Un método para la generación de cuadrados latinos de orden 256

Gallego Sagastume, Ignacio
Facultad de Informática, Universidad Nacional de La Plata

Abstract
Los quasigroups son estructuras algebraicas con aplicaciones en seguridad informática, en particular en criptografía. Sus tablas de multiplicación son matrices de dos dimensiones, llamadas "latin squares" (LSs) o "cuadrados latinos". Si los LSs son aleatorios, pueden ser usados como generadores de claves para algoritmos de encriptación. En el contexto de un protocolo de comunicación seguro, debe generarse un nuevo LS cada cierta cantidad de tiempo o cantidad de datos transmitida. El tiempo y recursos requeridos para generar nuevos LSs no deben implicar una gran sobrecarga en la transmisión.
En este trabajo, se analiza el tiempo y la complejidad de un algoritmo para generar LSs, junto con una solución práctica para generar LSs aleatorios de orden 256 (usar una operación de producto entre dos LSs de orden 16), con una distribución aproximadamente uniforme. Se presenta el pseudocódigo de los scripts Python y un análisis de cuán uniformes son los LSs generados.

Palabras Clave
Quasigroups, latin squares, cuadrados latinos, ls, lss, aleatorios, distribución uniforme, producto.

Introducción

Los algoritmos de encriptación de datos usan claves secretas para convertir una entrada (texto plano) o un bloque de datos (dado en caracteres, bytes o bits) en un texto cifrado. Si el algoritmo es simétrico, la misma clave se usa para encriptar y desencriptar un mensaje en cada extremo de la comunicación. Estas clases de algoritmos se usan para transmitir datos en forma segura en el contexto de un protocolo de intercambio de información sobre Internet que se propone diseñar [13][14]. El mejor algoritmo de encriptación es aquel que pueda encriptar de tal forma de que no se revele información sobre el texto plano o la clave. Un algoritmo con esta característica es OTP ("One Time Pad", [15]), pues el tamaño de la clave es igual que el del texto plano y es imposible deducir que clave se utilizó para encriptar, ya que existen muchas potenciales claves válidas que llevan a distintos textos planos válidos.

Es posible utilizar cuadrados latinos o latin squares [10][12] (LSs de ahora en más) aleatorios como claves o como generadores de claves para un OTP en nuestro protocolo criptográfico. Por ejemplo, los LSs podrían intercambiarse en el "handshake" (como si fueran claves) y luego utilizarse para generar claves aleatorias para usar con el OTP. Cada vez que se usa completamente la clave generada o se agotan las posibilidades del LS actual, debe generarse un LS nuevo. Se presenta entonces el problema de cómo generar un LS aleatorio en forma eficiente.

Un LS es la tabla de multiplicación de un quasigroup (QG), que es una estructura algebraica conocida en criptografía [6]. Los QGs pueden cumplir con ciertas propiedades como la no-asociatividad, no-conmutatividad, no-idempotencia y el no tener unidad a derecha o izquierda (ver [3] para más detalles). Si cumple con todas estas "no" propiedades, el QG se dice "shapeless" o "sin forma", y es útil para usar en criptografía. A continuación se formaliza una definición de LS:

Definición 1. *Un latin square (LS) de orden n es una matriz de n×n que se completa con n símbolos diferentes, donde cada símbolo aparece exactamente una vez en cada fila y en cada columna.*

Los símbolos utilizados pueden ser números de 1 a n, o letras, o inclusive colores (ver [10]). Por ejemplo, el siguiente es un LS de orden 3:

123
231
312

Los símbolos pueden estar aleatoria o pseudoaleatoriamente distribuidos en el LS. Este hecho puede ser usado para generar claves o texto cifrado como se muestra en [2] (en la sección siguiente se mostrará cómo usar un LS para encriptar un texto plano de ejemplo).

