, necesita usar un cantidad de bits que sea aleatoria. Para obtener tal cantidad aleatoria es posible usar hardware especial diseñado con tal propósito, sin embargo no todas las computadoras lo poseen y adquirirlo es demasiado costoso. Otra opción es usar el hardware que cualquier computadora posee, para lo cual se tienen varias opciones.

Una de las formas ampliamente usadas para generar cantidades realmente aleatorias es medir el tiempo y contenido del movimiento del ratón, o los teclazos de un usuario. Estos mecanismos suelen considerarse una buena fuente de aleatoriedad, de hecho es la técnica utilizada por PGP para la generación de claves [STA03]. Sin embargo una aplicación que use estos parámetros depende de que haya un usuario frente a la máquina, hecho que en algunas ocasiones puede ser una desventaja. También es posible tomar los bits producidos por los dispositivos de video o de audio y después al conjunto de bits así obtenido pasarlo por un programa de compresión para quitar la posible redundancia que exista. [RFC95]

Una técnica adicional es utilizar la variación que existe en la velocidad de rotación del motor de la unidad de disco. Davis, Ihaka y Fenstermacher [DAV94] averiguaron que dicha variación en la velocidad aunque pequeña era irregular, debido a la turbulencia de aire dentro de la unidad de disco. Realizando algunas mediciones sobre tal variación, y realizando un procesamiento adicional es posible generar aproximadamente 100 bits por minuto.

5.1.2. Funciones de un sólo sentido

Otra técnica muy usada para generar sucesiones pseudoaleatorias es tomar cierto conjunto de bits, obtenidos por hardware de la forma en que se explicó al principio de la sección anterior, y posteriormente a dicho conjunto de bits, aplicarle una función hash. Actualmente existen un par de funciones hash estándar que son ampliamente usadas en aplicaciones criptográficas y son la MD5 y la SHA, de las cuales se puede saber más en

[KAU95, MEN97, STA03] o inclusive algún método de cifrado de clave privada (como DES o AES) [RFC95]. Esto es porque tanto las funciones hash como los métodos de cifrado son considerados como *funciones de un sólo sentido (one way functions).* Una función de un sólo sentido $f : X \longrightarrow Y$, es aquella para la cual se calcula fácilmente $f(x)$, para cualquier $x \in X$, pero es computacionalmente imposible calcular $f^{-1}(y)$, para casi todas las y.

Aunque no se ha demostrado formalmente que las funciones hash, puedan ser usadas como un generador de sucesiones criptográficamente fuertes, es un hecho que las sucesiones de bits, así obtenidas, pasan todas las pruebas estadísticas de aleatoriedad. Además es sumamente complicado, si la entrada de las mismas no es débil [1], predecir el siguiente bit en la sucesión.

5.1.3. Curvas elípticas

La teoría de curvas elípticas e hiperelípticas sobre campos finitos, comenzó a tener aplicaciones en criptografía, en 1985; año en el que de forma independiente Victor Miller y Neal Koblitz propusieron los criptosistemas basados sobre curvas elípticas [KOB87, MIL86]. Desde entonces las curvas elípticas también se han utilizado para la factorización de enteros, para pruebas de primalidad y en los últimos años para generar sucesiones pseudoaleatorias.

Gong, Berson y Stinson [GON00] y posteriormente Beelen y Doumen [BEE03] han propuesto formas de obtener sucesiones pseudoaleatorias basándose en curvas elípticas. Beelen y Doumen, proponen generar sucesiones usando curvas sobre un campo finito $\mathbb{F}_{q^e}$ de característica p, donde $\mathbb{F}_{q^e}$ no necesariamente es $\mathbb{F}_p$. En su trabajo [BEE03], dan varios ejemplos de como generar sucesiones usando caracteres tanto aditivos como multiplicativos. Una ventaja de usar la técnica propuesta es que el período de las sucesiones es grande.

Es importante señalar que dos de las razones por las cuales últimamente se están usando cada vez más las curvas elípticas, es porque existen un sinnúmero de grupos que se pueden utilizar. Y la segunda es que se ofrece la misma seguridad que ofrecen otros métodos, pero utilizando parámetros de tamaño más pequeño, lo que hace las implementaciones cada vez más eficientes. Un trabajo interesante al respecto de la eficiencia es el de Baier [BAI05], quien

[1] Si se usa como entrada la fecha y hora de la máquina en la que se esta trabajando, un adversario podría adivinar fácilmente, después de cierto número de intentos cual fue la entrada y por tanto averiguar cual es la sucesión generada.

propone un implementación eficiente de un generador de sucesiones pseudoaleatorias, en el lenguaje de programación Java. Dicho generador está basado en el problema del logaritmo discreto sobre curvas elípticas. La implementación usa curvas elípticas sobre $\mathbb{F}_p$, y la estructura proporcionada por la *Java Cryptography Architecture (JCA)*.

5.2. Otras aplicaciones

A lo largo del presente trabajo se ha podido observar que generar sucesiones pseudoaleatorias para usos criptográficos, no es una tarea fácil puesto que deben cumplir determinadas características, que son muy difíciles de conseguir. Sin embargo, las cualidades que debe cumplir una sucesión dependen mucho de la aplicación que la vaya a utilizar, en ésta sección se describen brevemente otros campos distintos de la criptografía en donde se necesitan sucesiones pseudoaleatorias, así como las propiedades que deberán poseer.

5.2.1. Ampliación de Espectro

Una de las aplicaciones en las que se requiere generar sucesiones es en los sistemas de comunicación basados en la ampliación del espectro o *spread spectrum*. Tales sistemas se han desarrollado desde mediados de los 50's. Al principio se usaba en la milicia para evitar interferencias en las comunicaciones tácticas. Actualmente esta tecnología se aplica en áreas como telefonía celular, posicionamiento global vía satélite ó GPS, por sus siglas en inglés, entre otros.

La idea principal detrás de la ampliación del espectro, es dispersar los datos que se van a transmitir a través de un ancho de banda más grande de lo que convencionalmente es. Para ello se hace uso de una *sucesión pseudoaleatoria*, ya que al hacer la dispersión de los datos, éstos se modulan usando una sucesión pseudoaleatoria en una frecuencia mucho mayor. Para comprender con mayor profundidad este tema consúltese la referencia [PIC82].

