Diseño de un nuevo generador de secuencias de bits aleatorios por entrada de teclado

Pedro María Alcover Garau[1], José M. García Carrasco[2], Luis Hernández Encinas[3]

[1] Depto. de Tecnología de la Información y las Comunicaciones, Universidad Politécnica de Cartagena; [2] Depto. de Ingeniería y Tecnología de Computadores, Universidad de Murcia; [3] Depto. de Tratamiento de la Información y Codificación, Consejo Superior de Investigaciones Científicas - Instituto de Física Aplicada

<pedro.alcover@upct.es>, <jmgarcia@ditec.um.es>, <luis@iec.csic.es>

***Resumen**: en numerosas situaciones es necesario utilizar colecciones de números o de bits generados de forma aleatoria (método de Monte Carlo, teoría de juegos, simulaciones, diseño de circuitos, etc.). En particular, la mayor parte de los algoritmos y protocolos criptográficos requieren de la generación de grandes cantidades de bits (generados de forma aleatoria o pseudoaleatoria), bien para los procesos de cifrado en flujo, bien para la generación de claves. Tales secuencias de bits deben poder ser consideradas aleatorias en el sentido de que su comportamiento no debe ser previsible por un atacante al criptosistema. En este artículo se presenta un algoritmo de generación de secuencias de bits aleatorias basado en las pulsaciones arbitrarias que un usuario lleva a cabo sobre un teclado de ordenador. Se estudia el comportamiento estadístico de diferentes secuencias obtenidas por este generador y se concluye que su comportamiento es aleatorio.*

***Palabras clave**: generador de secuencias aleatorias, criptografía, tests de aleatoriedad.*

1. Introducción

Es sabido que en determinadas situaciones se hace preciso generar secuencias de números o de bits de forma aleatoria, como en los métodos de Monte Carlo, teoría de juegos, simulaciones, diseño de circuitos, etc. Esta generación no puede hacerse, en general, de forma manual, por lo que se suele recurrir a métodos informáticos basados en hardware o software. En el primer caso se emplean fenómenos físicos controlados por ordenador para recuperar la secuencia aleatoria generada, como la excitación de átomos radiactivos, ruido térmico de un diodo semiconductor, etc., en cuyo caso las secuencias obtenidas son aleatorias. Cuando la generación se hace mediante software se suelen emplear algoritmos determinísticos, de modo que las secuencias generadas son pseudoaleatorias, es decir, cada vez que se ejecuta el mismo algoritmo con los mismos parámetros, se obtiene la misma colección de bits.

En el caso de la criptología, y más concretamente en los cifrados en flujo ([12]) se utilizan generadores pseudoaleatorios con el fin de obtener secuencias de bits que, sumadas bit a bit con los del mensaje o del criptograma, permiten cifrar y descifrar la información que se transmite. En estos casos, la clave del criptosistema es la semilla del generador, que necesariamente ha de ser una secuencia realmente aleatoria mucho más corta que la secuencia cifrante.

Exiten compiladores y otras herramientas de programación que ofrecen al usuario librerías para la generación de números aleatorios. Sin embargo, estas librerías no suelen ser apropiadas en criptología porque no ofrecen la seguridad necesaria como para evitar que un adversario pueda generar la misma secuencia utilizando técnicas estadísticas y herramientas del cálculo de probabilidades. Así pues, es necesario disponer de generadores de secuencias de bits que ofrezcan la suficiente seguridad como para ser empleadas en este tipo de supuestos. Una condición necesaria de seguridad es que tales secuencias deben pasar determinados tests estadísticos diseñados *ad hoc* de modo que un atacante no pueda, con una probabilidad mayor que $1/2$, conocer el bit que sigue a una secuencia parcial conocida. Tales tests están recomendados tanto en la literatura al uso ([12]) como por diferentes organismos de estandarización internacionales ([5][4]).
En este trabajo se presenta un generador de secuencias de bits por entrada de teclado y los resultados de someter las secuencias obtenidas a los tests de aleatoriedad estándares. Se concluye que las secuencias generadas pasan tales tests y, por tanto, que pueden ser consideradas aleatorias.

El resto de este trabajo se organiza como sigue. En la **sección 2** se recoge un breve resumen sobre los tests estadísticos empleados. En la **sección 3** se describe el generador basado en pulsaciones de teclado. En la **sección 4** se muestran algunos ejemplos del funcionamiento del generador. Los resultados del análisis estadístico del generador son recogidos en la **sección 5**. Finalmente, en la **sección 6** se incluyen las conclusiones de este trabajo.

2. Análisis de aleatoriedad

Como es sabido, no se dispone de ninguna prueba matemática que asegure de forma categórica la aleatoriedad de una secuencia de bits. No obstante, si las secuencias obtenidas mediante un determinado generador superan todos los tests diseñados con tal fin, entonces es aceptado como generador de secuencias aleatorias. A continuación se presentan las características fundamentales de los tests recomendados para este análisis.

