Facultad
de
Ciencias

Criptografía desde el punto de vista computacional
(A computational approach to Cryptography)

Trabajo de Fin de Grado
para acceder al

GRADO EN MATEMÁTICAS

Autor: Álvaro García García

Director: Domingo Gómez Pérez

Febrero - 2016

Índice general

1. Criptografía **5**
- 1.1. Introducción e historia 5
- 1.2. Protocolos criptográficos 7
- 1.3. Números pseudoaleatorios 9
- 1.4. Estructura del Trabajo 11

2. Complejidad **13**
- 2.1. Máquina de Turing 13
- 2.2. Lenguajes Aceptados por una Máquina de Turing 16
- 2.3. Clases de Complejidad 17

3. Fundamentos **23**
- 3.1. Funciones one-way 23
  - 3.1.1. Funciones one-way fuertes 23
  - 3.1.2. Funciones one-way débiles 24
  - 3.1.3. Funciones One-Way Débiles Implica Funciones One-Way Fuertes 25
  - 3.1.4. Función one-way Universal 25
- 3.2. Hard-core bits 26
- 3.3. Números pseudoaleatorios 27
  - 3.3.1. Blum-Blum-Shub 28
  - 3.3.2. RSA 29
  - 3.3.3. Generador de Curvas Elípticas Dual 30

Resumen

La criptografía es la ciencia que estudia métodos para enviar información de forma segura en forma eficiente. Los cimientos teóricos de la criptografía se basan en teoría de la complejidad computacional y estadística que la hacen poco accesible.

Este trabajo dará una introducción a la teoría de generación de números aleatorios, que es de suma importancia por sus aplicaciones a los cifradores en flujo. Para ello, se introducirán todos los conceptos básicos necesarios para entender la teoría. Se comenzará introduciendo la teoría de lenguajes formales y su relación con las clases de complejidad P, NP y BPP. A continuación, hablaremos de las funciones one-way (de un solo sentido) para introducir Hard-Core bits que son necesarios para definir algunos generadores de números pseudoaleatorios seguros y por último discutiremos algunas construcciones de generadores.

Keywords: Criptografía, funciones one-way, complejidad, lenguajes formales

---

Abstract

Cryptography concerns with algorithms to secure information from unauthorised third parties. The theorical basis are supported by complexity theory and statistics.

This bachelor thesis gives an introduction to the theory of pseudorandom number generation. This subject is specially important for its applications to stream ciphers. All the necessary concepts to understand the thesis are given in the first chapters. These concepts include the ones from Formal Language theory and a introductory exposition of complexity classes P, NP and BPP.

Next, the definition of one-way functions is given and the related concept of hard-core bits is explained. The last part is devoted to several constructions of secure pseudorandom number generators.

Keywords: cryptography, one-way functions, complexity, formal languages

Capítulo 1

Criptografía

1.1. Introducción e historia

La información es poder y poder esconder información a un adversario es vital en muchos aspectos de la vida. La criptografía es el estudio de algoritmos que permitan proteger la privacidad de la información frente a usuarios no autorizados. El proceso de ocultar el significado de un mensaje se llama **encriptación** y el resultado se denomina **mensaje encriptado** y el proceso inverso se conoce como desencriptación. La criptografía data al menos de la época de los romanos, donde los generales mandaban mensajes encriptados para en el caso de que el mensajero fuese capturado que los enemigos no pudiesen entender el mensaje. El método es conocido como el criptosistema Cesar. A continuación, pasamos a detallar y dar un ejemplo del funcionamiento de este criptosistema.

Este método consiste en trasladar tantos puestos como se indica, en otro mensaje que se llevaba a parte, las letras del abecedario y hacer la operación inversa para desencriptar.

A B C D E F G H I J K L M N Ñ O P Q R S T U V W X Y Z
T U V W X Y Z A B C D E F G H I J K L M N Ñ O P Q R S

En este caso, vamos a trasladar el abecedario 7 puestos las letras, y vamos a encriptar el mensaje "HOLA MUNDO", que pasaría a ser "AIET FÑGWI", y de manera inversa el mensaje "VIFX MYBAH"pasaría a ser "COME FRUTA". En nuestra época esto no es más que un encriptación de "juguete" pero si los adversarios tenían conocimientos limitados de latín, o no sabían mucho sobre encriptación en general o este método en particular, resultaría muy difícil que obtuviesen la información.

Un caso muy curioso, que se encuentra explicado más extensamente en

[17], fue el de María de Escocia la cual se cree que conspiró contra su hermana Isabel II ya que tenía aspiraciones de tomar el control de la corona (que al final obtuvo su hijo, legítimamente) y las pruebas que demostraron esta conspiración fueron unas cartas cifradas. La tarea de entender el significado de estas cartas fue asignada al mayordomo de la corona inglesa, el cuál fue miembro del servicio secreto inglés. El mayordomo al final consiguió descifrar las cartas y con ello se condenó a María a prisión. Tras este incidente, hubo más conspiraciones hacia la corona inglesa, en muchas de las cuales se cree que María estuvo implicada. Su hermana no deseaba condenarla por el precedente que sentaba, matar a alguien de sangre real era algo impensable. Solamente la acumulación de pruebas en su contra no dejo otra salida a Isabel II y se la condenó a muerte. Fue decapitada y su cuerpo quemado en 1587.

La criptografía ha cambiado mucho, pasando de ser una serie de trucos basados en creencias a basarse en fundamentos sólidos de otras áreas de las matemáticas y computación. Ahora las técnicas de encriptación y de desencriptación han avanzado, pero el concepto sigue siendo el mismo, transmitir símbolos por un canal que puede ser escuchado por un atacante y que la información no sea recuperada por nadie más que a los legítimos destinatarios de ese mensaje. La diferencia entre los usuarios legítimos y los no legítimos es una información extra, llamada clave privada o secreta, que es una cadena de símbolos. La base fundamental en criptografía es que, según A. Kerckhoffs, la seguridad debe depender solamente de lo difícil que sea averiguar la clave secreta. Esto es, el funcionamiento del algoritmo de encriptación y desencriptación son públicos.

Este principio fue enunciado por Claude Shannon como “el enemigo conoce el sistema” y lo utilizo para definir el secreto perfecto. El primer algoritmo de encriptación que demostró ser irrompible fue el conocido como one-time pad, propuesto en 1917. El one-time pad consiste en aplicar el criptosistema Cesar a cada letra, utilizando para cada letra un desplazamiento aleatorio. Todos los desplazamientos son conocidos tanto por el emisor como por el receptor del mensaje por lo que pueden cifrar y descifrar la información. Por tener la propiedad de ser irrompible, este criptosistema fue utilizado para la comunicación entre la URSS y Estados Unidos en el conocido teléfono rojo. La clave secreta que compartián erá llevada en maletines por agentes secretos. Este sistema era suficiente para la comunicación entre gobiernos pero el desarrollo de las comunicaciones telefónicas hizo que hubiera un campo para la popularización de la criptografía.