Ahora bien múltiples usuarios pueden usar el mismo ancho de banda, si a cada uno se le asigna un *código de dispersión* diferente. Dicho código no es más que una sucesión pseudoaleatoria, mejor conocida como *pseudonoise sequence* o *sucesión pn*, la cual se multiplica por los datos de entrada. A ésta técnica se le denomina *Acceso Múltiple por División de Código* o *CDMA* por sus siglas en inglés. A continuación, se revisará a grandes rasgos, cómo se

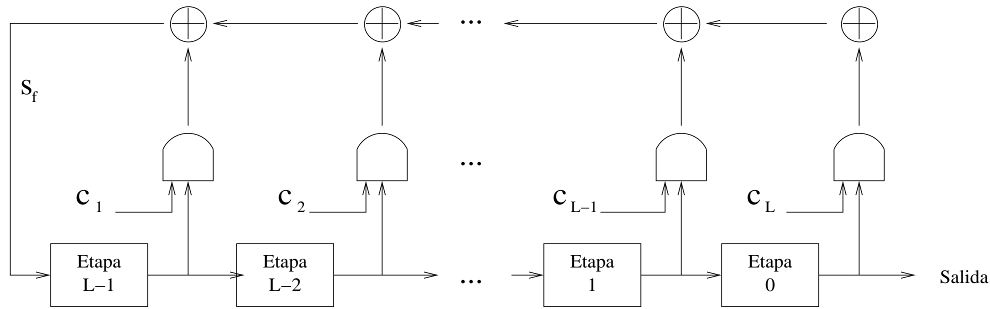

Figura 5.1: LFSR de longitud L

generan éste tipo de sucesiones.

5.2.2. LFSR

Las *sucesiones pn* se generan a través de registros lineales de corrimiento con realimentación, ó *linear feedback shift register (LFSR)*. Los registros de éste tipo se pueden implementar muy fácilmente en hardware, producen sucesiones de periodo largo y con muy buenas propiedades estadísticas. A continuación se mencionará a grandes rasgos cómo funcionan éstos registros y las características de las sucesiones generadas por los mismos. [DIN98, MEN97]

Definición 5.1 *Un LFSR de longitud L consiste de L etapas numeradas* $0, 1, \ldots, L-1$, *y de un reloj que controla el movimiento de los datos. Cada etapa tiene una sola entrada, una sola salida y es capaz de almacenar un bit. Durante cada unidad de tiempo se llevan a cabo las siguientes operaciones:*

1. *el contenido de la etapa 0 sale y se toma como un elemento de la sucesión,*
2. *el contenido de la etapa i pasa a la etapa i-1 para cada* $0 \leq i \leq L-1$, *y*
3. *el nuevo contenido de la etapa* $L-1$ *es el bit de realimentación* s_j, *que se calcula como sigue:*

$$s_j = (c_1 s_{j-1} + c_2 s_{j-2} + \cdots + c_L s_{j-L}) \bmod 2$$

donde cada c_i *es el coeficiente correspondiente del polinomio de conexión* $C(D) = 1 + c_1D + c_2D^2 + \cdots + c_LD^L \in \mathbb{Z}_2[D]$.

La figura 5.1, muestra un diagrama de un LFSR. Éste tipo de registros tiene un *estado inicial*, representado como $[s_{L-1}, \ldots, s_1, s_0]$, en donde cada s_i es el contenido de la etapa i.

Ejemplo

Si se tiene el polinomio de conexión $C(D) = 1 + D + D^3$ y el LFSR correspondiente que se con estado inicial $[0, 1, 0]$, la tabla muestra el contenido de las etapas D_2, D_1, D_0 al final de cada unidad de tiempo t.
La sucesión generada por tal LFSR, es: $0, 1, 0, 0, 1, 1, 1, \ldots$, la cuál es periódica, con periodo 7.

t	D_2	D_1	D_0
0	0	1	0
1	0	0	1
2	1	0	0
3	1	1	0
4	1	1	1
5	0	1	1
6	1	0	1
7	0	1	0

Es bien conocido que cada sucesión generada por un LFSR de longitud L, es periódica si y sólo si su *polinomio de conexión* $C(D)$ es de grado L. Si por el contrario, el grado es menor que L, entonces no todas las sucesiones generadas por el LFSR son periódicas. Si se considera a $C(D) \in \mathbb{Z}_2[D]$ de grado L entonces

- Si $C(D)$ es irreducible sobre $\mathbb{Z}_2$, entonces cada uno de los posibles estados iniciales del LFSR genera una sucesión con periodo igual al del menor entero positivo N para el cual $C(D)$ divide a $1 + D^N \in \mathbb{Z}_2[D]$.
- Si $C(D)$ es un polinomio primitivo[2] entonces cada uno de los $2^L - 1$ estados iniciales distintos de cero del LFSR correspondiente generará una sucesión con periodo máximo $2^L - 1$.

[2] Un polinomio irreducible $f(x) \in \mathbb{Z}_p[x]$ de grado m, es un *polinomio primitivo* si x es un generador de $\mathbb{F}_{p^m}^*$, el grupo multiplicativo de todos los elementos distintos de cero en $\mathbb{F}_{p^m} = \mathbb{Z}_p[x]/ < f(x) >$.

Definición 5.2 *Si $C(D) \in \mathbb{Z}_2[D]$ es un polinomio primitivo de grado L, entonces el LFSR de longitud L correspondiente se denomina LFSR de longitud máxima. A la sucesión generada por un LFSR de longitud máxima con estado inicial distinto de cero, se le denomina* **sucesión m**.

Como se mencionó al principio de la sección, las sucesiones generador por un LFSR, además de tener un periodo largo, también tienen buenas propiedades estadísticas, entre las cuales se encuentran las establecidas por los postulados de Golomb [GOL82], que se listan a continuación.

- El número de ceros y el número de unos en la sucesión debe ser casi igual, es decir, es balanceada.
- En un periodo de la sucesión, al menos la mitad de *runs* debe ser de longitud 1, al menos una cuarta parte de los *runs*[3] debe ser de longitud 2, al menos un octavo de los *runs* debe ser de longitud 3, etc,
- La *función de autocorrelación* $C(t)$ deberá tomar sólamente dos valores, es decir, para algún entero K

$$C(t) = \frac{1}{N}\sum_{i=0}^{N-1}(2s_i - 1)(2s_{i+t} - 1) = \begin{cases} N & \text{si } t = 0 \\ K & \text{si } 1 \leq t \leq N-1 \end{cases}$$

  donde N es el periodo de la sucesión y t, indica un corrimiento de la sucesión s.

La función de autocorrelación mide qué tan similar es una sucesión s y la misma sucesión s con un corrimiento de t posiciones. En las aplicaciones de ampliación del espectro, se desea que la autocorrelación tome valores bajos. Otro parámetro importante es la *correlación cruzada* la cuál se usa para medir qué tan similares son dos sucesiones diferentes.

Las *sucesiones m* son de gran importancia para las redes de amplio espectro. Puesto que la correlación entre dos corrimientos diferentes de una sucesión s es casi cero, éstas se pueden usar cómo códigos distintos. Tal es el caso de las sucesiones de Gold y de Kasami, que son ampliamente usadas en aplicaciones CDMA [DIN98].

[3] Recordar que un *run* es una secuencia de unos o de ceros consecutivos.

Una característica más de las sucesiones generadas por los LFSR es la *complejidad lineal de una sucesión* s_n, donde s_n denota una sucesión finita con n elementos. La complejidad lineal se define como la longitud del LFSR más pequeño cuyos primeros elementos son los de la sucesión s_n. Es posible hallar la complejidad lineal de una sucesión, a través del *algoritmo de Berlekamp-Massey* [MAS69] . Dicho algoritmo también permite encontrar el LFSR que genera a la sucesión. De hecho es por ello que éstas sucesiones no pueden ser usadas para aplicaciones criptográficas.