2.1. Postulados de aleatoriedad de Golomb

Los postulados establecidos por Golomb [7][12] tienen un importante interés histórico puesto que fueron los primeros definidos y usados para establecer las principales condiciones necesarias para que una secuencia de bits pudiera ser considerada aleatoria. Sin embargo, en la actualidad estas condiciones no se consideran suficientes, aunque sí necesarias, para poder aceptar como aleatoria una secuencia estudiada.

Los postulados de aleatoriedad de Golomb son los siguientes:

1. En la secuencia de bits de longitud n,s_n, el número de unos debe diferir del número de ceros, como máximo, en una unidad.
2. En la secuencia s_n , al menos la mitad de las cadenas de dígitos iguales (precedidas y seguidas por el otro dígito) tiene longitud 1, al menos una cuarta parte de esas cadenas tienen longitud 2, al menos una octava parte tienen longitud 3, etc. Y además para cada una de las longitudes se dispone de tantas cadenas de unos como cadenas de ceros. (El postulado 2 implica el postulado 1).
3. La función de correlación $C(T)$)tiene dos valores. Esto es, para algún valor K

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

Cada uno de estos postulados tiene su traducción inmediata mediante un test de aleatoriedad, como se verá posteriormente.

2.2. Tests de contraste de hipótesis

Sea $s = s_0,s_1,s_2,\ldots s_{n-1}$ una secuencia binaria de longitud n. Los cinco tests que se presentan a continuación son usados habitualmente para determinar si la secuencia binaria s posee características que permitan considerarla como verdaderamente aleatoria.

1. Test de Frecuencia (test monobit). Determina si el número de ceros y el número de unos en la secuencia *s* son aproximadamente los mismos. Si llamamos n_0 y n_1 al número de ceros y de unos de la secuencia, el valor estadístico a contrastar es:

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

que sigue una distribución Chi cuadrado con un grado de libertad ($l = 1$). Este test se basa en el primer postulado de Golomb.
2. Test de Series (test de los dos bits). Busca y cuenta el número de ocurrencias de las cuatro subsecuencias: 00, 01, 10 y 11 y estudia si su distribución es tal y como se esperaría de una secuencia generada por un generador verdaderamente aleatorio.

Se definen los parámetros n_0, n_{01}, n_{10} y n_{11} como el número de ocurrencias de cada una de las cuatro subsecuencias anteriores. Se verifica que $n_{00} + n_{01} + n_{10} + n_{11} = n-1$.

El valor estadístico que se contrasta en este test es:

$$X_2 = \frac{4}{n-1}(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2) - \frac{2}{n}(n_0^2 + n_1^2) + 1$$

que sigue una distribución Chi cuadrado con dos grados de libertad ($l = 2$).
3. Test de Póker. Estudia todas las subsecuencias diferentes de un tamaño *m* que se determina previamente. El valor *m* es el mayor entero positivo que verifique que $\lfloor n/m \rfloor \geq 5 \cdot (2^m)$. Se toma entonces el valor $k = \lfloor n/m \rfloor$ y se divide la secuencia *s* en *k* partes no solapadas, cada una de ellas de longitud *m*. Se definen 2^m parámetros n_i $(1 \leq i \leq 2^m)$ para almacenar el número de ocurrencias de cada una de las secuencias de la longitud indicada. Este test determina si cada una de las posibles secuencias de longitud *m* aparecen un número semejante de veces.

El valor estadístico estudiado es:

$$X_3 = \frac{2^m}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k$$

que sigue una distribución Chi cuadrado con $2^m - 1$ grados de libertad ($l = 2^m - 1$).
4. Test de Rachas. El test toma una secuencia binaria *s* y calcula el número de subsecuencias de *s* formadas por una determinada cantidad consecutiva de ceros o de unos, y que no vienen ni precedidas ni seguidas por el mismo dígito. Si la racha está formada por ceros se conoce como hueco mientras que si está formada por unos se denomina bloque.

Para determinar la longitud de las diferentes rachas a estudiar, primero se calculan los sucesivos parámetros e_i definidos según la siguiente expresión:

$$e_i = \frac{n - i - 3}{2^{i+2}} \text{ , para } 1 \leq i \leq k$$

donde *k* es el mayor valor de *i* que verifica que $e_i \geq 5$.

Se definen además los parámetros H_i y B_i que almacenan, respectivamente, el número de huecos y el número de bloques de longitud *i*. El valor estadístico a contrastar es:

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(H_i - e_i)^2}{e_i}$$

que sigue una distribución Chi cuadrado con $2k - 2$ grados de libertad ($l = 2k - 2$).

Los tests de series, rachas y póker se basan en el segundo postulado de Golomb.
5. Test de Autocorrelación. El propósito de este test es analizar la correlación entre la secuencia *s* y la secuencia *s'* obtenida de desplazar sin rotación la secuencia *s* un número determinado de posiciones (*d*) a derecha o a izquierda.

El número de bits de *s* que no coinciden con la correspondiente secuencia desplazada *d* posiciones se denota por $A(d)$ y se calcula por la siguiente expresión

$$A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d} \text{ , } 1 \leq d \leq \lfloor n/2 \rfloor,$$

donde $\oplus$ es la operación "or exclusiva". El valor estadístico a contratar es:

$$X_5 = \frac{2 \cdot (A(d) - \frac{n-d}{2})}{\sqrt{n-d}}$$

que sigue una distribución normal $N(0,1)$.