Desde 1970 se ha añadido a este campo otros problemas como la tolerancia de fallos, que lo que se pretende es que cuando un ordenador produce

un error en uno de sus bits, se sepa identificar que es un error, y las firmas digitales. La importancia de este campo de la investigación se ve en la regulación de que és una firma electrónica segura y los requisitos legales que deben cumplir las entidades certificadoras se encuentran en [8]. Realmente se podría asociar a la criptografía con el diseño de cualquier sistema que necesite protegerse de posibles ataques para obtener o dañar información.

Nosotros usamos la criptografía todos los días aunque no seamos conscientes de ello. A la hora de ir al banco, cuando hablamos por teléfono, con las facturas electrónicas, o en algunas elecciones de ocio como las apuestas o los casinos electrónicos.

Uno de los campos que más retos plantea es el campo de la criptografía en dispositivos móviles y en redes de sensores. La limitación de computo en estos dispositivos han hecho que se investigue en algoritmos muy eficientes. Una de las propuestas que se utiliza en criptografía son los conocidos de cifradores en flujo. La idea es utilizar, como en el one-time pad, aplicar un desplazamiento distinto para cada símbolo. Los desplazamientos que se aplican a cada símbolo dependiendo de varios parámetros secretos (la clave secreta). Poseyendo esta información tanto el receptor como el emisor, pueden cifrar información y, si la secuencia de desplazamientos **parece lo suficientemente aleatoria**, esperar que la información que se transmiten permanezca privada.

Uno de los algoritmos más conocidos para la encriptación de mensajes es el conocido como Advanced Encryption Standard (AES), este algoritmo está basado en uno anterior. Este algoritmo funciona de la siguiente manera:

- Este algoritmo tiene dos entradas, una de ellas es el mensaje (como máximo de 128 bits, si no se llenan los 128 bits el própio algoritmo lo rellana con ceros) M y la otra es una clave que se usará para encriptar el mensaje K.
- El proceso que se seguirá con este algoritmo será el siguiente. $AES(M, K) = C$, donde C es el mensaje cifrado.

1.2. Protocolos criptográficos

Un protocolo es una serie de pasos, que involucra a dos o más partes y está diseñado para cumplir una tarea. Otras características de un protocolo son,

- Todos los involucrados en el protocolo deben conocer todos los pasos del mismo.

- Todas las partes involucradas deben de estar de acuerdo en seguirlo.
- El protocolo no puede ser ambiguo, es decir, los pasos han de estar bien definidos y ser claros.
- Para cada situción debe de haber una acción específica.
- El protocolo tiene una terminación en un tiempo alcanzable.
- Si todas las partes siguen el protocolo, se alcanza el objetivo.

Además, si el protocolo es criptográficamente seguro, se cumple una o más de las siguientes propiedades:

- **Confidencialidad**: ninguna información adicional a la que se quiere transmitir es revelada.
- **Autoría**: cada una de las partes tiene que ser capaz de generar una etiqueta de identificación para cualquier mensaje, a la vez de saber identificar a quien corresponde cada mensaje.
- **No repudio**: que ninguna de las partes pueda rechazar la autoría de un mensaje que ha emitido.

Veamos el ejemplo de la firma digital La firma digital es un concepto que a surgido a partir de la evolución de la comunicación informática en el entorno empresarial. Las relaciones entre los métodos de cifrado y la firma se hizo posible con la "Digitalización" de ambos y la introducción del enfoque de complejidad computacional a la seguridad. Una estrategia para firmas infalsificables requiere:

- Que cada usuario sea capaz de generar de manera eficiente su propia firma en los documentos de su elección.
- Que cada usuario sea capaz de verificar de manera eficiente si el documento ha sido firmado por otro usuario.
- Que nadie sea capaz de reproducir eficientemente las firmas de otros usuarios en documentos que esos usuarios no firmaron.

Las firmas digitales comparten mucho con las firmas manuales ya que se requiere que un individuo sea capaz de firmar algo y que se pueda probar si lo ha firmado él o no de manera eficiente.

Un protocolo muy similar es el de autentificación de mensajes. Pongámonos en el caso de que un agente esta monitoreando el canal por el que dos personas se comunican de manera que puede alterar los mensajes que se mandan. Y lo que se intenta es que las partes sean capaces de distinguir que mensajes han sido modificados y cuales no. Para conseguir esto se requiere:

- Que cada una de las partes que se comunican sean capaces de generar de manera eficiente mensajes que pueda ser verificada la autoría.
- Que cada una de las partes que se comunican sean capaces de verificar si un mensaje ha sido emitido por un usuario autorizado.
- Que ningún agente externo sea capaz de producir eficientemente mensajes no emitidos por alguna de las partes que se comunican.

Creo que es importante remarcar la diferencia entre la firma y la autentificación. La autentificación sirve para que el receptor sepa quien es el autor del mensaje, mientras que la firma es algo vinculante con respecto a un documento.

1.3. Números pseudoaleatorios

Un número pseudoaleatorio es un número que es generado en un proceso que parece producir números al azar, determinista. Las secuencias de números pseudoaleatorios no muestran ningún patrón o regularidad aparente desde un punto de vista estadístico, a pesar de haber sido generadas por un algoritmo completamente determinista, con el que las mismas condiciones iniciales producen siempre el mismo resultado.

Al principio los investigadores que necesitaban secuencias de números aleatorios tenían que generarlos ellos mismos mediante dados, monedas, cartas, etc. o utilizar tablas de números aleatorios existentes. El primer intento de dotar a los investigadores con un suministro de números aleatorios tuvo lugar en 1927, cuando el Cambridge University Press publicó una tabla de 41.600 dígitos desarrollada por Leonard H.C. Tippet [19]. En 1947 la RAND Corporation generó una secuencia de números a partir de una simulación electrónica de una rueda de ruleta; los resultados fueron publicados en 1955 bajo el título A Million Random Digits with 100.000 Normal Deviates [7].

John von Neumann fue un pionero en la investigación de los generadores de números aleatorios implementados en computadoras. En 1951, Derrick

Henry Lehmer inventó el generador congruencial lineal, utilizado en un gran número de generadores pseudoaleatorios actuales [12]. Con la proliferación de los ordenadores, los algoritmos de generación de números pseudoaleatorios fueron reemplazando las tablas de números aleatorios.

Uno de los primeros generadores de números pseudoaleatorios fue el Middle-Square Method que fue descrito por primera vez alrededor de 1240 y más tarde reinventado por John von Neumann en 1949. No es un muy buen generador de números pseudoaleatorios ya que tiene varias debilidades. El método consiste en que si queremos crear una secuencia de números de 4 cifras, hemos de introducir una semilla de 4 cifras (1645), luego elevamos al cuadrado ese número (2706025), si el número resultante no tiene el doble de cifras, en este caso 8 se rellena con ceros (27060250), y a continuación se cogen los cuatro dígitos centrales (0602).