Capítulo 6

Conclusiones

En presente capítulo se dará un resumen de los temas tratados, los resultados que se obtuvieron al término de la investigación y algunos puntos sobre los que se puede realizar trabajo a futuro.

6.1. Resumen

Se analizaron tres generadores de sucesiones pseudoaleatorias:

- el generador de congruencia lineal (GCL),
- el generador Blum-Blum-Shub y
- el generador Blum-Micali.

Respecto al primero de ellos, se describió cuáles son las características que deben cumplir sus parámetros, para obtener una sucesión de período máximo. También se estudió el algoritmo que permite predecir tales parámetros, cuando se tienen suficientes elementos de la sucesión. Para el generador Blum-Blum-Shub, se dió la construcción del mismo, así como los fundamentos por los que dicho generador es impredecible. Análogamente se realizó el estudio del generador Blum-Micali, con la diferencia, de que para éste último fue necesario explicar el modulo computacional basado en circuitos booleanos. También se describieron a grandes rasgos algunos otros mecanismos para generar sucesiones, entre los que se debe resaltar el uso de curvas elípticas.

Finalmente, dada la importancia de generar sucesiones para las nuevas tecnologías inalámbricas, gracias a las cuales las personas se pueden comunicar actualmente, casi sin importar el lugar en el que se encuentran; se describen brevemente los LFSR, uno de los mecanismos más usados por su sencillez y por las cualidades de las sucesiones que éstos generan.

6.2. Conclusiones obtenidas

Después de haber analizado tres de los generadores clásicos de sucesiones pseudoaleatorias, se reitera que su estudio no es una cuestión que se pueda dejar a la ligera, cuando de aplicaciones criptográficas se trata. Pues además de la seguridad que las sucesiones generadas puedan ofrecer, también se de considerar la eficiencia de los algoritmos generadores.

Así tenemos que el generador de congruencia lineal (GCL), es sumamente eficiente pero no funciona para aplicaciones criptográficas, dado que sus parámetros son totalmente predecibles. Por otro lado, los generadores Blum-Micali y Blum-Blum-Shub garantizan el nivel de seguridad de las sucesiones resultantes, sin embargo no son usados cotidianamente debido al tiempo que consume generar una sucesión de este estilo; lo cual no es adecuado en aplicaciones de tiempo real. De hecho, de éstos dos algoritmos, quizá el menos recomendado es el generador Blum-Micali. debido a que dado que el exponente involucrado es muy grande (1024 bits por ejemplo), la obtención de un bit es más costosa que en el caso del Blum-Blum-Shub, donde el exponente es 2.

Una forma de ver ésto, es considerar el número de operaciones binarias o más formalmente operaciones de *precisión sencilla (single-precision)*, al realizar la exponenciación modular en cada uno de ellos. Para ello se denotará a $(x_{t-1} \cdots x_1 x_0)_2$ y $(m_{l-1} \cdots m_1 m_0)_2$ a la expansión binaria de x y m respectivamente. En el caso del generador Blum-Blum-Shub, es necesario calcular $x^2 \bmod n$; si se lleva a cabo la exponenciación utilizando métodos clásicos, es decir, primero se hace x^2 y posteriormente se divide por m entonces se requiere de $t(t+1)/2$ multiplicaciones de precisión sencilla y de l divisiones de precisión sencilla. El generador Blum-Micali requiere calcular $g^x \bmod m$, para obtener un sólo bit de la sucesión. Con la notación anterior y utilizando ahora el *algoritmo de exponenciación modular de Montgomery* [MON85, MEN97], se requiere de $3l(l+1)(t+1)$ multiplicaciones de precisión sencilla.

Es importante resaltar que aunque los algoritmos Blum-Blum-Shub y Blum-Micali no son muy usados, sí tienen una relevancia teórica, ya que representan la base de muchos otros generadores de sucesiones para aplicaciones criptográficas.

También es claro que es sumamente importante considerar el uso que va a tener una sucesión pseudoaleatoria. Puesto que en base a ello, se podrá elegir de entre los métodos que se encuentren disponibles, alguno que tenga las propiedades necesarias para determinada aplicación. Por ejemplo las sucesiones utilizadas en las comunicaciones inalámbricas, deben tener un periodo largo sin importar si el generador es predecible, pero en criptografí a, ambas cosas deben tomarse en cuenta.

Para investigaciones futuras queda el construir implementaciones eficientes que permitan reducir los tiempos de ejecución de los algoritmos generadores. Igualmente importante es buscar otras formas de generar sucesiones tanto para aplicaciones criptográficas como para las que no lo son.

Apéndice A

Resultados Clásicos

En el presente apéndice se incluyen algunos resultados clásicos de Teoría de Números, necesarios en el desarrollo del presente trabajo. Su demostración no se incluye, pero es posible consultarla por ejemplo en [HAR83, NIV85, ROS83].

Teorema A.1 *Si* a, b, r *y* s *son enteros tales que* $(r, s) = 1$ *entonces*

$$a \equiv b \text{ mód } rs \textit{ si y sólo si } \begin{cases} a \equiv b \text{ mód } r \quad y \\ a \equiv b \text{ mód } s \end{cases}$$

Teorema A.2 *Si* $a \equiv b \text{ mód } m_1, a \equiv b \text{ mód } m_2, \ldots, a \equiv b \text{ mód } m_k$ *donde* a, b, $m_1, m_2, \ldots m_k$ *son enteros y además* $m_1, m_2, \ldots, m_k$ *son positivos, entonces*

$$a \equiv b \text{ mód } [m_1, m_2, \ldots, m_k],$$

donde $[m_1, m_2, \ldots, m_k]$ *es el mínimo común múltiplo de* $m_1, m_2 \ldots, m_k$.

Corolario A.1 *Si* $a \equiv b \text{ mód } m_1, a \equiv b \text{ mód } m_2, \ldots, a \equiv b \text{ mód } m_k$ *donde* a, b, *son enteros y* $m_1, m_2, \ldots, m_k$ *son enteros positivos y primos relativos por pares, entonces*

$$a \equiv b \text{ mód } m_1 m_2 \cdots m_k,$$

Teorema A.3 *(Teorema de Fermat) Sea* p *primo y* $a \in \mathbb{Z}$ *tal que* $mcd(a, p) = 1$ *entonces*

$$a^{p-1} \equiv 1 \text{ mód } p$$

Definición A.1 *Si n es un entero positivo, entonces la función de Euler, $\varphi(n)$, se define como el número de enteros positivos menores que n que son primos relativos a n.*

Teorema A.4 *(Teorema de Euler) Si m y a son enteros positivos con $m > 0$ y $(a, m) = 1$, entonces $a^{\varphi(m)} \equiv 1 \text{ mód } m$.*

Teorema A.5 *Si p es primo, entonces $\varphi(p) = p - 1$. Recíprocamente, si p es un entero positivo tal que $\varphi(p) = p - 1$, entonces p es primo.*

Teorema A.6 *Si $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ es la factorización en primos de n, entonces $\varphi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_k}\right)$.*

Definición A.2 *Un exponente universal de un entero positivo n es un entero positivo U tal que $a^U \equiv 1 \text{ mód } n$ para cada $a \in \mathbb{Z}$, tal que $(a, n) = 1$.*

Definición A.3 *El exponente universal más pequeño de un entero positivo n se denomina el mínimo exponente universal de n, y se denota por $\lambda(n)$ y esta dado de la siguiente forma:*

$$\begin{array}{l} \lambda(2) = 1,\ \lambda(4) = 2,\ \lambda(2^e) = 2^{e-2},\ e \geq 3 \\ \lambda(p^e) = p^{e-1}(p-1),\ si\ p > 2 \\ \lambda(p_1^{e_1} \ldots p_t^{e_t}) = [\lambda(p_1^{e_1}), \ldots \lambda(p_t^{e_t})] \end{array}$$

λ también es conocida como la función lambda de Carmichael.