Para valores pequeños de $A(d)$ se recomienda repetir la prueba para diferentes desplazamientos. Este test se basa en el tercer postulado de Golomb.

2.3. Tests estadístico universal de Maurer

Otro de los tests ampliamente recomendado [12] para analizar la aleatoriedad de una secuencia de bits, es el conocido como test universal de Maurer [11], que permite detectar algunas deficiencias en la supuesta aleatoriedad de un generador.

Para realizar el test de Maurer hay que calcular el valor estadístico para la secuencia de salida *s* . Se escoge un valor para un parámetro *L* dentro del intervalo [6,16]. La secuencia se divide en bloques no superpuestos de *L* bits. El número total de bloques necesario es igual a $Q + K$, donde el mínimo valor para *Q* recomendado es $2^L \cdot 10$ y el mínimo valor de *K* recomendado es $2^L \cdot 1000$. En total, para el caso $L = 16$, hay que descomponer la serie en $2^L \cdot 1010$ (es decir, 66.191.360) bloques de 16 bits cada uno, lo que supone un total de 1.059.061.760 bits. La principal desventaja del test de Maurer es, por tanto, la enorme cantidad de bits (más de mil millones) que se requieren para poder someter el generador a estudio. Por eso, este test queda habitualmente reservado para los generadores de secuencias de bits pseudoaleatorios (PRBG, *PseudoRandom Bit Generator*).

El test obtiene un valor estadístico X_n que sigue una distribución Normal. A partir de la media esperada, de la desviación estándar calculada y del nivel de significación se obtienen los valores umbrales de aceptación o de rechazo k_1 y k_2. El valor estadístico debe quedar entre estos dos valores calculados. El proceso se realiza para diferentes valores del parámetro *L* (entre 6 y 16). El valor del nivel de significación α que se recomienda en [12] está entre 0,001 y 0,01. Para valores recomendados de *L*, al tomar el último extremo $L = 16$, se tiene que (tomando un valor de $\alpha = 0{,}005$) $k_2 - k_1 = 2 \cdot x_\alpha \cdot \sigma = 0{,}000853$. El valor estadístico calculado después de 65.536.000 sumas y logaritmos, debe caer en un intervalo de valores verdaderamente estrecho.

2.4. Estudio de la complejidad de las secuencias producidas

Una medida importante que indica la incapacidad para predecir los valores de una secuencia aleatoria es su complejidad. Para que una secuencia se pueda considerar impredecible deberá mostrar una elevada complejidad. Se podría decir que la complejidad de una secuencia finita es una medida de cuánto se parece dicha secuencia a una verdaderamente aleatoria.

Además, cualquier subsecuencia de la secuencia considerada debe mostrar también una complejidad elevada. Esta característica se estudia mediante los perfiles de complejidad.

Las definiciones de los diferentes tipos de complejidad son:
1. Complejidad de Turing–Kolmogorov–Chaitin [3]: Kolmogorov propuso utilizar la longitud del programa más corto capaz de generar la secuencia estudiada como la medida de su complejidad. A ese programa es al que se le llama máquina de Turing. No existe vía posible para el cómputo de esa complejidad. El interés de ésta es, por tanto, teórico.
2. Complejidad de Ziv–Lempel [3]: otra definición más reciente de complejidad fue introducida por Ziv y Lempel [8], y mide la velocidad con la que emergen nuevas cadenas de bits a medida que se recorre la secuen-

> No se dispone de ninguna prueba matemática que asegure de forma categórica la aleatoriedad de una secuencia de bits

cia. Un valor mínimo de esta complejidad rechaza la secuencia.

3. Complejidad Lineal: esta complejidad se define como la longitud del menor Registro de Desplazamiento Realimentado Linealmente (LFSR, *Linear Feedback Shift Register*) [6] capaz de generar la secuencia en estudio. Determina, por tanto, qué porción de subsecuencia es necesaria para lograr reproducir la totalidad de la secuencia. Una gran complejidad lineal implica siempre un gran periodo en la secuencia estudiada. Y cuanto mayor sea su periodo mayor será también la incapacidad para predecirla.

Existen secuencias con alta complejidad lineal pero no válidas como pseudoaleatorias: bastaría por ejemplo, una secuencia formada por todo ceros excepto un uno en el último lugar; una secuencia así presenta una alta complejidad lineal.

4. Perfil de la complejidad lineal: el estudio del perfil de la complejidad lineal resulta útil para detectar esas secuencias de buena complejidad lineal pero fácilmente predecibles.

Si $s = s_0,s_1,s_2,...s_{N-1}$ es una secuencia de N bits, la secuencia $L_1,L_2,...L_N$ formada por las longitudes del mínimo LFSR capaz de generar cada tramo de la secuencia s forma el perfil de la complejidad lineal. Si se representan los diferentes valores de las sucesivas complejidades lineales, fijando los puntos (N, L_N) en el plano N x L se obtiene la gráfica de su perfil. Esta gráfica, en el caso de una secuencia verdaderamente aleatoria, tiene la forma de una escalera creciente que sigue próxima y de forma regular a la recta L x $N/2$.