Otra de las propiedades que tienen los LSs es que el número *L(n)* de diferentes LSs que existen de orden *n* es tan grande (de hecho el número *L(12)* no se conoce ni puede ser computable con el poder computacional actual, ver [7]), que usar un LS de orden suficientemente grande hace que sea prácticamente imposible adivinarlo por "fuerza bruta". Para tomar dimensión del tamaño de *L(n)* y lo complejo del problema, pueden mencionarse dos cotas conocidas (ver trabajo de Koscielny [6] o Jacobson y Matthews [4]):

$$\frac{n!^{2n}}{n^{n^2}} \leq L(n) \leq \prod_{k=1}^{n} k!^{\frac{n}{k}}$$

Fórmula 1: cotas superior e inferior para el número de diferentes LS de orden n

Por ejemplo, esta cota inferior[1] de *L(n)* para $n = 256$ es $\approx 3{,}047 \times 10^{101723}$.

En nuestro protocolo seguro de transmisión de datos, se utilizarán LSs aleatorios como generadores de claves para encriptar todos los datos que se transfieran. Se debe generar un nuevo LS cada una determinada cantidad de bytes transmitidos o, al menos, una vez por conexión, para no utilizar siempre el mismo y correr el riesgo de que un atacante potencial lo adivine o deduzca. Más precisamente, se utilizará un LS como generador de una clave de tamaño igual que el mensaje a transmitir, para luego utilizarla con OTP, cuya clave deberá usarse una sola vez y luego descartarse. En las siguientes secciones, se estudiará cuán costoso en términos de tiempo computacional es generar LSs aleatorios. Se utilizó el lenguaje Python por ser éste un lenguaje moderno, práctico y con algunas características interesantes, como las operaciones sobre listas y conjuntos. El principal objetivo del trabajo es generar en una fracción de segundo LSs aleatorios de orden 256 con distribución aproximadamente uniforme.

[1] Todas las estimaciones de este trabajo fueron calculadas usando Wolfram (ver [1])

¿Cómo se usa un LS para encriptar o generar claves?

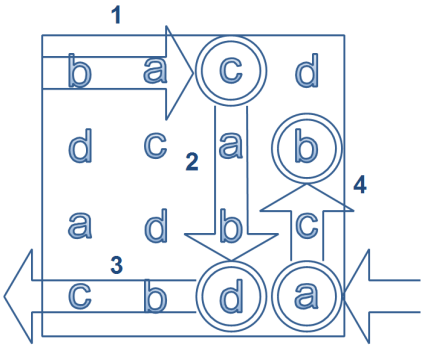

Figura 1: encriptación de la cadena "abcc" a la manera de Gibson

Un LS de orden 256 puede almacenar todos los caracteres de la tabla ASCII y así encriptar casi cualquier texto plano. Nuestro interés es tener LSs de orden 256 para estos propósitos. Supóngase (para simplificar) que se tiene el LS de orden 4 de la **figura 1** y se quiere utilizarlo para encriptar la palabra "abcc". Se puede recorrer el LS, a la manera de Gibson [2], alternando movimientos hacia derecha, abajo, izquierda y arriba. Cuando se termina de recorrer en una dirección y se alcanza el borde del LS, se ingresa cíclicamente por el otro lado.

1. **Primer paso**: se busca la primera letra a encriptar, la "a", comenzando por la primera fila de izquierda a derecha hasta encontrar esa letra. Se toma la letra siguiente en el LS, la letra "c".
2. **Segundo paso**: Ahora se cambia la dirección y se va hacia abajo. Se busca la segunda letra de la palabra a encriptar, la "b" y se toma la letra siguiente, la "d". Se tiene hasta ahora el texto cifrado "cd".
3. **Tercer paso**: Se cambia nuevamente la dirección para ir hacia la izquierda. Se encuentra la letra "c" y se toma la siguiente en la misma dirección, ingresando al LS cíclicamente por el otro lado. Se toma la

siguiente letra del LS, la “a”. Se tiene hasta ahora “cda”.

4. **Cuarto paso**: Se cambia otra vez la dirección y se va hacia arriba. Se busca la última letra del texto plano, la otra “c”, y se toma la letra siguiente, la “b”.