Corolario A.2 *(Extensión de Carmichael al teorema de Euler) Si m es un entero positivo y a es un entero tal que $(a, m) = 1$ entonces*

$$a^{\lambda(m)} \equiv 1 \text{ mód } m$$

Teorema A.7 *Sean a, b y m enteros tales que $m > 0$ y $(a, m) = d$. Si $d \nmid b$, entonces $ax \equiv b \text{ mód } m$ no tiene soluciones. Si $d | b$, entonces $ax \equiv b \text{ mód } m$ tiene exactamente d soluciones no congruentes módulo m, dadas por,*

$$x = x_0 + (m/d)t \ \textit{donde}\ t = 0, 1, \ldots, d - 1$$

Teorema A.8 *(Teorema chino del residuo) Sean $m_1, m_2, \ldots, m_r$ enteros positivos tales que $(m_i, m_j) = 1$ para $i \neq j$ con $1 \leq i, j \leq r$. Entonces el sistema de congruencias*

$$\begin{aligned} x &\equiv a_1 \text{ mód } m_1, \\ x &\equiv a_2 \text{ mód } m_2, \\ &\vdots \\ x &\equiv a_r \text{ mód } m_r \end{aligned}$$

tiene una solución única módulo $M = m_1 m_2 \cdots m_r$.

Definición A.4 *Sea* n *un número entero positivo, se define*

$$Q_n = \{a \in \mathbb{Z}_n^* \mid x^2 \equiv a \text{ mód } n \textit{ tiene solución}\}$$

como el conjunto de residuos cuadráticos módulo n.

Lema A.1 *Si* p *es un número primo impar y* $a \in \mathbb{Z}_p^*$, *entonces la congruencia*

$$x^2 \equiv a \text{ mód } p$$

no tiene solución o tiene exactamente 2 soluciones módulo p, *no congruentes entre sí.*

Definición A.5 *Sea* p *un número primo impar y* a *un entero no divisible por* p. *El símbolo de* Legendre $\left(\frac{a}{p}\right)$ *esta definido de la siguiente manera*

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \textit{si } a \in Q_p \\ -1 & \textit{en otro caso} \end{cases}$$

Definición A.6 *Sea* n *un entero positivo impar cuya factorización es* $n = p_1^{e_1} p_2^{e_2} \cdots p_m^{e_m}$ *y* a *un entero tal que* $(a, n) = 1$. *Entonces el símbolo de* Jacobi $\left(\frac{a}{n}\right)$ *está dado por*

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1^{e_1} p_2^{e_2} \cdots p_m^{e_m}}\right) = \left(\frac{a}{p_1}\right)^{e_1} \left(\frac{a}{p_2}\right)^{e_2} \cdots \left(\frac{a}{p_m}\right)^{e_m}$$

donde $\left(\dfrac{a}{p_i}\right)$ *con* $1 \leq i \leq m$ *es el símbolo de* Legendre.

Teorema A.9 *Sea* p *un número primo impar,* a *y* b *enteros positivos tales que* $(a, p) = 1$ *y* $(b, p) = 1$, *entonces*

1. $\left(\dfrac{a}{p}\right) \equiv a^{(p-1)/2} \text{ mód } p$

2. *Si* $a \equiv b \bmod p$ *entonces* $\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$

3. $\left(\frac{a}{p}\right)\left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right)$

4. $\left(\frac{a^2}{p}\right) = 1$

5. $\left(\frac{-1}{p}\right) = \begin{cases} 1 & si\ p \equiv 1 \bmod 4 \\ -1 & si\ p \equiv -1 \bmod 4 \end{cases}$

6. $\left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}$, *es decir* $2 \in Q_p$ *para los primos* p *tales* $p \equiv \pm 1 \bmod 8$ *y no es un residuo cuadrático para todos los primos* p *tales que* $p \equiv \pm 3 \bmod 8$.

Teorema A.10 *Sea* n *un entero positivo impar y sean* a *y* b *enteros primos relativos con* n.

1. *Si* $a \equiv b \bmod n$ *entonces* $\left(\frac{a}{n}\right) = \left(\frac{b}{n}\right)$,

2. $\left(\frac{a}{n}\right)\left(\frac{b}{n}\right) = \left(\frac{ab}{n}\right)$,

3. $\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2}$,

4. $\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8}$

Proposición A.1 *Si* $a \equiv 3 \bmod 4$ *entonces* $(a^{2^{e-1}} - 1)/(a-1) \equiv 0 \bmod 2^e$ *para cualquier entero* $e > 1$.

Demostración.
Utilizando inducción en e, si $e = 2$ entonces

$$\frac{a^{2^{e-1}} - 1}{a-1} = \frac{a^2 - 1}{a-1} = a + 1$$

y puesto que $a \equiv 3 \bmod 4$, i.e., $a = 3 + 4t$ para algun $t \in \mathbb{Z}$ entonces $a + 1 = 3 + 4t + 1 = 2^2(1 + t)$. Por lo tanto

$$\frac{a^2 - 1}{a - 1} \equiv 0 \text{ mód } 2^2$$

Supóngase que la afirmación es válida para $e = k$, se demostrará para $e = k + 1$. Así

$$\frac{a^{2^{k+1-1}} - 1}{a - 1} = \frac{a^{2^k} - 1}{a - 1} = a^{2^{k-1}} + 1 \cdot \frac{a^{2^{k-1}-1}}{a - 1}$$

pero dado que la afirmación es cierta para $e = k$, entonces

$$\frac{a^{2^{k-1}-1}}{a - 1} = 2^k s \text{ para alguna } s \in \mathbb{Z}$$

y como a es impar, $a^{2^{k-1}+1} = 2q, q \in \mathbb{Z}$ por consiguiente

$$a^{2^{k-1}} + 1 \cdot \left(\frac{a^{2^{k-1}-1}}{a - 1} \right) = 2q \cdot 2^k s = 2^{k+1} qt$$

de donde se concluye que

$$\frac{a^{2^{k+1-1}} - 1}{a - 1} \equiv 0 \text{ mód } 2^{k+1}$$

□

Proposición A.2 *Si p es un número primo tal que $p \equiv 3 \bmod 4$ y $a \in Q_p$, entonces las soluciones a la congruencia $x^2 \equiv a \bmod p$ están dadas por: $\pm a^{(p+1)/4} \bmod p$. Además dichas soluciones son a su vez residuos cuadráticos módulo n.*