Para obtener la complejidad lineal de una secuencia se utiliza el algoritmo de Berlekamp–Massey, que permite determinar las características del mínimo LFSR capaz de generar la secuencia estudiada (ver [9] y [12]).

El perfil de la complejidad lineal de las secuencias es la medida de impredictibilidad más

VARIABLES:			`unsigned long t_1,t_2,rot;`
			`unsigned short aleat;`
			`unsigned char letra;`	

VALORES INICIALES:				
	`t_1 = times(NULL);`			
REPETIR MIENTRAS NO SE PULSE LA TECLA ESCAPE:				
	1.	Esperar pulsación de una tecla por parte del usuario. Cuando se produzca:		
		1.1.	letra igual al carácter introducido por teclado.	
			SI `letra = 27` IR A FIN (Tecla ESCAPE)	
		1.2.	`T_2 = times(NULL);`	
	2.	Ajustar intervalos de tiempo...		
		2.1.	`dift = t_2 - t_1;`	
		2.2.	`T_1 = t_2;`	
	3.	Primera rotación (a izquierda): sobre el valor `t_2`. La rotación depende del intervalo de tiempo transcurrido (`dift`) y del valor previo de la variable `aleat`.		
		3.1.	`dift *= dift;`	
		3.2.	`SI aleat DISTINTO DE 0, ENTONCES dift *= aleat;`	
		3.3.	`rot = dift % 31;`	
		3.4.	`t_2 *= t_2;`	
		3.5.	`t_2 = (t_2 >> rot | t_2 << (32 - rot));` (`t_2` variable de 32 bits).	
	4.	Primera operación XOR sobre el valor aleatorio a generar.		
		4.1.	`aleat ^= t_2;`	Se emplean los 16 bits menos significativos de la variable `t_2`. La variable `aleat` tendrá el valor final del proceso en la iteración anterior o un valor inicial cualquiera si estamos en la primera iteración.
	5.	Segunda rotación (a derecha): sobre el valor `aleat`. La rotación depende del valor ASCII de la tecla pulsada y del actual valor `aleat`.		
		5.1.	`SI aleat` DISTINTO DE `0, ENTONCES letra *= aleat;`	
		5.2.	`rot = (unsigned long)letra % 13;`	
		5.3.	`aleat = (aleat << rot | aleat >> (16 - rot))`	
	6.	Guardar el valor introducido en un vector de valores aleatorios.		
FIN				

Algoritmo 1. Pasos del generador aleatorio por entrada de teclado.

utilizada, debido fundamentalmente a que existe un algoritmo eficiente para calcularlo.

3. Diseño del generador aleatorio

En esta sección se describe el generador de secuencias de bits por entradas de teclado. Dicho generador toma sus valores a partir de las distintas entradas del teclado realizadas por el usuario y de los tiempos empleados entre las sucesivas pulsaciones de las teclas.
En dicho generador, el valor aleatorio *i*–ésimo generado depende de:

1. El valor del carácter *i*–ésimo introducido por teclado.

2. La diferencia de tiempo transcurrido entre la introducción del carácter (*i*-1)–ésimo y la introducción de carácter *i*–ésimo. Para el cálculo de esos tiempos hemos utilizado la función `times()` de la biblioteca `sys/times.h`. Esta función devuelve un valor proporcional al número de pulsos de reloj transcurridos desde un instante arbitrario del pasado. En Linux, si el parámetro de la función es el puntero `NULL`, este instante es el momento del último arranque del sistema. El número de pulsos por segundo que recoge la función viene señalado con el parámetro `_SC_CLK_TCK`.

3. El valor aleatorio previo generado.

Para la función mezcla recomendada en [10], se ha utilizado el operador a nivel de bit XOR y la aplicación, también a nivel de bit, rotación, definida mediante desplazamientos a izquierda y derecha y el operador OR a nivel de bit.

Los pasos que sigue el generador definido quedan recogidos en el **algoritmo 1**. Por cada pulsación de tecla quedan generados 16 bits de secuencia.

El generador definido es de sencilla implementación y los valores generados dependen de dos ocurrencias aleatorias como son el valor de las sucesivas teclas pulsadas por el usuario y el tiempo -- medido mediante la función `times()` -- transcurrido entre pulsación y pulsación. En cada pulsación de tecla se obtienen 32 bits de información procedentes del valor devuelto por la función `times(NULL)` y 8 bits que dependen del valor del carácter (su código ASCII) introducido por teclado.

El valor 31 que empleamos como divisor para obtener el módulo que nos indicará el número de desplazamientos de la rotación (paso 3.3.) está elegido como el mayor primo que es menor que el número de bits de una variable de tipo **`unsigned long`**. El valor 13 que empleamos como divisor en el paso 5.2. está elegido como el mayor primo que es menor que el número de bits de una variable de tipo **`unsigned short`**.