Los generadores de números aleatorios no son diseñados al azar, ya que se necesita una teoría. La mayoría de las aplicaciones simples, como los juegos de ordenador, necesitan números aleatorios con pocas propiedades, por ejemplo, para jugar al tetris con no saber cual va ser la siguiente pieza que aparezca ya es suficiente. Sin embargo, la criptografía es extremadamente sensible a las propiedades de los generadores de números aleatorios. Si se depende de un generador de números aleatorios para generar contraseñas, claves temporales o cualquier otro parámetro aleatorio, las extrañas correlaciones es lo último que se puede desear.

El problema es que un generador de números aleatorios no produce una secuencia aleatoria. Probablemente no produce nada que se parezca ni remotamente a una secuencia aleatoria. Es decir, sabiendo la función que utiliza el generador y la entrada que recibe podríamos ser capaces de reproducir el mismo resultado. Es evidente que es imposible producir algo verdaderamente aleatorio en un ordenador. Un ordenador sólo puede tener un número finito de estados (un número finito grande, pero un número finito no obstante), y el resultado que sale siempre viene dado por una función determinista, de la entrada y el estado actual del equipo. Esto significa que cualquier generador de números aleatorios en un ordenador (al menos, en una maquina de estados finitos) es, por definición, periódica. Cualquier cosa que es periódica es, por definición, predecible. Y si algo es predecible, no puede ser aleatorio.

Las secuencias de números aleatorios son muy utilizadas en general en la criptografía, ya que se necesitan números para generar claves de seguridad y si nadie puede prever la clave es más seguro. También se sabe que, si existen las funciones one-way, de las cuales hablaremos más adelante, entonces los generadores pseudoaleatorios también existen. Y es más, dada cualquier función one-way, se puede construir un generador de números pseudoalea-

torios [10]. El primer generador de números pseudoaleatorios basado en el Problema del Logaritmo Discreto (DLP sus siglas en inglés) fue propuesto por Blum y Micali en [5], y después Blum, Blum y Shub en [4] (más adelante se hablará de este método en particular) basado en el problema de la factorización de números.

Con la llegada de la criptografía de curva elíptica (ECC sus siglas en inglés), los sistemas criptográficos han disminuido su impacto, confiando su seguridad en el DLP sobre curvas elípticas. Una de las primeras normas de ECC para crear secuencias de números pseudoaleatorios, fue el generador Dual Elliptic Curve, propuesto por por el NIST (Instituto Nacional de Estándares). Por desgracia, los números generados acabaron siendo distinguibles de una secuencia de números aleatorios [16, 9]. Actualmente, el NIST recomienda generadores de números usando cifrados de bloque (como el AES), mapas de hash y códigos de autenticación de mensajes en clave hash (ver [2]) y se ha vuelto a abrir la publicación especial 800-90 para la inclusión de un nuevo generador de números pseudoaleatorios usando curvas elípticas.

El algoritmo AES antes mencionado, también funciona como un generador de números pseudoaleatorio, pero se usa de una manera un poco diferente. Primero eliges un número al azar que será la semilla, este número podría ser 123. Y en vez de aplicar AES a 123 hacemos lo siguiente: $AES(1, K)$, $AES(2, K)$, $AES(3, K)$ y de cada uno de los resultados cogemos, por ejemplo, 1 bit, que podría ser el bit de paridad.

1.4. Estructura del Trabajo

Este trabajo como se ha podido ver en el índice, está separado en tres partes bien diferenciadas.

La primera es la que acabamos de ver que es una simple introducción.

La segunda parte nos centraremos más en las Máquinas de Turing, un instrumento teórico muy potente, y los lenguajes que aceptan estas máquinas para poder explicar a continuación las clases de complejidad que es la base de que existan generadores de números pseudoaleatorios eficientes o no.

La tercera y última parte nos centraremos en las funciones one-way, y en los hard-core bits, dos temas que son necesarios para la construcción eficiente de generadores de números aleatorios que será lo que trataremos en el último punto de este trabajo, pero ahondando más que en la introducción.

Capítulo 2

Complejidad

2.1. Máquina de Turing

Una máquina de Turing es un dispositivo que manipula símbolos que se encuentran en una tira de cinta de acuerdo a una tabla de reglas. A pesar de su simplicidad, una máquina de Turing puede ser adaptada para simular la lógica de cualquier algoritmo de computador (Tesis de Church-Turing) y es particularmente útil en el estudio de las funciones de un computador.

Este formalismo fue definido por Alan Turing en 1936, ver [20]. La máquina de Turing es un dispositivo hipotético que puede leer el contenido de la cinta, borrar la cinta o modificarla en un símbolo cada paso. Intuitivamente, esto son los mínimos requerimientos que se deben hacer a un dispositivo que simule cualquier computación. La definición formal es la siguiente.

Definición 1 (Máquina de Turing)**.** *Una máquina de Turing es un modelo computacional que realiza una lectura/escritura de manera automática sobre una entrada llamada cinta, generando una salida en esta misma.*

Una máquina de Turing cuenta con las siguientes partes.

$$M = (Q, \Sigma, \Gamma, s, B, F, \delta)$$

- Q *es un conjunto finito de estados*
- Σ *es un conjunto de símbolos distintos del espacio en blanco, denominado alfabeto de entrada.*
- Γ *es el alfabeto de la máquina, el cual es el conjunto finito de símbolos de la cinta.*
- $s \in Q$ *es el estado inicial*

- $B \in \Gamma$ *es l símbolo blanco.*
- $F \subseteq Q$ *es el conjunto de estados finales.*
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *es una función de transición donde* L *es un movimiento a la izquierda y* R *es un movimiento a la derecha.*

Este modelo está formado por un alfabeto de entrada y uno de salida, un símbolo especial llamado blanco B, un conjunto de estados finitos y un conjunto de transiciones entre dichos estados. Su funcionamiento se basa en que la máquina recibe un estado inicial y una cadena de caracteres pertenecientes al alfabeto de entrada. La máquina va leyendo una celda de la cinta en cada paso, borrando el símbolo en el que se encuentra posicionado su cabezal y escribiendo un nuevo símbolo perteneciente al alfabeto de salida, para luego desplazar el cabezal a la izquierda o a la derecha. Esto se repite según se indique en la función de transición, hasta llegar a un estado final o hasta que no halla más transiciones, representando así la salida. Explicando mejor lo anterior, la máquina realiza los pasos predefinidos incluyendo la parada al finalizar o por el contrario encuentra un paso no predefinido en el que también para. Su funcionamiento se basa en una función de transición, que recibe un estado inicial y una cadena de caracteres (que se escriben en la cinta, la cuál puede es infinita) pertenecientes al alfabeto de entrada.

Ejemplo 1. *Veamos un ejemplo para aclarar los conceptos anteriores. En él queremos crear una máquina de Turing que verifique si el número de ceros en la entrada es par.*

$$M = (Q, \Sigma, \Gamma, q_0, \delta, F)$$

- $Q = \{q_0, q_1\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, 1, B\}$
- $F = \{q_2\}$
- δ *es definida como:*
  - $\delta(q_0, 0) = (q_1, B, R)$
  - $\delta(q_0, 1) = (q_0, B, R)$
  - $\delta(q_1, 0) = (q_0, B, R)$
  - $\delta(q_1, 1) = (q_1, B, R)$

- $\delta(q_0, B) = (q_2, 1, R)$
- $\delta(q_1, B) = (q_2, 0, R)$
- $\delta(q_0, B) = (q_2, 1, R)$
- $\delta(q_1, B) = (q_2, 0, R)$

De esta manera si el número de ceros es par, la máquina de Turing habrá escrito un 1, mientras que si es impar escribirá un cero.