Demostración.
Es claro que $\left(\frac{a}{p}\right) = 1$, y por la propiedad 1 del teorema A.9, se tiene que $a^{(p-1)/2} \equiv 1 \bmod p$. Por consiguiente,

$$\pm(a^{(p+1)/4})^2 = a^{(p+1)/2} = a^{(p-1)/2} a \equiv a \text{ mód } p.$$

Para probar que $a^{(p+1)/4}$ es también un residuo cuadrático, basta con ver que su símbolo de Legendre es igual con 1. Puesto que $a^{(p-1)/2} \equiv 1 \bmod p$, entonces:

$$(a^{(p+1)/4})^{(p-1)/2} = (a^{(p-1)/2})^{(p+1)/4} \equiv 1 \text{ mód } p.$$

□

Apéndice B

Código fuente

En este apéndice aparece el código de los programas utilizados a lo largo de la elaboración de esta tesis, entre ellos se encuentra el código para implementar los tres generadores estudiados en el presente trabajo, es decir, el generador de congruencia lineal (GCL), generador Blum-Blum-Shub y el generador Blum-Micali. Para facilitar la labor se tiene una sección para cada código.

B.1. Generador de congruencia lineal

Primero se incluye el código del generador y posteriormente el código utilizado para predecir los valores de los parámetros. El primer programa simplemente se ocupa de generar una sucesión sin importar si se tiene el maximo periodo; el segundo programa, cuida que los parámetros estén bien escogidos. Y el tercer programa sirvió de apoyo en la predicción de a, c, m.

```
/**
  *  Archivo: GCLBase.java
  *  Descripcion: Este programa implementa un generador de congruencia lineal
  *               sin importar si se consigue o no el periodo maximo.
  */

import java.io.*;
import java.math.*;
import java.util.Random;

public class GCLBase{
        private BigInteger m, a, c, x0; //parametros del generador
```

```
private byte[] sucesion;

public GCLBase(){
     this(512);
}

public GCLBase(int numbits){
      a = genera_numero(numbits-1); //se generan al azar a, c, x0 y m
      c = genera_numero(numbits-1);
      x0 = genera_numero(numbits-1);
      m = genera_numero(numbits);
      a=reduccion_modular(a);
      c=reduccion_modular(c);
      x0=reduccion_modular(x0);
}

public GCLBase(BigInteger m, BigInteger a, BigInteger c, BigInteger x0){
      this.m =(m.compareTo(BigInteger.ZERO)==1)? m:m.abs(); //m > 0
      this.a = reduccion_modular(a);
      this.c = reduccion_modular(c);
      this.x0 = reduccion_modular(x0);
}

public BigInteger genera_numero(int numbits){
        Random bitsazar;
        BigInteger numero;

        bitsazar=new Random();
        numero = new BigInteger(numbits, bitsazar);
        numero = numero.abs();
        bitsazar = null;

        return numero;
}

public BigInteger reduccion_modular(BigInteger x){
        if(x.compareTo(m)==1)
           x = x.remainder(m);
        return x;
}

public void genera(){
        BigInteger x = x0; // x inicial
        BigInteger suc = BigInteger.ZERO;
```

```
        BigInteger numceros = BigInteger.ZERO;
        BigInteger i = BigInteger.ZERO;   //i =0
        do{
                x = a.multiply(x).add(c).mod(m); // (ax + c) mod m
                if (x.testBit(0)) // tomo el primer bit
                     suc = suc.setBit(i.intValue());
                i=i.add(BigInteger.ONE); // i++
                if(x.compareTo(BigInteger.ZERO)==0)
                     numceros.add(BigInteger.ONE);
                if(numceros.compareTo(m)==1)
                     System.exit(-1); //termina si la sucesion degenera a 0
        }while(x.compareTo(x0)!=0); // toda la sucesion

        sucesion = suc.toByteArray(); // se guarda la sucesion en el
}

public void escribe(){
        BigInteger suc = new BigInteger(sucesion);
        System.out.println("sucesion = "+suc.toString(2));
}

public BigInteger mGet(){
       return this.m;
}

public void mSet(BigInteger val){
       this.m = val;
}

public BigInteger aGet(){
       return this.a;
}

public void aSet(BigInteger val){
      this.a = val;
}

public BigInteger cGet(){
      return this.c;
}

public void cSet(BigInteger val){
      this.c = val;
}
```

```
        public BigInteger x0Get(){
              return this.x0;
        }

        public void x0Set(BigInteger val){
              this.x0 = val;
        }



/**
  * Archivo: GCLOptimo.java
  *  Descripcion: Utilizando GCLBase, las propiedades del GCL, se aumentaran
  *             permitiendo generar una sucesion con periodo maximo.
  */



import java.math.*;
import java.util.Random;

public class GCLOptimo extends GCLBase{
        private BigInteger p, q; // primos p y q para construir el modulo
        private int bitsp; // longitud en bits de los parametros

        public GCLOptimo(int bits){
                bitsp=bits;
                genera_m();
                genera_c();
                genera_a();
                genera_x0();
        }

        public void imprime_parametros(){
                System.out.println("a: "+super.aGet());
                System.out.println("c: "+super.cGet());
                System.out.println("x0: "+super.x0Get());
                System.out.println("m: "+super.mGet());
        }

        public void genera_m(){
                Random bitsazar;

                bitsazar=new Random();
```

```
                p = BigInteger.probablePrime(bitsp, bitsazar);
                q = BigInteger.probablePrime(bitsp, bitsazar);
                System.out.println(p);
                System.out.println(q);
                BigInteger mod = p.multiply(p).multiply(q); //m = p*p*q;
                super.mSet(mod);
        }

        public void genera_c(){
                BigInteger c;
                do{
                     c = super.genera_numero(bitsp); // 0 <=c< m
                     c = super.reduccion_modular(c); // c mod m;
                }while(c.gcd(super.mGet()).compareTo(BigInteger.ONE) != 0);//mcd(c,m)=1

                super.cSet(c);
         }

         public void genera_a(){
               BigInteger a;

               a = p.multiply(q).add(BigInteger.ONE);
               a=super.reduccion_modular(a);
               super.aSet(a);
         }

        public void genera_x0(){
              BigInteger x0;

              x0=super.genera_numero(bitsp);
              x0=super.reduccion_modular(x0);
              super.x0Set(x0);
        }
}



/*-----------------------------------------------------------------------
 * Archivo: crakgcl.c
 * Autora: Sandra Diaz Santiago.
 * Descripcion: Este programa fue utilizado en la prediccion de los
 * parametros del Generador de Congruencia Lineal (GCL).
 * -----------------------------------------------------------------------*/
#include <stdio.h>
```

```
void convirtiendo(FILE *suc_gcl, int *sucesion),
      sucesionxprima(int *sucesion, int *xprima);

int mcd(int x, int y), mcd_devarios(int *numeros);

main()
{
  FILE *suc_gcl; /*apuntador al archivo que contiene la sucesion del gcl*/
  int sucesion[100], i, d; /*arreglo que contiene la sucesion*/
  int xprima[100]; /*sucesion de x^\prime*/

  suc_gcl=fopen("suce_gcl.txt", "r");
  if(suc_gcl!=NULL){
    convirtiendo(suc_gcl, sucesion);
    sucesionxprima(sucesion, xprima);
    d=mcd_devarios(xprima);
    fclose(suc_gcl);
    printf("\n mcd : %d", d);
  }
}

void convirtiendo(FILE *suc_gcl, int *sucesion)
{
  int elemento; /*elemento de la sucesion*/
  int i; /*indice para el arreglo sucesion*/

  i=0;
  while(fscanf(suc_gcl,"%d", &elemento)!=EOF) sucesion[i++]=elemento;
  sucesion[i]=-1;
}

void sucesionxprima(int *sucesion, int *xprima)
{

  int j, /*indice de xprima*/
      i; /*indice de sucesion*/

  j=0;
  for(i=1; sucesion[i]!=-1; i++){
    xprima[j++]=sucesion[i]-sucesion[i-1];
    printf("%d ", xprima[j-1]);
  }
  xprima[j]='\0';
}
```

```
int mcd(int x, int y)
{
  int d; /*maximo comun divisor de x y y*/

  if(x<0) x=-1*x;
  if(y<0) y=-1*y;

  d=y;
  while(x>0){
    d=x;
    x=y%x;
    y=d;
  }
  return d;
}

int mcd_devarios(int *numeros)
{
  int i, /*indice del arreglo numeros*/
    d; /*mcd de los numeros*/

  d=numeros[0];
  for(i=1; numeros[i]!='\0'; i++){
    d=mcd(d, numeros[i]);
  }

  return d;
}
```