Al iniciar la ejecución del algoritmo se inicializa el valor de la variable `t_1`. Comienza entonces la generación de valores de la secuencia, en función de la tecla pulsada por el usuario (cuyo código ASCII queda recogido en la variable `letra`) y del instante en que se realice la pulsación de la tecla (valor recogido en la variable `t_2`). Se calcula entonces el intervalo de tiempo trascurrido desde la pulsación previa (o desde el inicio de la ejecución del algoritmo si estamos en la primera pulsación) y la última realizada. Finalmente se asigna el valor del tiempo tomado en la última pulsación a la variable `t_1`.

Comienza entonces el paso 3 del algoritmo, que realiza una rotación hacia la izquierda del valor almacenado en la variable `t_2`. Esta rotación va a depender del valor del intervalo de tiempo entre las dos últimas pulsaciones de teclado (almacenado en la variable `dift`) y del valor del último bloque de 32 bits generado (almacenado en la variable `aleat`): tantos bits como resulte de la operación `dift % 31`, donde el valor de la variable `dift` ha sufrido las modificaciones recogidas en los pasos 3.1. y 3.2. Para evitar la posibilidad de que se tuvieran valores de `t_2` siempre con ceros en los bits más significativos, antes de rotar su valor se eleva al cuadrado.

En el paso 4 se realiza la primera modificación sobre el valor de la variable `aleat`, que recogerá los próximos 32 bits generados: la operación XOR entre esta variable (que en este momento almacena la última subsecuencia de 32 bits generada) y la variable `t_2`, ya rotada a izquierda. Posteriormente, en el paso 5, se realiza la segunda modificación de la variable `aleat`, sometiéndola a una rotación hacia la derecha un número de bits que dependerá del valor del código ASCII de la última tecla pulsada.

Terminada esta operación, se almacenan los 32 bits en el vector donde estemos guardando la secuencia de bits generada. El proceso se repetirá hasta que el usuario pulse la tecla de escape (código ASCII 27).

Dado que la implementación de una función con este código no es muy compleja, no se incluirá aquí. Sí se recoge, en cambio, la implementación para Linux de la función que definida para obtener los valores ASCII de cada pulsación de tecla (función que hemos llamado `caracter()`):

```
char caracter()
{
   char a;
   initscr();
   nonl();
   leaveok(curscr,0);
   raw();

   fcntl(0,F_SETFL,O_RDONLY);
   while(!read(0,(char*)&a,1));

   noraw();
   nocrmode();
   nl();
   endwin();

   fcntl(0,F_SETFL,O_NDELAY   |
O_RDONLY);

   return(a); }
```

4. Funcionamiento del generador

A continuación se presentan algunos ejemplos que muestran el funcionamiento del generador descrito en la sección 3. Por razones de espacio, estos ejemplos utilizan entradas de teclado muy cortas y sólo pretenden ilustrar el funcionamiento del generador.

En la **tabla 1** se muestran los valores que se obtienen al introducir por teclado una secuencia cualquiera de pulsaciones de teclado: por ejemplo, la cadena de caracteres "*valores*".

En la **tabla 2** se muestran los valores, con la suposición de que en la ejecución siempre se introduce el mismo carácter: por ejemplo, una secuencia constante de letras '*A*'.

En la **tabla 3** están incluidos los valores obtenidos con la suposición de que en la ejecución siempre se introducen los diferentes caracteres con la misma cadencia de tiempos: por ejemplo un incremento de 6 en el valor que devolvería la función `times(NULL)` entre pulso de tecla y pulso de tecla. La secuencia de caracteres introducida vuelve a ser "*valores*".

Finalmente, en la **Tabla 4**, se presentan los valores que se obtendrían si siempre se introdujese el mismo carácter (por ejemplo, una vez más, el carácter '*A*'), y siempre con la misma cadencia (una vez más un incremento de 6 por cada pulsación). En este caso la secuencia generada no será necesariamente siempre la misma, pues la salida depende también del valor inicial de la variable `aleat` y del valor de `t_1` en el momento de empezar la ejecución del proceso.

En la siguiente sección se presenta un estudio estadístico sobre la calidad de la aleatoriedad para cada una de estas cuatro formas de obtener las secuencias.

5. Análisis de los resultados

En esta sección se presenta el análisis realizado sobre las secuencias producidas por el generador en diferentes supuestos. Hemos realizado el análisis sobre las cuatro formas

aleat inicial					2AC3
t_1 inicial					69.899.601
1.1.	1.2. t_2	2.1. dift	3.3. rot	5.2. rot	5.3. aleat
v	69.899.619	18	25	8	7A0E
a	69.899.641	22	24	2	2C91
l	69.899.653	12	20	1	A8FD
o	69.899.670	17	5	9	C4AE
r	69.899.685	15	17	4	8600
e	69.899.702	17	25	10	4CD0
s	69.899.720	18	16	1	E325

Tabla 1. Entrada por teclado aleatoria en cadencia de pulsaciones aleatoria.

aleat inicial					21E1
t_1 inicial					69.914.454
1.1.	1.2. t_2	2.1. dift	3.3. rot	5.2. rot	5.3. aleat
A	69.914.466	12	26	3	F08E
A	69.914.489	23	1	1	062D
A	69.914.498	9	0	1	F052
A	69.914.513	15	20	0	E4EF
A	69.914.529	16	12	12	3F77
A	69.914.547	18	11	6	2A76
A	69.914.565	18	1	6	2E8F

Tabla 2. Entrada por teclado fija en cadencia de pulsaciones aleatoria.

diferentes de uso del generador:

Medición 1. Uso estándar: cadena arbitraria con cadencia variable. La cadena introducida ha sido "Algoritmo de BERLEKAMP–MASSEY para determinar el perfil de la complejidad lineal del generador".