Imaginemos ahora que tenemos la siguiente palabra 001, entonces en la cinta de entrada de la máquina de Turing tendríamos lo siguiente

$$\cdots BBB001BBBBBB\cdots$$

Es decir, la palabra introducida queda intacta al principio, pero luego está seguida de infinitas veces el símbolo B (símbolo blanco) tanto a la izquierda como a la derecha. En el primer paso, la máquina estaría en el estado q_0 y leería un 0. Para representar el estado actual de la computación, se ha elegido poner el estado justo delante de el número o letra donde estamos en ese momento

$$\cdots BBBq_0001BBBBBB\cdots$$

Después cambiaría el 0 por una B, se movería a la derecha y cambiaría el estado a q_1.

$$\cdots BBBBq_101BBBBBB\cdots$$

Para el siguiente número estaríamos en el estado q_1 y leería un 0 así que lo cambiaría por una B, se movería a la derecha y cambiaría el estado por q_0, y en ese momento tendríamos lo siguiente en nuestra máquina:

$$\cdots BBq_01BBBBBB\cdots$$

Ahora la máquina que está en el estado q_0 lee un 1, así que lo cambia por una B se mueve a la derecha y no cambia el estado.

$$\cdots BBBBq_0BBBBBB\cdots$$

Por ello, lo que ocurre es que estamos en el estado q_0 y nuestra máquina lee una B que cambia por un 1, ya que el número de ceros es par, se mueve a la derecha y llega al estado final q_2 con lo que la palabra ha sido aceptada y tenemos nuestra solución.

$$\cdots BBBBBq_21BBBB\cdots$$

2.2. Lenguajes Aceptados por una Máquina de Turing

Primero, antes de hablar de los lenguajes aceptados por una Máquina de Turing tenemos que definir lo que es un lenguaje y de las partes que consta.

Definición 2 (Alfabeto)**.** *Un alfabeto es un conjunto finito de símbolos.*

Definición 3 (Palabra)**.** *Dado un alfabeto* Σ *una palabra sobre dicho alfabeto es la yuxtaposición finita de símbolos del alfabeto.*

Definición 4 (Lenguaje)**.** *Dado un alfabeto* Σ*, un lenguaje sobre* Σ *es un conjunto de palabras sobre dicho alfabeto, es decir un subconjunto de* Σ^* *.*

Definición 5 (Lenguaje aceptado por una máquina de Turing)**.** *Dada una máquina de Turing* M*, el lenguaje aceptado por la máquina de Turing* $\mathcal{L}(M)$ *es el conjunto de todas las palabras en el que la computación de la máquina de Turing acaba en un estado final.*

Antes de hablar de la complejidad tenemos que explicar algunas cuestiones relacionadas con los lenguajes, concretamente de los lenguajes recursivamente enumerables y los lenguajes recursivos. Empecemos por definir cada uno de ellos.

Definición 6 (Lenguaje Recursivamente Enumerable)**.** *Un lenguaje* L *es recursivamente enumerable si existe una Maquina de Turing* M *tal que* $\mathcal{L}(M) = L$

Un ejemplo de lenguaje recursivamente enumerable es el lenguaje de los Palíndromos $P := \{w \in \{0,1\} | w^R = w\}$ donde w^R es el reverso de la palabra. La función f caracteristica en nuestro caso, esta definida en Σ^* y devuelve:

$$f(x) := \begin{cases} 1 & \text{si } x = x^R \\ 0 & \text{en caso contrario} \end{cases}$$

La función característica se utilizará para definir la clase de complejidad a la que pertenece el algoritmo en la siguiente sección. Definamos ahora el concepto de lenguaje Recursivo.

Definición 7 (Lenguaje Recursivo)**.** *Un lenguaje* L *es recursivo si existe una Máquina de Turing* M *que siempre para de manera que* $\mathcal{L}(M) = L$*. Una Máquina de Turing que siempre para se llama algoritmo.*

Propiedades 1. *Los lenguajes Recursivamente Enumerables y los lenguajes recursivos son cerrados con las siguientes operaciones:*

- *La unión.*
- *Intersección.*
- *Concatenación.*
- *El cierre estrella, es decir concatenado consigo mismo el número de veces que queramos.*
- *Reverso, es decir las palabras del lenguaje pero al reves.*

Una propiedad que no tienen los lenguajes Recursivamente Enumerables y que si poseen los lenguajes Recursivos es que estos últimos están cerrados con la diferencia y el complementario.

Antes hemos hablado de Maquinas de Turing que paran, lo que queremos decir con eso, es que teniendo una Máquina de Turing M y una palabra w es que M terminará en un número finito de pasos cuando es ejecutada usando w.

El problema de la parada es saber si la Máquina de Turing terminará en un número finito de pasos antes de ejecutarla. En 1936 Alan Turing demostró que este problema es indecidible. Lo que se querría por lo tanto es encontrar un programa P, que dándole un programa Q y una entrada x, nos muestre si el programa Q termina en un número finito de pasos o no.

La irresolubilidad del problema se puede mostrar de varias formas, pero en esencia todas equivalen a un argumento diagonal de Cantor. En nuestro caso nos restringiremos a máquinas de Turing que paran despúes de un número finito de pasos. Más aún, el número de pasos que realizan es polinomial en el tamaño de la entrada.

2.3. Clases de Complejidad

En informática los cálculos eficientes se asocian con la clase de complejidad P, es decir, aquellos problemas que por mucho que crezcan los bits de la entrada del problema el tiempo de ejecución de ese problema no crece de manera desproporcionada. Dentro de esta clase podemos encontrar problemas como el cálculo del máximo común divisor o el producto de matrices. La definición formal es la siguiente:

Definición 8 (Clase de Complejidad P)**.** *Un lenguaje L es reconocible en tiempo polinomial si existe una maquina de Turing determinista M y un polinomio $p()$ de manera que,*

- *Con una entrada de una palabra x, la maquina M termina después de como mucho $p(|x|)$ pasos.*
- *$M(x) = 1$ si y solo si $x \in L$.*

Asimismo, la clase de complejidad NP se asocia con problemas de cálculo que tienen soluciones que, una vez dada, se pueden validar de manera eficiente. Es decir, nos dan una posible solución del problema donde saber si es solución o no es relativamente sencillo. Aquí encontramos otros problemas muy conocidos a nivel académico como puede ser encontrar un Ciclo Hamiltaniano en un grafo, o encontrar un camino entre dos vértices de un grafo con un número de saltos especifico.