B.2. Generador Blum-Blum-Shub

En este caso, solamente se incluye el código para implementar al generador.

```
/**
  * Archivo: bbs.java
  * Autora : Sandra Diaz Santiago
  * Descripcion : Esta clase implementa al generador Blum-Blum-Shub
  *               basado en el problema de la residuosidad cuadratica.
  *
  */

import java.math.*;
import java.util.Random;
```

```
public class bbs{
    private int tam_primo;          //numero de bits para p y q
    private BigInteger p, q, n, x_0, x_i;
    private UnoaCuatro bi;          //1,2,3,4 usando BigInteger

    public bbs(){
       this(512);    //tamanio por default 512 bits
    }

    public bbs(int tam){
        bi = new UnoaCuatro();
        inicializa(tam);
    }

    public void inicializa(int tam){
        tam_primo = tam;
        p = obten_primo(tam_primo); //  p y q congruentes con 3 modulo 4.
        q = obten_primo(tam_primo);
        n = p.multiply(q);
        x_0 = residuo_cuadratico(); // residuo cuadratico mod n
    }

     public void genera(){
         int elemento;

         x_i = x_0;
         siguiente_xi();
         while( x_i.compareTo(x_0)!=0 ){
            siguiente_xi();
            elemento = siguiente();
            System.out.print(elemento+", ");
         }
    }

    public void datos_generador(){
        System.out.println("Generador Blum-Blum-Shub con parametros:");
        System.out.print("p: "+ p);
        System.out.print("  q: "+ q);
        System.out.print("  n: "+ n);
        System.out.println("  x_0: "+ x_0);
    }

    public int siguiente(){   //genera el siguiente bit de la sucesion
        return x_i.mod(bi.num(2)).intValue();
```

```
}

public void siguiente_xi(){  //genera el siguiente x_i
    x_i = x_i.modPow(bi.num(2), n);
}

public BigInteger obten_primo(int tam){
    Random bitsAleatorios;
    BigInteger primo;

    do{
        bitsAleatorios = new Random();
        primo =  BigInteger.probablePrime(tam, bitsAleatorios);
    }while( !congruente3mod4(primo) );

    return primo;
}

public boolean congruente3mod4(BigInteger primo){

    if( primo.remainder(bi.num(4)).compareTo(bi.num(3)) != 0 ){
        return false;
    }
    return true;
}

public BigInteger residuo_cuadratico(){
    Random bitsAleatorios;
    BigInteger probableResCuad;
    int simbp, simbq;

    bitsAleatorios = new Random();

    do{
        probableResCuad = new BigInteger(tam_primo, bitsAleatorios);
        simbp = this.simbLegendre(probableResCuad, p);
        simbq = this.simbLegendre(probableResCuad, q);

    }while(simbp != 1 || simbq !=1);

    return probableResCuad;
  }

public int simbLegendre(BigInteger a, BigInteger p){
    BigInteger exponente;
```

```
        exponente = p.subtract(bi.num(1)).divide(bi.num(2)); // (p-1)/2
        return a.modPow(exponente, p).intValue();  //a^[(p-1)/2] mod p
    }
}
```