Medición 2. Uso de un único carácter con cadencia variable. El carácter empleado como entrada en este caso ha sido la letra 'A'.

Medición 3. Uso de una cadena arbitraria con cadencia constante. En este caso se ha considerado una cadencia `dift` constante igual a 6: `t_2 = t_1 + 6`.

Medición 4. Uso de un único carácter con cadencia constante. El carácter utilizado ha vuelto a ser la letra 'A' y el ritmo de cadencia constante de nuevo ha sido igual a 6.

5.1. Tests estadísticos básicos.

En la **tabla 5** se muestran los resultados de los cinco primeros tests estadísticos presentados en la subsección 2.2. para las "Mediciones" que se acaban de señalar.

Para las secuencias generadas mediante pulsaciones de teclado de cadencia de pulsación aleatoria, los valores que presentamos son las medianas de los obtenidos en 20 secuencias. Para los otros dos casos, en que la cadencia es constante, y la entrada de teclado o es una frase establecida o es una secuencia de caracteres siempre iguales hemos preferido mostrar un caso concreto y no calcular medianas de diferentes casos de teclas y de cadencias (hemos tomado la cadencia igual a 6 y para el cuarto caso la tecla pulsada 'A' mayúscula). En las cuatro mediciones, el valor de los grados de libertad para X_3 es 8, y para X_4 es 31. Como se puede observar, el generador mantiene sus valores por debajo de los límites de los tests estadísticos aún en el caso más desfavorable en que la secuencia de entrada se realiza mediante una misma entrada de teclado en una cadencia constante.

Hemos hecho otro análisis, con los mismos tests, centrando el estudio en un empleo ordinario del generador: secuencias obtenidas por pulsaciones aleatorias de teclado en una cadencia irregular. Hemos probado con diferentes secuencias de diferentes longitudes de bits.

En la **tabla 6** se incluyen los estadísticos de contraste, X_i , de cada uno de los cinco primeros tests para diferentes longitudes de la se-cuencia, así como sus grados de libertad, l_i , cuando son diferentes de 1 (test de frecuencias) o de 2 (test de series). Los valores en cada columna son las medianas de los valores X_i sobre un muestreo de 40 secuencias en cada uno de los tamaños. Todos los valores superan las exigencias de los tests estadísticos, tanto si $\alpha = 0{,}05$ como incluso para $\alpha = 0{,}1$.

aleat inicial					8254
t_1 inicial					69.927.638
1.1.	1.2. t_2	2.1. dift	3.3. rot	5.2. rot	5.3. aleat
v	69.927.644	6	9	4	A427
a	69.927.650	6	28	7	33AE
l	69.927.656	6	27	11	4BDD
o	69.927.662	6	13	8	3914
r	69.927.668	6	24	12	AC68
e	69.927.674	6	22	12	8532
s	69.927.680	6	21	5	A450

Tabla 3. Entrada por teclado aleatoria en cadencia de pulsaciones fija.

aleat inicial					F49B
t_1 inicial					69.935.139
1.1.	1.2. t_2	2.1. dift	3.3. rot	5.2. rot	5.3. aleat
A	69.935.145	6	26	3	86EA
A	69.935.151	6	20	10	5E49
A	69.935.157	6	2	8	F78C
A	69.935.163	6	9	2	1FA8
A	69.935.169	6	3	9	71AB
A	69.935.175	6	12	1	9E2C
A	69.935.181	6	30	0	C289

Tabla 4. Entrada por teclado fija en cadencia de pulsaciones fija.

	X_1	X_2	X_3	X_4	X_5
medición 1	0,3021	1,5849	28,5000	8,6665	0,2272
medición 2	0,2042	0,7209	23,8333	9,3400	0,4220
medición 3	0,2042	1,0084	23,1667	8,3557	0,3571
medición 4	1,3500	2,3820	33,8333	12,0554	0,1623
x_α	3,8410	3,8410	44,9850	15,5070	1,2860

Tabla 5. Estadísticos de los 5 primeros tests para los diferentes tipos de mediciones, con $\alpha = 0{,}05$.