Definición 9 (Algoritmo de Verificación)**.** *Un algoritmo de verificación A es un algoritmo que toma como entrada dos palabras $x, y \in \{0,1\}^*$ y devuelve 0 ó 1.*

Definición 10 (Lenguaje Verificado)**.** *El lenguaje verificado por un algoritmo de verificación A se define de la siguiente forma:*

$$L = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^*,\ A(x,y) = 1\}.$$

Definición 11 (Clase de Complejidad NP)**.** *La clase de complejidad NP es la clase de lenguajes que pueden ser verificados en tiempo polinomial. Es decir, son los lenguajes L para los cuales existe un algoritmo polinomial A y*

$$L = \{x \in \{0,1\} \mid \exists y \in \{0,1\}^*, |y| = O(|x|^c),\ A(x,y) = 1\},$$

donde c es un número mayor que 1 y la constante es absoluta.

A parte de la definición que acabamos de ver de la clase de complejidad NP, se ha decidido añadir esta otra que se puede consultar en [11].

Definición 12 (Clase de Complejidad NP)**.** *Un lenguaje L esta en NP si existe una relación booleana $R_L \subseteq \{0,1\}^* \times \{0,1\}^*$ y un polinomio $p()$ de manera que R_L pueda reconocer en tiempo polinomial, y $x \in L$ si y solo si existe un y de manera que $|y| \leq p(|x|)$ y $(x,y) \in R_L$.*

Por lo tanto, NP consiste en el conjunto de lenguajes que pueden ser verificados de manera eficiente. El problema abierto en teoría de complejidad que ha despertado mayor interés es si $P = NP$. Aunque no hay pruebas definitivas, la opinión más generalizada es que $P \neq NP$.

La clase de complejidad NP-Completo podríamos definirlo como aquellos problemas de la clase NP a los que se puede reducir cualquier problema de la clase NP. El problema de satisfacción de circuitos booleanos es un problema NP completo. Una idea de la demostración se puede encontrar en [6].

Definición 13 (Clase de Complejidad NP-Completo)**.** *Un lenguaje L' es NP-Completo si está en NP y cada lenguaje en NP es reducible en tiempo polinomial a L'.*

Cuando hablamos de un lenguaje NP es reducible en tiempo polinomial a otro lenguaje, lo que se quiere decir es que, si tenemos que comprobar si una palabra x pertenece a un lenguaje L , entonces podemos transformar en tiempo polinomial el problema de ver si otra palabra x' pertenece a otro lenguaje L'.

Asociar computación eficiente con BPP

Ahora vamos a mirar cálculos en tiempo polinómico probabilístico, de manera que vamos a considerar eficientes aquellos algoritmos aleatorios para los cuales el tiempo de ejecución está limitado por un polinomio en la longitud de la entrada. La clase de complejidad que recoge estos algoritmos es BPP.

Definición 14 (BPP)**.** *Decimos que L es reconocido por el polinomio probabilístico de la maquina de Turing M si*

- *Para cada* $x \in L$ *se cumple que* $Pr[M(x) = 1] \geq 2/3$
- *Para cada* $x \notin L$ *se cumple que* $Pr[M(x) = 0] \geq 2/3$

Las máquinas de Turing probabilistas toman decisiones basándose también en el azar. De manera informal, podemos definir una Máquina de Turing Probabilística (MTP) como una máquina en el que las decisiones se toman en base a una determinada probabilidad.

En la definición que acabamos de ver el 2/3 se puede sustituir por cualquier numero mayor que 1/2 y menor que 1 y se obtiene una definición equivalente. Durante mucho tiempo se usaron algoritmos con complejidad BPP para determinar si un número era primo o no, hasta que en 2004 se encontro un algoritmo en la clase de complejidad P [1].

Veamos ahora un ejemplo en más profundidad con el Test de Solovay-Strassen. Este test sirve para saber si un número no par es primo o no.

Este test tiene una característica que es que si dice que un número no par n no es primo, puedes estar seguro de que ese número no es primo, el problema viene cuando dice que el número es primo, ya que ahí es cuando el método no es 100 % fiable.

El proceso del test, para saber si un número natural no par n es primo, es el siguiente:

- Se elige a un natural al azar entre 2 y $n-2$.
- Se calculan u y v de la siguiente manera. $u = a^{\frac{n-1}{2}} (\text{mod } n)$ y $v = (\frac{a}{n})$ es el símbolo de Jacobi.
- Una vez hallados u y v, simplemente hay que ver si $u \neq v$ lo cual implica que n no es primo, o $u = v$ por lo que es posible que n sea primo.

Lo que hay que hacer por lo tanto es que si con un determinado n tenemos el resultado de que es posible que sea un primo, habrá que repetir el proceso varias veces para poder asegurar que ese número es primo con una probabilidad elevada. Con este método si tenemos un número n no primo, en el peor de los casos vamos a tener una probabilidad de fallar de $\frac{1}{2}$ por eso repitiendo el método varias veces hacer que esta probabilidad baje de manera que la probabilidad de fallar en el peor de los casos es menor o igual a $(\frac{1}{2})^k$ donde k es el número de veces que se ha repetido el proceso.

Ejemplo 2. *Empecemos cogiendo un número que no sea primo como por ejemplo 15. Y un número al azar entre 2 y 13 como puede ser el 11. En este caso en concreto* $u = 11$ *y* $v = -1$ *por lo que podemos afirmar que 15 no es primo.*

En cambio si cogemos $n = 7$ *tenemos que para:*

- $a = 1$ $v = 1$ *y* $u = 1$ *por lo que 7 podría ser primo.*
- $a = 2$ $v = 1$ *y* $u = 1$ *por lo que 7 podría ser primo.*
- $a = 3$ $v = -1$ *y* $u = 6 = -1$ *por lo que 7 podría ser primo.*
- $a = 4$ $v = 1$ *y* $u = 1$ *por lo que 7 podría ser primo.*
- $a = 5$ $v = -1$ *y* $u = 6 = -1$ *por lo que 7 podría ser primo.*

En este caso al tratarse de números pequeños hemos podido comprobar todos y ver que en efecto 7 es un número primo.

Hemos visto en la introducción lo importante que es que existan algoritmos que no sean fáciles de resolver para poder encriptar mensajes, a la vez que han de ser eficientes, ya que no interesa mandar un mensaje que tarde mucho tirmpo en encriptarse. En este capítulo hemos hablado de las clases de complejidad, que hemos visto que hay algunos problemas fáciles de computar y otros que son fáciles saber si una posible solución es solución o no. En el siguiente capítulo introduciremos las funciones one-way (o funciones de un solo sentido). Estás funciones son ejecutables en tiempo polinomial en tamaño de la entrada mientras que invertirlas es un proceso muy costoso, pero es fácil comprobar si una solución lo és o no. Así que estas funciones pueden existir siempre y cuando $P \neq NP$ y que resolver un problema NP completo sea, en la mayoría de los casos, difícil. Y este tipo de funciones son las que permiten que encriptar un mensaje sea un procesa rápido pero al contrario desencriptarlo sea un proceso realmente difícil. A la vez la existencia de este tipo de funciones, como se menciono en el primer capítulo, permiten la creación de generadores de números pseudoaleatorios.