B.3. Generador Blum-Micali

El primer programa, se utilizó para generar los conjuntos D_i, que se mencionan en la sección de predicados no aproximables. El segundo programa es una implementacion del generador Blum-Micali.

```
/*---------------------------------------------------------------------------
  Archivo: genera_di.c
  Autora: Sandra Diaz S
  Descripcion: Este programa calcula el conjunto D_i para el ejemplo del
               apartado 2.2 del articulo de M. Blum & S. Micali acerca de
       los CSPRBG, dado un i en los enteros.
 ----------------------------------------------------------------------------*/
#include <stdio.h>

void calculaDi(int i),
     pot2(int a, int *e, int *a1);
int mcd(int x, int y),
  jacobi(int a, int n);

main()
{
  int i, d;
  printf("\ni:"); /*a y b deben ser dos enteros distintos de cero*/
  scanf("%d", &i);
  calculaDi(i);
  return 0;

}

/*esta funcion averigua cuales de los enteros entre 1 e i son
  primos relativos con i*/
void calculaDi(int i)
{
  int j;
  FILE *apdi;
  apdi=fopen{"dis.txt", "a+");
```

```
  if(apdi!=NULL){
    fprintf(apdi,"D_%d={", i);
    for(j=1; j<i; j++)
       if(mcd(j,i) && jacobi(j,i)==1) fprintf(apdi,"%d,", j);
    fprintf(apdi,"}\n");
  }
}

int mcd(int x, int y)
{
  int d; /*maximo comun divisor de x y y*/

  if(x<0) x=-1*x;
  if(y<0) y=-1*y;

  d=y;
  while(x>0){
    d=x;
    x=y%x;
    y=d;
  }
  return d;
}

int jacobi(int a, int n)
{
  int e, a1, s, mod8, mod4, n1;
  if(a==0) return 0;
  if(a==1) return 1;
  pot2(a, &e, &a1);
  if(e%2==0) s=1;
  else{
    mod8=n%8;
    if(mod8==1 || mod8==7) s=1;
    else
      if(mod8==3 || mod8==5) s=-1;
  }
  mod4=n%4;
  if(mod4==3 && a1%4==3) s=-s;
  if(a1>1) n1=n%a1;
  else{
    n1=a1;
    a1=n;
  }
  return(s*jacobi(n1,a1));
```

```
}

/*Funcion que descompone el entero a, como sigue a=2^e*a1, donde
a1 es claramente impar*/
void pot2(int a, int *e, int *a1)
{
  int cociente, j;

  j=0;
  cociente=a;
  while (cociente%2==0)
    {
      j++;
      cociente=cociente/2;
    }
  *e=j;
  if(j>0)
    *a1=cociente;
  else *a1=a;
}




/**
  *Archivo: GBlumMicali.java
  * Autora : Sandra Diaz Santiago
  * Descripcion : Esta clase implementa al generador Blum-Micali
  *               basado en el problema de la residuosidad cuadratica.
  *
  */

import java.math.*;
import java.util.Random;

public class GBlumMicali{
    private int numbits;         //numero de bits para p y q
    private BigInteger p, q, g, x_0, x_i;
    private UnoaCuatro bi;         //1,2,3,4 usando BigInteger

    public GBlumMicali(){
        this(512);    //tamanio por default 512 bits
    }

    public GBlumMicali(int tam){
```

```
        numbits = tam;
        bi = new UnoaCuatro();
        inicializa();
    }

    public void inicializa(){
        Random azar = new Random();
        obten_primo(); // se busca un primo p=2q+1, donde q es primo
        obten_g();
        x_0 = (new BigInteger(numbits-1,azar)).mod(p);  //se escoge 0 < x_0 < p-1
    }

     public void genera(){

         int elemento;

         BigInteger i = BigInteger.ZERO;
         x_i = x_0;
         do{
            siguiente_xi();
            elemento = siguiente();
            System.out.print(elemento);
            i=i.add(BigInteger.ONE); // i++
         }while(i.compareTo(p)==-1 && x_i.compareTo(x_0)!=0); // i<p && x_i!=x_0

    }

    public void datos_generador(){
        System.out.println("Generador Blum-Micalicon parametros:");
        System.out.print("p: "+ p);
        System.out.print("  g: "+ q);
        System.out.println("  x_0: "+ x_0);
    }

    public int siguiente(){
        int compara;

        compara = x_i.compareTo(q); //q = p-1/2
        if(compara==-1 || compara ==0){ //genera el siguiente bit de la sucesion
            return 1;
         }
        return 0;
    }

    public void siguiente_xi(){  //genera el siguiente x_i
```

```
        x_i = g.modPow(x_i, p);
    }

    public void obten_primo(){
        Random bitsAleatorios;

        do{
            bitsAleatorios = new Random();
            q =  BigInteger.probablePrime(numbits-1, bitsAleatorios);
            p = q.multiply(bi.num(2)).add(bi.num(1)); //2q+1
       }while(!p.isProbablePrime(80));
    }

    public void obten_g(){
         Random bitsazar;
         BigInteger b1, b2;

         bitsazar = new Random();
         do{
              g=new BigInteger(numbits, bitsazar);
              g=g.abs().mod(p);
              b1=g.modPow(bi.num(2),p);
              b2=g.modPow(q, p);
         }while(b1.compareTo(bi.num(1))==0 || b2.compareTo(bi.num(1))==0);
    }
```

Bibliografía

[ADL79] L. M Adleman. *A subexponential algorithm for the discrete logarithm problem with applications to cryptography*. Proceedings of the IEEE 20th Annual Symposium on Foundations of Computer Science, 55-60, 1979.

[COR90] T. Cormen, C. E. Leiserson, R. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[BEL97] M. Bellare, S. Goldwasser, D. Micciancio. *"Pseudo-Random" number generation within cryptographic algorithms: the DSS Case*. Advances in Cryptology, Proceedings of Crypto '97 (LNCS 1294), 277-291, 1997.

[BLU84] M. Blum, S. Micali. *How to generate cryptographically strong sequences of pseudo-random bits*. SIAM Journal on Computing,**13**,4, 850-864,1984.

[BLU86] L. Blum, M. Blum, M. Shub. *A simple unpredictable pseudo-random number generator*. SIAM Journal on Computing, **15**,2,364-383,1986.

[CHA66] G.J. Chaitin. *On the length of programs for computing finite binary sequences*. Journal of the ACM, **13**, 547-570, 1966.

[COP86] D. Coppersmith, A. M. Odlyzko, R. Schroeppel. *Discrete logarithms in GF(p)*. Algorithmica, **1**,1-15,1986

[DAV94] D. Davis, R. Ihaka, P. Fenstermacher. *Cryptographic randomness from air turbulence in disk drives*. Advances in Cryptology - Crypto '94 (LNCS 839),114-120, 1994.

[DIF76] W. Diffie, M. E. Hellman. *New directions in cryptography*. IEEE Transactions on Information Theory, **22**,6, 644-654, 1976.

[GOL99] O. Goldreich, *Modern cryptography, probabilistic proofs and pseudorandomness.* Springer Verlag, 1999.

[GOL01] O. Goldreich, *Foundations of Cryptography. Basic tools.* Cambridge University Press, 2001.

[HAR83] G. H. Hardy y E. M. Wright. *An introduction to the theory of numbers.* Oxford University Press, 1983.

[HEI93] R. Heiman. *A note on discrete logarithm with special structure*. Advances in Cryptology-EUROCRYPT 92, (LNCS 658), 454-457, 1993

[HEL83] M.E. Hellman, J.M. Reyneri. *Fast computation of discrete logarithms in GF(q)*. Advances in Cryptology-Proceedings of Crypto '82,3-13, 1983.

[HER86] I. N. Hernstein. *Algebra Abstracta.* Grupo Editorial Iberoamericana, 1986.

[KAH67] D. Kahn.*The Codebreakers. The Comprehensive History of Secret Communications from Ancient times to the Internet*. Scribner, 1967.

[KAU95] C. Kaufman, R. Perlman, M. Speciner. *Network Security. Private Communication in a Public World.* Prentice Hall, 1995.

[KNU81] D. E. Knuth. *The Art of Computer Programming:Volume 2, Seminumerical Algorithms.* 2a. ed., Addison-Wesley, 1981.

[KOL65] A. Kolmogorov. *Three approaches to the concept of "The amount of Information"*. Probl of Inform. Transm.,**1/1**, 1965.

[KRA86] E. Kranakis. *Primality and Cryptography.* John Wiley & Sons, 1986.

[KRA90] H. Krawczyk. *How to predict congruential generators*. Proceedings of CRYPTO '89, (LNCS 435),138-153, 1990.

[LAG90] J. C. Lagarias. *Pseudorandom Number Generators in Cryptography and Number Theory.* Proceedings of Symposia in Applied Mathematics.**42**, 1990.

[MAU92] Maurer, U. *A Universal Statistical Test for Random Bit Generators*, Journal of Cryptology,**5**,2,89-105, 1992.