El texto cifrado correspondiente al texto plano “abcc”, resulta entonces “cdab”. Notar que este proceso es reversible, y que recorriendo en el orden inverso (sabiendo la última posición utilizada), se puede volver a obtener el texto plano “abcc”.

Otras variantes de recorrido podrían ser alternar entre ir hacia izquierda, abajo, derecha y arriba, o recorrer en una dirección y rebotar contra los bordes del LS (en vez de reingresar cíclicamente). También se puede utilizar el mismo esquema de recorrido pero tomando los dos caracteres siguientes en vez de uno. En este caso, el texto cifrado tendría el doble de longitud del texto plano, pero se estaría revelando más información del LS y se debería generar más frecuentemente.

Generar claves aleatorias de esta manera parece razonable: se aprovecha el hecho de que el LS es aleatorio (dos símbolos contiguos no respetan ningún orden) y se toman los símbolos según algún recorrido secuencial. El problema es que el LS usado para encriptar debe ser lo más aleatorio posible (sin tendencias o valores similares en celdas adyacentes) y debe generarse uno cada cierto tiempo para que no pueda ser deducido por un atacante, asumiendo que éste conozca el esquema de recorrido que se utiliza.

El algoritmo recursivo base para la generación

Se presenta aquí un algoritmo base, que servirá para generar LSs de orden pequeño, como por ejemplo de orden 16 o 20. Para generar LSs de orden 256, se utilizarán dos LSs de orden 16 como punto de partida y luego se aplicará una operación de multiplicación sobre ellos (operación propuesta por Koscielny en [6]).

Este algoritmo es el más simple posible: se parece a la forma en que se generaría un LS usando papel, lápiz y un dado. Se van extrayendo números aleatorios, poniéndolos de izquierda a derecha, y se va completando el LS por filas. En caso de quedarse sin alternativas, el algoritmo borra elementos previos y vuelve a intentar con otras combinaciones de elementos. Se sabe que el algoritmo siempre termina, pues se conoce el siguiente teorema:

Definición 2. *Un rectángulo latino de $k \times n$ es un cuadrado latino parcial, en el cual las primeras k filas están completas y las restantes $n-k$ filas están completamente vacías, para algún valor k, siendo $k < n$.*

Teorema 1. *Todo rectángulo latino puede completarse hasta obtener un cuadrado latino (LS).*

Demostración. Se prueba utilizando el teorema “del casamiento” de Hall (ver [9]). □

Por cada posición, el algoritmo extraerá un símbolo aleatoriamente del conjunto resultante de restar los que tiene disponibles en la fila que se está generando, menos el conjunto los que ya se usaron en la columna actual, menos los elementos que ya han sido probados en esa posición y llevan a colisiones. Es decir, se toma aleatoriamente del conjunto:

Disp. (i,j) = { disponibles en fila i } -
{ usados en columna j } -
{ provocan colisión en (i,j)}

Si este conjunto de elementos disponibles es vacío, hay “colisión”, es decir, es un punto en el algoritmo en que no hay más posibilidades de símbolos para la posición actual, pues los únicos símbolos disponibles en la fila ya se han usado en la columna actual, o ya se han probado todas los símbolos posibles y todos ellos llevan a colisiones. En este punto, el algoritmo tiene que borrar el último elemento generado en la fila para probar con otro, a fin de terminar de generar esa fila, y además debe

anotar el último símbolo generado como "prohibido" para esa posición, dado que lleva a una colisión. Una colisión se da por ejemplo cuando se ha generado:

1 2 3 4
4 3 1 2
3 1 2

y el único símbolo disponible para la fila (el 4), ya existe en la última columna. En ese caso, el algoritmo debe hacer "backtracking" (volver hacia atrás) la generación del LS, borrando el último símbolo de la fila, el 2, y anotar el símbolo 2 como prohibido en la posición (3,3), dado que lleva a una colisión. Entonces el algoritmo puede continuar eligiendo el último símbolo disponible (el 4) en la posición (3,3):