Capítulo 3

Fundamentos

3.1. Funciones one-way

En este capítulo vamos a hablar de las funciones one-way, vamos a definirlas y estudiarlas. Para tener una primera idea, las funciones one-way son funciones que son fácilmente evaluables, pero difíciles de invertir. Se sabe que existen entonces $NP \neq P$. Un ejemplo claro de esto es la multiplicación; multiplicar dos números es un proceso muy sencillo, que se encontraría dentro de los problemas de complejidad P, pero dar los factores de un producto puede ser muy complicado si es producto de dos números primos del mismo tamaño. Este problema esta en la clase de complejidad NP, aunque también se sabe que no es NP-completo.

3.1.1. Funciones one-way fuertes

A partir de esta sección, utilizaremos U_n para denotar una variable aleatoria con distribución uniforme en $\{0,1\}^n$. Una definición de andar por casa sería que es una función fácil de evaluar y prácticamente en todos los casos es difícil de invertir, la definición que damos ahora más restrictiva que la definición 16.

Definición 15 (Función one-way fuerte)**.** *Una funcion* $f : \{0,1\}^* \to \{0,1\}^*$ *se llama one-way fuerte si se cumplen las siguientes condiciones.*

- *Fácil de computar: Existe un algoritmo de tiempo polinómico A de manera que para un input x el algoritmo A devuelve f(x).*
- *Difícil de invertir: Para cada algoritmo probabilista de tiempo polino-*

mial A', para todo polinomio p(), y un n suficientemente grande,

$$Pr[A'(f(U_n), 1^n) \in f^{-1}(f(U_n))] < \frac{1}{p(n)}$$

Tenniendo en cuenta que U_n es una variable aleatoria uniformemente distribuida sobre $\{0,1\}^n$

Notad que para hallar la inversa de la función se nos da explicitamente la longitud de la contra imagen. Una de las razones que se hace esto es para evitar que funciones que comprimen la imagen sean trivialmente funciones one-way, para más detalles consultar [11].

3.1.2. Funciones one-way débiles

La primera definición puede parecer muy restrictiva. Una definición no estricta sería que los algoritmos que invierten estas funciones en tiempo polinomial fallan con una probabilidad no despreciable. Esta es la idea de la siguiente definición.

Definición 16 (Función one-way débil)**.** *Una función $f : \{0,1\}^* \to \{0,1\}^*$ se llama one-way débil si se cumplen las siguientes condiciones.*

- *Fácil de computar: Existe un algoritmo de tiempo polinómico A de manera que para un input x el algoritmo A devuelve f(x).*

- *Ligeramente difícil de computar: Existe un polinomio p() de manera que para cada algoritmo probabilista de tiempo polinomial A' y un n suficientemente grande,*

$$Pr[A'(f(U_n), 1^n) \notin f^{-1}(f(U_n))] > \frac{1}{p(n)}$$

Un ejemplo de función one-way débil sería la función que multiplica dos números cualesquiera. Es muy fácil de realizar, pero la inversa que consistiría en encontrar dos factores de un número. Si el número que se pretende multiplicar es par, se encontrará una solución casi instantáneamente, pero habrá bastantes casos en los que no sea tan fácil calcularlo, pudiendo llegar a ser un problema muy costoso de calcular.

3.1.3. Funciones One-Way Débiles Implica Funciones One-Way Fuertes

Antes de seguir, quiero aclarar que no todas las funciones one-way débiles son también fuertes. La demostración que precede al teorema que se va a enunciar ahora se puede encontrar en [11]

Teorema 1. *Las funciones one-way débiles existen si y solo si las funciones one-way fuertes también existen.*

Demostración. Esta demostración no es una demostración al uso, simplemente vamos a dar una idea general de la idea de como convertir funciones one-way débiles a funciones one-way.

La idea es que tenemos una función f que es one-way débil. Entonces se construye una función g de manera que $g(x_1, ..., x_{t(n)}) = f(x_1), ..., f(x_{t(n)})$, y de manera que $|x_i| = |x_j|$ $\forall 1 \leqslant i, j \leq t(n)$, donde $t(n)$ es un polinomio que crezca suficientemente rápido. Esto significa que la entrada de g esta dividida en $t(n)$ entradas de la misma longitud y se aplica f a cada una. Para computar g se ha de computar a cada una de las entradas f, y para hallar la contraimagen de g hay que hallar la contraimagen de cada $f(x_i)$. Y la idea es llegar a que g es una función one-way fuerte porque para alguna de esas entradas, f será, con una probabilidad alta, difícil de invertir. □

Para entender esta construcción un poco mejor veamos un ejemplo. Como hemos visto antes una función f de manera que $f(x, y) = x \cdot y$ una función one-way débil. Ahora creamos una función g de manera que $g(x_1, x_2, ..., x_t) = f(x_1, x_2) \cdot ... \cdot f(x_{t-1}, x_t) = x_1 \cdot x_2 \cdot ... \cdot x_t$. Como vimos anteriormente la función f es one-way débil porque es probable que uno de los factores que se multiplican sea par o múltiplo de tres y de esa manera se factoriza muy rápido, el problema es cuando los dos factores eran números primos bastante grandes, porque entonces el tiempo de computación para hallar la contraimagen se dispara. Al crear g de esta manera al querer calcular la contraimagen con que dos de los factores sean primos muy grandes el tiempo de computación sera enorme, y si t es un número muy grande eso puede ocurrir con una probabilidad bastante elevada.

3.1.4. Función one-way Universal

La siguiente construcción es un ejemplo de una función que es one-way débil si existen funciones one-way. Las funciones universales para hacernos una idea más clara, es una función f con dos entradas una descripción de una máquina de Turing M y una entrada x, que se corresponde con el

lenguaje de M, y ha de tener fin, no puede quedarse en un bucle infinito. Forzaremos que la máquina acabe siempre, bien porque ha encontrado una solución satisfactoria o bien porque llega a un número máximo de iteraciones permitidas, predeterminado desde el principio.

El primer problema que esto puede plantear es como introducir la Máquina de Turing en la función. Una manera de hacerlo si tomamos como referencia la máquina de Turing que creé en el ejercicio sería la siguiente:

$$Desc(M) = (q_0, 0), (q_1, B, R), ..., (q_1, B), (q_2, 0, R).$$

Entonces para el ejercicio que hemos mencionado la entrada para una función universal que evalue si de una palabra el número de ceros es par sería $f(Desc(M), x)$ donde x sería la palabra a evaluar. También es posible codificarla solamente utilizando unos y ceros, pero eso no lo vamos a discutir aquí porque no modifica la idea de la demostración.

Para demostrar que es una función one-way débil solamente hay que ver algunos valores donde sea difícil de invertir. Si la función que estamos evaluando con la máquina de Turing resulta que es una función one-way será difícil de invertir para muchas palabras x. Esta es la idea general. Esta proposición y su defininión también han sido consultadas en [11]. Se deduce inmediatamente el siguiente resultado.