6.2. Test universal de Maurer

Para proceder a este estudio se debe diseñar una forma de utilizar el generador que no requiera la intervención de ningún usuario puesto que para generar la cantidad de bits necesaria se requeriría que un usuario realizara 66.191.360 pulsaciones de teclado, lo que supondría un trabajo prolongado durante algo más de 76 días, realizado por una persona que pudiese mantener un ritmo constante de diez pulsaciones por segundo, sin interrupción. Como además es necesario someter a test un número amplio de series, el trabajo para someter nuestro generador al test de Maurer se convierte en imposible.

Hemos diseñado un procedimiento para poder testear el generador con el test universal de Maurer. Este procedimiento consiste en emplear nuestro generador como si fuese un generador de bits pseudoaleatorio (PRBG), de forma que las entradas por teclado estén prefijadas y los valores de tiempo transcurrido entre una pulsación y la siguiente también: es decir, la entrada de teclado será constante así como la cadencia de pulsación.

Como se puede apreciar en los sucesivos pasos del generador propuesto (ver **algoritmo 1**), hay dos valores iniciales que no dependen de las entradas del teclado, ni de su cadencia: el valor inicial de la variable `aleat` (variable de tipo **`unsigned short`**, de 16 bits) y el primer valor que toma la variable `t_1` (variable de tipo **`unsigned long`**, de 32 bits). Los dos valores que hemos mencionado y que dejamos prefijados son el de la variable `dift` (variable tipo **`unsigned long`**, de 32 bits) y el de la variable `letra` (variable tipo **`char`** de 8 bits).

En total son 88 bits que podemos considerar como una semilla inicial de nuestro generador, convertido ahora en generador determinista. El generador así empleado tarda menos de 20 segundos en presentar al test de Maurer la secuencia de más de mil millones de bits.

Hemos ejecutado 50 veces el test de Maurer sobre 50 secuencias de 1.059.061.760 bits, generadas a partir de 50 'semillas' iniciales. Los valores de las medianas de cada X_u para cada uno de los 11 valores de L estudiados quedan recogidos en la **tabla 7**. En esta tabla, además de los valores de las medianas, recoge los extremos k_1 y k_2 que le corresponden, calculados con $\alpha = 0{,}005$ (ver [12]).

En cuatro de los once casos (L = 6, 7, 9, 10) los valores de X_u han caído dentro de intervalo. En dos casos (L = 8, 16) el valor está por debajo del mínimo en cinco centésimas y en una centésima, respectivamente. En cinco casos (L = 11, 12, 13, 14, 15) el valor está por encima del máximo en valores de entre una centésima y tres milésimas.

6.3. Estudio de la complejidad de las secuencias

Una medida baja en la complejidad de Ziv–Lempel ofrece la misma información que se revela con el estudio del perfil de la complejidad lineal de la secuencia [3, capítulo 6]. Además, los resultados de la entropía basada en la complejidad de Ziv-Lempel son irrelevantes porque todos los valores son muy similares y cercanos a la unidad. Por esta razón, esta medida de la complejidad no suele resultar útil para remarcar las diferencias entre los generadores [1]. Hemos centrado por tanto nuestro estudio en ese perfil de complejidad, utilizado en algoritmo de Berlekamp–Massey.

En los cuatro casos antes señalados (Me-

	X_1	X_2	X_3	l_3	X_4	l_4	X_5
96 bits	0,521	1,577	1,750	3	3,875	2	0,3254
x_α	3,841	3,841	7,815		5,991		1,6449
160 bits	0,400	1,811	6,925	7	4,894	4	0,0000
x_α	3,841	3,841	14,067		9,488		1,6449
320 bits	0,256	1,518	11,400	15	5,025	4	0,3413
x_α	3,841	3,841	24,996		9,488		1,6449
480 bits	0,613	1,517	12,200	15	5,011	4	0,2276
x_α	3,841	3,841	24,996		9,488		1,6449
640 bits	0,531	1,409	10,400	15	4,230	4	0,2844
x_α	3,841	3,841	24,996		9,488		1,6449

Tabla 6. Estadísticos de los 5 primeros tests para diferentes longitudes, con $\alpha = 0{,}05$.

L	k_1	X_n	k_2
6	5,206854	5,219121179	5,228556
7	6,188070	6,200019001	6,204432
8	7,177613	7,121962139	7,189719
9	8,172000	8,179261904	8,180850
10	9,169116	9,174770197	9,175533
11	10,167718	10,17270238	10,172346
12	11,167102	11,17094146	11,170428
13	12,166879	12,1708851	12,169261
14	13,166841	13,17018484	13,168545
15	14,166880	14,170022	14,168096
16	15,166945	15,15182785	15,167813

Tabla 7. Resultados para el test de Maurer.

diciones 1 a 4), el perfil de complejidad lineal se comporta de tal manera que la recta de tendencia de la gráfica (N, L_N) en el plano N x L tiene la forma $y = 0{,}5 \cdot x + c_i$, donde c_i vale, en cada uno de los cuatro casos indicados, lo siguiente:

$c_1 = 0{,}2478$ $c_2 = 0{,}2484$ $c_3 = 0{,}2483$ $c_4 = 0{,}2478$

Como cabía esperar, para un generador que tiene un comportamiento aleatorio, la gráfica del perfil de la complejidad lineal sigue siempre próxima a la recta $L = N/2$.