1 2 3 4
4 3 1 2
3 1 4

generando una nueva colisión, ya que el único símbolo disponible en la fila, el 2, ya existe en la última columna. Se anotan como prohibidos en la posición (3,3) los símbolos [2,4]. Como no se tienen más símbolos para probar en la posición (3,3), el algoritmo debe hacer backtracking otra vez, borrando el símbolo 1 de la última fila y eliminando la lista de símbolos que llevan a colisión en la posición (3,3), ya que estos dependían de la elección del símbolo 1 en la posición (3,2). Ahora se anota el símbolo 1 como prohibido en la posición (3,2) y se prueba con el último símbolo posible, el 4:

1 2 3 4
4 3 1 2
3 4

permitiendo así generar el resto de la fila:

1 2 3 4
4 3 1 2
3 4 2 1

y terminar la generación del LS, sin colisiones en la última fila, con:

1 2 3 4
4 3 1 2
3 4 2 1
2 1 4 3

El pseudocódigo del algoritmo recursivo base es el siguiente:

```
def genLSRec(n, ls, i):
       if (i==n):
              #si ya generé n, terminé
       #genero fila
       ls[i]= generateRow(i, ls)
       #invocación recursiva
       genLSRec(n, ls, i+1)

def generateRow(i_row, ls):
       #genero fila de tamaño n en la
       #posición i

       while (i_col < n):
         #disponibles es:
         disp = (dispCol[i_col] - dispRow)
                 - prohib[i_col]

         #si me quedan disponibles
              #saco un símbolo al azar:
              sym = random.choice(disp)

              #contabilizo el elegido:
              dispCol[i_col].remove(sym)
              i_col = i_col + 1
              dispRow.remove(sym)
              row.append(sym)
         else:
              #hay colisión
              #limpio los prohibidos de las
              #columnas de más a la derecha
              #hago backtracking
              i_col = i_col - 1
              #saco el último simbolo de la
              # fila actual
              sym = row.pop()
              #guardo el símbolo prohibido
              prohib[i_col].append(sym)
              #marco disponible el símbolo
              dispRow.append(sym)
              dispCol[i_col].append(sym)
       return row
```

Pseudocódigo 1: algoritmo recursivo base para generación de LSs de orden 16

Aquí, el símbolo "-" es la resta de conjuntos. La función *choice()* elige un elemento al azar del conjunto que se pasa como parámetro. La función *append()* agrega al final de la lista y *pop()* extrae el último elemento de una lista.

Cantidad de colisiones en el peor caso

Para comenzar se demuestra el siguiente teorema, observando dónde se ubican las colisiones al generar el cuadrado (ver **figura 2**):

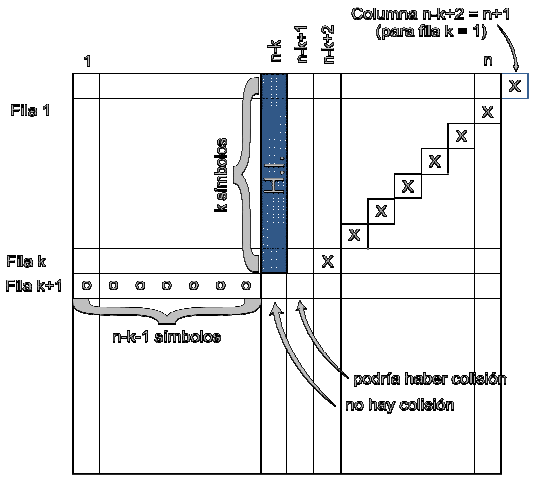

Figura 2: ubicación de las colisiones

Teorema 2. *Teorema de la diagonal. En la generación de la fila i sólo puede haber colisiones a partir de la columna n-i+2 (y hasta la columna n).*