Proposición 1. *Existe una función computable en tiempo polinomial que es one-way si y sólo si existen funciones one-way.*

3.2. Hard-core bits

Definición 17 (Hard-core bits)**.** *Sea f una función de un solo sentido. Una función $b : \{0,1\}^* \to \{0,1\}$ es un Hard-core bit si,*

- *b es computable eficientemente*
- *para todos los algoritmos probabilísticos en tiempo polinomial existe una función ε que tiende a cero cuando $n \mapsto \infty$, es decir para todo $n \in \mathbb{N}$,*

$$Pr_x[A(1^n, f(x)) = b(x)] \leq \frac{1}{2} + \varepsilon(n)$$

Conjetura 1. *Toda función de un solo sentido tiene un Hard-core bit.*

Desafortunadamente no sabemos si esto es verdad o no, pero lo que si sabemos por [11] es que toda función f de un solo sentido, puede ser

transformada en otra función f' de manera que f' es de un solo sentido y tiene un Hard-core bit.

También es importante saber que si tenemos la salida de una one-way function hay bits de la entrada que no podemos adivinar cuales son en un tiempo polinomial con probabilidad mucho mayor que un medio([11]). Un ejemplo sencillo de esto es la funcion $f(x, y) = x \cdot y$ si ponemos $x = 3$ e $y = 5$ $f(x, y) = 15$, esto en binario sería $x = 101$, $y = 1001$, $15 = 1111$. Como 15 es un número impar su representación binaria acaba en 1, por lo tanto si preguntamos a alguien cual es el séptimo bit de la entrada, que coincide con el último, nos podría decir con una probabilidad de al menos $\frac{2}{3}$ que es 1, en cambio si le preguntamos cual es el quinto bit de la entrada ya no sería tan trivial tener más de $\frac{1}{2}$ de opciones de acertar sin hacer algún calculo adicional. Si en vez de números primos de una o dos cifras usásemos números primos de 2000 cifras el tiempo que llevaría factorizar el número para hallar el bit que les pidamos sería enorme y cualquier otra método para adivinar ese bit tendría una probabilidad muy alta de equivocarse, y eso convierte a ciertos bits en hard-core bits.

3.3. Números pseudoaleatorios

Ya hemos visto que los números pseudoaleatorios no tienen nada de aleatorios, pero si que nos podrían valer siempre y cuando tardemos mucho en darnos cuenta de que no lo son, es decir, si en tiempo polinómico no nos damos cuenta de que no son aleatorios puede que sean lo suficientemente buenos para ser utilizados en protocolos criptográficos o como secuencias para cifrar en flujo.

Definición 18 (Probabilidad de conjuntos)**.** *Sea I un conjunto de índices contable. Un conjunto indexado por I que es una secuencia de variables aleatorias indexadas por I. Es decir, cualquier $X = \{X_i\}_{i \in I}$, donde cada X_i es una variable aleatoria, es un conjunto indexado por I.*

Definición 19 (Indestinguible en tiempo polinomial)**.** *Dos conjuntos, $X = X_n$ $n \in N$ e $Y = Y_n$ $n \in N$, son indistinguibles en tiempo polinomial si para cada algoritmo probabilístico en tiempo polinomial D, cada polinomio $p(\cdot)$ positivo , y para todo n suficientemente grande,*

$$|Pr[D(X_n, 1^n) = 1] - Pr[D(Y_n, 1^n) = 1]| < \frac{1}{p(n)}.$$

En la definición anterior ha aparecido $D(X_n, 1^n) = 1$, esto es, para comprobar cuales de las palabras han sido generadas por nuestro generador en

un tiempo polinomial. Esto es un poco más débil que decir que el lenguaje de las secuencias generadas pertenece a P.

Ahora la cuestión es como decidir si un conjunto es indistinguible en tiempo polinomial o no, y para eso tenemos test que nos ayudan a ello. Hay muchos test que sirven para esto y entre ellos podemos encontrar los test empíricos, los test teóricos y los test espectrales.

3.3.1. Blum-Blum-Shub

Ahora vamos a ver uno de los generadores de números aleatorios más famosos como es el Blum-Blum-Shub o también conocido como BBS, propuesto en 1986 por Lenore Blum, Manuel Blum y Michael Shub [5].

Algoritmo 1. *Éste es un algoritmo recursivo. La ecuación de congruencia se establece como:* $x_i = (x_{i-1})^2 \ mod \ p \cdot q \ i \in \mathbb{N}$.

Para conseguir una secuencia óptima de números pseudoaleatorios hay que tener ciertas consideraciones con los parámetros que se han elegido.

- p, q *son números primos tales que* $p = q = 3 \ mod \ 4$, *además el máximo común divisor entre* $\varphi(p) = p - 1$ *y* $\varphi(q) = q - 1$ *debe ser mínimo*
- *La semilla llamada* x_0 *debe ser primo relativo con el producto* pq.

Con este algoritmo se ha de generar una secuencia binaria pseudoaleatoria, y para esto la elección del bit se toma de la siguiente manera.

- *Se toma el bit menos significativo de* x_i.
- *Se toma el bit más significativo de* x_i.
- *Se toma el bit de paridad de* x_i.

La idea de coger uno de estos bits es que son aquellos que se conjeturan que son hard-core bits, y de esa manera la posibilidad de que alguien pueda acertar de antemano un bit conociendo los anteriores tiene una probabilidad muy cercana a $\frac{1}{2}$ *si* N *es suficientemente grande.*

Ejemplo 3. *Hagamos ahora un pequeño ejemplo para entender como funciona. Vamos a hacer la elección del bit tomando el bit de paridad. Tomemos* $p = 7$ $q = 19$ $pq = 133$ *y* $x_0 = 25$. *De esta manera tenemos que* $x_1 = 25^2 \ mod \ 133 = 93$, $x_2 = 93^2 \ mod \ 133 = 4$, $x_3 = 4^2 \ mod \ 133 = 16$, $x_4 = 16^2 \ mod \ 133 = 123$ *por lo tanto la salida que obtenemos de aquí es 1,1,1,0.*

3.3.2. RSA

Ahora vamos a hablar del generador de números aleatorios RSA o también conocido como "power generator". Este generador de números aleatorios es algo parecido al de BBS, tanto en su construcción como en el uso de los hard-core bits para hacer más difícil saber cuál puede ser el número que nos va a devolver. Ahora vamos a ver su algoritmo [18].

Algoritmo 2. *Éste es un algoritmo recursivo, al igual que el anterior. La ecuación de congruencia se establece como:* $x_i = (x_{i-1})^e \ mod \ N \ i \in \mathbb{N}$. *En este algortimo no se pide que* N *sea producto de primos. Lo que si tiene que cumplir este método es que* e *sea coprimo con* $\varphi(N)$.