6. Conclusiones

El generador de bits que se ha propuesto en el presente trabajo y que está basado en entradas por teclado, ha obtenido un perfil de complejidad lineal muy próximo a la recta $L = N/2$ y ha superado los cinco primeros tests estadísticos analizados. Consideramos, por tanto, que dicho algoritmo es válido para generar secuencias que se pueden tomar como realmente aleatorias.

En un segundo estudio, hemos analizado el generador en un comportamiento determinista, en el que no ha intervenido ningún factor externo (pulsación aleatoria de teclado y cadencia aleatoria de pulsación) de modo que toda la secuencia generada ha dependido únicamente de un valor inicial bastante reducido: 88 bits. Hemos analizado las secuencias obtenidas mediante este procedimiento con el test universal de Maurer. Los resultados obtenidos mediante ese test son aparentemente buenos. Pero sólo aparentemente, puesto que la semilla inicial aporta muy poca entropía a la serie total (de cada 88 bits iniciales generamos más de mil millones de bits).

Ya hemos mencionado que el test de Maurer está diseñado para los generadores pseudoaleatorios pero puesto que disponíamos de un medio de obtener una secuencia suficientemente larga, hemos querido analizar el comportamiento estadístico del generador propuesto cuando no hay un agente externo que introduzca incertidumbre o aleatoriedad. El resultado, como se ha visto, ha sido que el algoritmo logra superar el test de Maurer en varios de los valores de L; y se queda a una distancia de milésimas en los valores restantes. No se puede afirmar que el generador ha superado las pruebas estadísticas de este test, cosa que tampoco se pretendía, pues como ya se ha señalado, el origen de cada secuencia analizada arranca de una semilla que se expande... ¡22.625.410 de veces!

Sin embargo, hemos querido recoger el estudio por dos razones:

1. Los resultados ofrecidos por el test de Maurer nos han parecido realmente buenos, de hecho mejores de los que cabría esperar inicialmente.
2. La velocidad de generación de una secuencia de 1.059 millones de bits nos ha parecido elevada. Especialmente si la comparamos con otros generadores de reconocido valor criptográfico, como el BBS [2], que requiere unos tiempos mucho más dilatados para esas longitudes. En un ordenador PC actual cada secuencia de esa cantidad de bits ha sido generada en unos pocos segundos.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el proyecto PITÁGORAS de la Fundación SENECA (Comunidad Autónoma de Murcia) con referencia PB/13/FS/99.

Referencias

[1] J. Areitio Bertolín. "Evaluación de generadores de secuencias pseudoaleatorios utilizadas en telemática e ingeniería criptográfica". <http://www.conectronica.com/articulos/cripto34.htm>.

[2] L. Blum, M. Blum, M. Shub. "A simple unpredictable Pseudo-Random Number Generator". *SIAM Journal of Computing*, Vol. 15, Nº 2, pp. 364–381, 1986.

[3] Ernesto Cruselles Fomer, José Luis Melús Moreno. *Secuencias pseudoaleatorias para telecomunicaciones*. Ediciones UPC, 1996.

[4] "Electronic Signatures and Infrastructures (ESI); Algorithms and Parameters for Secure Electronic Signatures". European Telecomunications Standars Institute 2003. ETSI SR 002 176 v1. 1. 1. (2003–03). Special Report. <http://webapp.etsi.org/action%5CPU/20030401/sr_002176v010101p.pdf>.

[5] Jean Campbell, Randall J. Easter, Annabelle Lee, Ray Snouffer. NIST, "Annex C: Approved Random Number Generators for FIPS PUB 140–2, Security Requirements for Cryptographic Modules". FIPS PUB 140 - 2. Annex C, Draft. March 17, 2003. <http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexc.pdf>.

[6] A. Fuster Sabater, D. de la Guía Martínez, L. Hernández Encinas, F. Montoya Vitini, J. Muñoz Masqué. *Técnicas criptográficas de protección de datos*. RA–MA, 3ª ed, Madrid 2004.

[7] S. W. Golomb. *Shift Register Sequences*. Prentice Hall Inc., Holden–Day, San Francisco, 1967.

[8] Abraham Lempel and Jacob Ziv. "On the complexity of Finite Sequences". *IEEE Transactions on Information Theory*, Vol. 22, Nº 1, January 1976, pp. 75–81.

[9] James L. Massey. "Cryptography: Fundamentals and Applications (Copies of Transparencies)". Apuntes de curso impartido en Advanced Technology Seminars (Zürich, Suiza), 1994.

[10] Tim Mathews. "Suggestions for Random Number Generation in Software". Bulletin nº 1. *News and advice from RSA Laboratories*. 22, January, 1996.

[11] Ueli M. Maurer. "A Universal Statistical Test for Random Bits Generators". *Journal of Cryptography*, Vol 5, nº 2. 1992. <ftp://ftp.inf.ethz.ch/pub/crypto/publications/maurer92a.pdf>.

[12] A. Menezes, P. Van Oorschot, S. Vanstone. *Hand Book of Applied Cryptography*, CRC Press, 1997.