[MEN97] A. Menezes; S. Vanstone ; P. van Oorschot. *Handbook of Applied Cryptography*. CRC Press, 1997.

[MON85] P. Montgomery. *Modular multiplication without trial division*. Mathematics of Computation. **44**, 97-108, 1985.

[MOT95] R. Motwani, P. Raghavan. *Randomized Algorithms* Cambridge University Press, 1995.

[NEU94] B.C. Neuman, T.Ts'o. *Kerberos: An Authentication Service for Computer Networks*. IEEE Communications, **32(9)**, 33-38, 1994.

[NIV85] I. Niven, H. Zuckerman. *Introducción a la teoría de los números.* Editorial Limusa, 1985.

[POH78] S.C. Pohlig, M.E. Hellman. *An improved algorithm for computing logarithms over GF(p) and its cryptographics significance*. IEEE Transactions of Information Theory, **24**, 106-110, 1978.

[POL75] J.M. Pollard. *A Monte Carlo method for factorization*. BIT,15,331-334, 1975.

[PLU82] J. Plumstead. *Inferring a sequence generated by a linear congruence*. Proceedings of the IEEE 23rd Annual Symposium on Foundations of Computer Science, 153-159, 1982.

[RFC95] RFC 1750. *Randomness requirements for security*. Internet Request for Comments 1750. D. Eastlake, S. Crocker and J. Schiller. Diciembre 1994.

[RFC99] RFC 2246 *The TLS Protocol*. Internet Request for Comments 2246. T. Dierks, C. Allen. Enero 1999.

[RIV78] R.L. Rivest, A. Shamir, L. M. Adleman. *A method for obtaining digital signatures and public-key cryptosystems*. Communications of the ACM, 21, 120-126, 1978.

[ROS83] K. H. Rosen. *Elementary number theory and its applications.* Addison-Wesley, 1983.

[SHA48] C.E. Shannon. *A mathematical theory of communications*. Bell Systems Technical Journal,**27**,379-423 y 623-656, 1948.

[SCH94] B. Schneier. *Applied Cryptography. Protocols, algorithms and source code in C.* John Wiley & Sons, 1994.

[SIN99] S. Singh. *The Code Book. The science of secrecy from ancient Egypt to quantum cryptography*. Anchor Books, 1999.

[SOL64] R.J. Solomonov. *A formal theory of inductive inference*. Inform. and Control,**7/1**, 1-22, 1964.

[STA03] W. Stallings. *Cryptography and Network Security. Principles and Practices.* 3a ed. Prentice Hall. 2003.

[STE88] J.G. Steiner, C. Neuman, J.I. Schiller. *Kerberos: an authentication service for open network systems*. Proceedings of the Winter 1988 Usenix Conference, 191-201, 1988.

[STI95] D.R. Stinson, *Cryptography: Theory and practice*, CRC Press, 1995.

Sucesiones pseudoaleatorias y curvas elípticas

[BAI05] H. Baier. *A fast Java Implementation of a Provably Secure Pseudorandom Bit Generator*. Preprint. 2005.

[BEE03] P.H.T. Beelen, J.M. Doumen. *Pseudorandom sequences from elliptic curves*. Proceedings of Sixth Conference on Finite Fields. Springer Verlag. 37-52,2003.

[GON00] G. Gong, T.A. Berson, D.R. Stinson. *Elliptic curve pseudorandom sequence generators*. Selected areas in cryptography. Springer Verlag, 34-48, 2000.

[KOB87] N. Koblitz. *Elliptic curve cryptosystems.* Math Comp. 48, 203-209, 1987.

[MIL86] V. Miller. *Uses of elliptic curve in cryptography.* Advances in Cryptology-Crypto '85 (LNCS 218), Springer Verlag, 417-426,1986.

Sucesiones y aplicaciones

[BRO96] D.S. Broomhead, J.P. Huke, M.R. Muldoon. *Codes for Spread Spectrum Applications Generated Using Chaotic Dynamical Systems*. 1996

[DIN98] E.H. Dinan, B. Jabbari. *Spreading codes for direct sequence CDMA and wideband CDMA cellular networks*. IEEE Communications Magazine, 48-54, 1998.

[GOL82] S.W.Golomb. *Shift Register Sequences*. Holden-Day, San Francisco. Reimpresión por Aegean Park Press, 1982.

[KOH93] T. Kohda, A. Tsuneda. *Pseudonoise sequences by chaotic nonlinear maps and their correlation properties*. IEICE Trans. Communications E76-B, 855-862,1993.

[MAS69] J.L. Massey. *Shift register synthesis and BCH decoding*. IEEE Transactions on Information Theory, **15**, 122-127, 1969.

[NO89] J.S. No, P.V. Kumar. *A new family of binary pseudorandom sequences having optimal periodic correlations properties and large linear span*. IEEE Trans. Info. Theory,**35**, 373-379, 1989.

[PIC82] R. L. Pickholtz, D. L. Schilling, L. B. Milstein. *Theory of spread spectrum communications*. IEEE Trans. Communications, **30**, 855-884, 1982.

Sitios de Internet

http://www.pgpi.org/

http://csrc.nist.gov/rng/

http://www.cdc.informatik.tu-darmstadt.de/reports/TR/TI-03-07.ecprng.pdf

http://web.mit.edu/kerberos/www/

http://www.ietf.org/rfc/rfc2246.txt

http://www.openssh.com

Índice alfabético

algoritmo
  Berlekamp-Massey, 70
algoritmos
  complejidad, 3
  notación asintótica, 4
  probabilísticos, 5
    complejidad, 5

C.E. Shannon, 2
circuitos booleanos
  aleatorios, 51
  definición, 50
  familia, 52
  tamaño, 50

función
  lambda de Carmichael, 19, 43, 76

generador
  Blum-Blum-Shub, 31
    periodo, 42
  Blum-Micali, 57
  de bits aleatorios, 6
  de bits pseudoaleatorios, 6
    criptográficamente fuerte, 7
  de congruencia lineal, 11
  GBPCF, 56
  puramente multiplicativo, 18

Kolmogorov, 2

LFSR, 67
  definición, 67

mínimo exponente universal, 19, 43

parámetros del GCL, 11
  predicción, 20
PLD
  algoritmos, 48
  definición, 47
predicado, 52
  accesible, 53
  aproximable, 53
  no aproximable, 53
problema de
  logaritmo discreto, 47
  los residuos cuadráticos, 35
prueba del siguiente bit, 7
pruebas de aleatoriedad, 7
  prueba de autocorrelación, 9
  prueba de corridas, 9
  prueba de Maurer, 10
  prueba del monobit, 8
  prueba del póker, 8
  prueba serial, 8

raíz cuadrada módp, 48
  principal
    no principal, 49
residuo cuadrático, 31, 48

semilla, 6

simbolo
    de Jacobi, 31, 32, 77
    de legendre, 77
Solomonov, 2
sucesión
    de números aleatorios, 1
    periodo de la, 12
        longitud del, 12
    pseudoaleatoria, 2

teorema
    chino del residuo, 76
    de J. Plumstead, 22
    Euler, 76
    Fermat, 75