En el generador de números aleatorios BBS, vimos como por cada iteración se generaba un único bit, de la forma que quisiésemos de las que se nos ofrecía. En este método podemos generar el bit exactamente de la misma manera, pero también se admite que se saquen más. Si N *tiene una longitud de* n *bits, se pueden llegar a generar* $r = O(log(n))$ *bits por iteración y aun así el método continuaría siendo seguro.*

Ejemplo 4. *Veamos ahora un ejemplo en el que usemos este generador de números aleatorios. Pongamos que* $N = 17$ *y que nuestra semilla* $x_0 = 3$. *Entonces tenemos que* $\varphi(17) = 16$ *así que podemos elegir* $e = 5$. *También voy a fijar que vamos a generar el bit minimal de cada iteración.* $x_1 = 3^5 \ mod \ 17 = 7$, $x_1 = 7^5 \ mod \ 17 = 11$, $x_3 = 11^5 \ mod \ 17 = 10$, $x_4 = 3^5 \ mod \ 17 = 6$, *por lo tanto la salida que obtenemos aquí es 1,1,0,0.*

Una desventaja que tiene este método al usar números pequeños es que es muy fácil que los números empiecen a repetirse antes de lo que nos gustaría, pero si se usan números suficientemente grandes eso no debería de ser un problema habitual.

En la introducción habíamos hablado del algoritmo AES para generar números pseudoaleatorios. Hay una tabla en [13] donde se compara ese método con el AES. Lo que se ha hecho es una previsión de la capacidad de computación para el 2010 (ya que el trabajo en cuestión es del 2001), 2020 y para el 2030. Y nos muestra el número de bits necesario para que sean equivalentemente seguros, es decir, que se requiere el mismo esfuerzo computacional para en media "romper"la seguridad del generador. Antes mencionare que en la introducción hablé de que AES se usaba con 128 bits, pero hay otros dos métodos que usan 192 y 256 bits.

En 2010 RSA necesitaba 3560 bits para ser equivalentemente seguro a AES-128, 8493 bits para ser equivalentemente seguro a AES-192 y 16246

bits para ser equivalentemente seguro a AES-256. En 2020 RSA necesitará 3956 bits para ser equivalentemente seguro a AES-128, 9169 bits para ser equivalentemente seguro a AES-192 y 17235 bits para ser equivalentemente seguro a AES-256. En 2030 RSA necesita 3956 bits para ser equivalentemente seguro a AES-128, 9860 bits para ser equivalentemente seguro a AES-192 y 18260 bits para ser equivalentemente seguro a AES-256. Esto nos da una idea clara de la potencia de AES respecto a RSA a la hora de la seguridad.

3.3.3. Generador de Curvas Elípticas Dual

El generador de Curvas Elípticas Duales es un generador de números pseudoaleatorio propuesto para estándar para NIST como generador de números pseudoaleatorios en [3].

Sea $\mathbb{F}_p$ un cuerpo finito con p elementos, donde los elementos de $\mathbb{F}_p$ son representados como números en el rango $\{0, ..., p-1\}$. Dados a y $b \in \mathbb{F}_p$, cogemos el siguiente conjunto de puntos.

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p^2 \mid y^2 = x^3 + ax + b\}$$

$E(\mathbb{F}_p)$ se llama una *curva elíptica*. En una curva elíptica se puede definir una operación, de forma que los puntos de la curva elíptica forman un grupo abeliano. La operación entre dos puntos se puede computar eficientemente, además dado un punto P y un entero s, denotaremos sP al resultado de sumar P s veces consigo mismo. Ahora explicaremos como funciona el generador de curvas elípticas dual, para más detalles ver [16].

Algoritmo 3. *Lo primero se elige una semilla $s_0 \in \{0, 1, ..., \#E(\mathbb{F}_p) - 1\}$. Una vez tenemos la semilla se eligen dos puntos cualesquiera de $E(\mathbb{F}_p)$ que en este caso los llamaremos P y Q. Una vez hecho esto vamos a hallar s_1 que es la coordenada de las x resultante de la operación s_0P. Ahora tenemos que hallar r_1 que es el resultado de coger los 240 últimos bits de la coordenada de las x de la operación s_0Q. Y repetimos el proceso cambiando s_0 por s_1 tantas veces como bits queramos sacar.*

Este generador se demostró que no es seguro, porque 240 bits son demasiados bits, [16], pero se sabe que el último bit es un hard core bit de la exponenciación en curvas elipticas. Esto quiere decir que modificaciones del generador que emita solo el último bit lo hacen seguro.

Bibliografía

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004.

[2] Elaine Barker, John Kelsey, et al. NIST special publication 800-90a: Recommendation for random number generation using deterministic random bit generators. 2012.

[3] Elaine B Barker and John Michael Kelsey. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators (revised)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, 2007.

[4] Lenore Blum, Manuel Blum, and Mike Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15(2):364–383, 1986.

[5] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM Journal on Computing*, 13(4):850–864, 1984.

[6] Thomas H Cormen. *Introduction to Algorithms*. MIT press, 2009.

[7] Rand Corporation. *A Million Random Digits with 100,000 Normal Deviates*. Minnesota Historical Society, 2001.

[8] Ministerio de España. El real decreto ley 14/1999, de 17 de septiembre, sobre firma electrónica, https://www.boe.es/buscar/doc.php?id=boe-a-2003-23399.

[9] Kristian Gjøsteen. Comments on dual-EC-DRBG/NIST sp 800-90, draft december 2005, march 2006.

[10] Oded Goldreich. *Foundations of Cryptography: volume 1, basic applications*. Cambridge university press, 2004.

[11] Oded Goldreich. *Foundations of Cryptography: volume 2, basic applications*. Cambridge university press, 2004.

[12] Donald E Knuth. The art of computer programming (volume 2), 1981.

[13] Arjen K Lenstra. Unbelievable security matching aes security using public key systems. In *Advances in Cryptology—ASIACRYPT 2001*, pages 67–86. Springer, 2001.

[14] Rafael Pass. Simulation in quasi-polynomial time, and its application to protocol composition. In *Advances in Cryptology—EUROCRYPT 2003*, pages 160–176. Springer, 2003.

[15] Bruce Schneider. *Applied Cryptography: Protocols, algorithms, and source code in C*. John Wiley & Sons, 1996.

[16] Berry Schoenmakers and Andrey Sidorenko. Cryptanalysis of the dual elliptic curve pseudorandom generator. *IACR Cryptology ePrint Archive*, 2006:190, 2006.

[17] Simon Singh. *The Code Book: the evolution of secrecy from Mary, Queen of Scots, to quantum cryptography*. Doubleday, 1999.

[18] Ron Steinfeld, Josef Pieprzyk, and Huaxiong Wang. On the provable security of an efficient rsa-based pseudorandom generator. In *Advances in Cryptology–ASIACRYPT 2006*, pages 194–209. Springer, 2006.

[19] Leonard H.C. Tippet. *Tables of Random Samples From a Normal Distribution*. Cambridge University, 1927.

[20] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[21] Luis Miguel Pardo Vasallo. La conjetura de cook (¿ p= np?). parte i: lo básico. *Gaceta de la Real Sociedad Matematica Española*, 15(1):117–148, 2012.