Demostración. Se prueba por inducción sobre el número de fila. Para el caso base p(1) se observa que no puede haber colisiones, pues siempre hay símbolos disponibles para elegir en cada columna. Entonces se cumple, pues sólo puede haber colisiones a partir de la columna n-1+2=n+1, que cae fuera del cuadrado. Se supone p(k): sólo puede haber colisiones a partir de la columna n-k+2. Debe probarse p(k+1): sólo puede haber colisiones a partir de la columna n-(k+1)+2= n-k+1. Supóngase que en la fila k+1 ya se han colocado n-k-1 elementos y se va a colocar el de la columna n-k. Por hipótesis inductiva, en la columna n-k no hubo colisiones durante la generación de la fila k, y en esa columna hay k elementos distintos colocados. Supóngase en el peor caso que los colocados en la fila k+1 son n-k-1 elementos distintos a los k elementos colocados en la columna n-k. Se tendrían utilizados (n-k-1)+k = n-1 elementos distintos. En este caso no habría colisión en la posición (k+1, n-k), por no haberse utilizado los n símbolos, pero sí podría haber colisión en la posición (k+1, n-k+1).□

Se sabe que en la primera y última fila del LS no ocurrirán colisiones: la primera por no tener ninguna restricción previa y la última por estar completamente determinada por el rectángulo latino de *n-1* filas × *n* columnas.

Si se toma en cuenta además el **teorema 2**, que expresa que las colisiones se pueden producir en el triángulo debajo la diagonal que va desde las posiciones *(n,1)* a *(1,n)* (exceptuando la primera y última filas), se puede tomar el peor caso (o una cota superior del mismo), es decir, que se tenga una colisión por cada símbolo colocado, sabiendo que ya se han colocado *n-i+1* símbolos en la fila *i*. La cantidad de colisiones para la fila *i* sería como sigue: para la columna *n-i+2* se tendrían *n-(n-i+1) = i-1* símbolos disponibles en la fila, que llevarían a *i-1* colisiones. En la posición *n-i+3* se tendrían *i-2* símbolos disponibles (porque ya se ha colocado uno en la posición anterior), que llevarían a *i-2* colisiones, y así sucesivamente, con lo cual se obtendría la siguiente fórmula considerando todas las filas del LS:

$$\sum_{i=2}^{n-1} (i-1)!$$

Fórmula 2: cota superior de la cantidad de colisiones para el peor caso

Esto implica que para la generación de un LS de orden 16 habría, en el peor caso, 93.928.268.313 colisiones en total. Para orden 256, el problema ya es más grande, porque puede haber del orden de 1×10^{502} en el peor caso y el algoritmo puede tardar varios miles de años en terminar. El número de colisiones crece exponencialmente, como se muestra en la siguiente tabla:

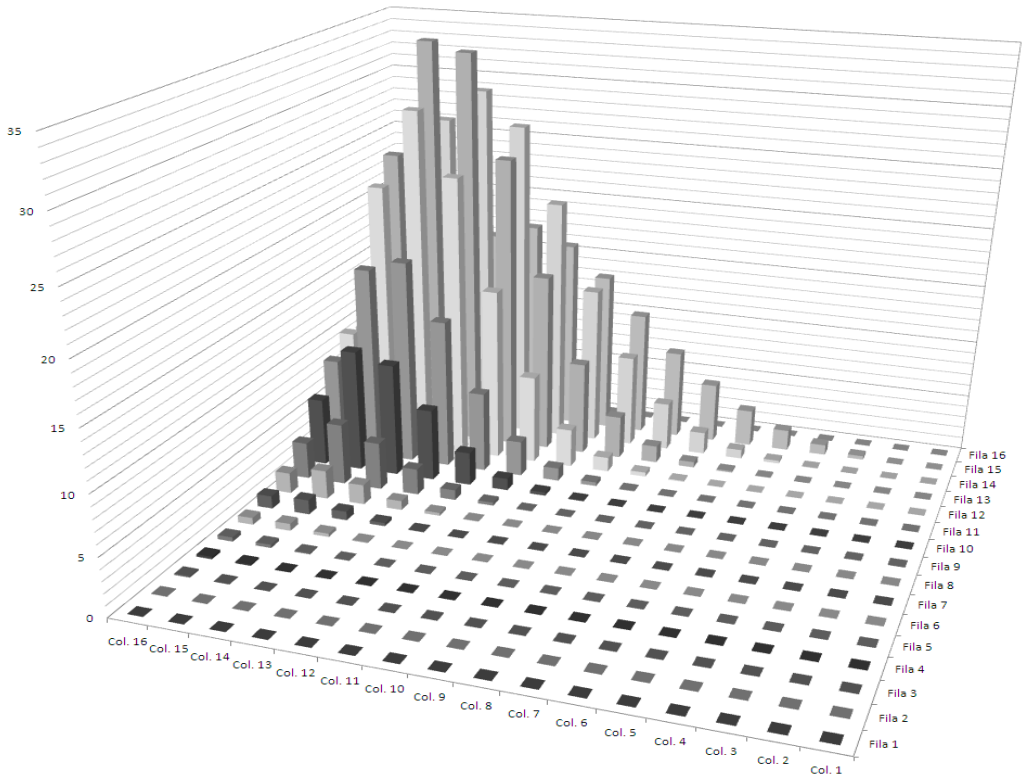

Figura 3: promedio de colisiones en 10.000 corridas del algoritmo

Orden	Cantidad de colisiones (peor caso)
4	3
5	9
6	33
8	873
16	93.928.268.313
20	6.780.385.526.348.313
32	$\approx 274{,}410 \times 10^{30}$
64	$\approx 3{,}1986015536 \times 10^{85}$
128	$\approx 2{,}3911518534 \times 10^{211}$
256	$\approx 1{,}3192530830 \times 10^{502}$

Tabla 1: número de colisiones en el peor caso

Se observa que para generar LSs de orden 16, el número de colisiones en el peor caso todavía puede ser resuelto en un tiempo aceptable, porque aunque posible, el peor caso es muy improbable. Para órdenes mayores el algoritmo puede demorar demasiado en terminar, lo que lo hace impráctico. Se analiza ahora el número esperado de colisiones o el caso promedio.

Cantidad de colisiones esperadas (caso promedio)

Dado que en la generación por filas del algoritmo base se van sacando números al azar, las colisiones que ocurran en cada posición del cuadrado dependen de cuáles sean los valores específicos que se hayan sacado en las filas y columnas previas. Por esto, puede considerarse al número de colisiones en cada posición del cuadrado como variables aleatorias. Se ejecutó 10.000 veces el algoritmo para n=16, pero esta vez contando las colisiones efectivas que se dan en la generación. Las cantidades de colisiones se almacenan en una matriz para luego calcular el promedio de colisiones para cada posición *(i,j)*, obteniendo así una estimación del valor esperado de colisiones en cada posición.

En la **figura 3**, puede observarse cómo se distribuye el promedio de colisiones para cada posición *(i,j)* del LS. Para las filas 1 y 16 no hay colisiones. Para las demás filas, el valor se va incrementando desde la columna $16\text{-}i\text{+}2 = 18\text{-}i$ (donde i es la fila)

hasta la columna 16. El valor máximo se alcanza en la posición (13,14) y es de 34,47. Se observa que el promedio de colisiones de todos los valores mayores que 0 de la matriz (que contiene promedio de todas las ejecuciones) nunca supera el valor 9. Se estima entonces el tiempo del algoritmo considerando que en todas las posiciones donde puede haber colisión se produjeran 9 colisiones. La cantidad de colisiones a partir de la posición *n-i+2* sería entonces la siguiente: dado que los primeros *n-i+1* elementos no llevan a colisión, se tendrían *n-(n-i+1)=i-1* elementos que si lo hacen. Si en cada posición hubiera 9 colisiones, se obtendría la siguiente fórmula:

$$\sum_{i=2}^{n-1} 9^{(i-1)}$$

Fórmula 3: cota superior de la cantidad colisiones caso promedio para n=16

Esto es, para toda fila distinta de la primera y la última, el número de colisiones será 9 multiplicado *(i-1)* veces por sí mismo, lo que para el caso de *n=16* da un total de 25.736.391.511.830 colisiones. Para *n = 16*, preferimos utilizar la cota del peor caso (93.928.268.313) porque es menor, aunque para *n* más grande la cota del caso promedio será mejor.

Complejidad del algoritmo base para orden 16

Supóngase que el costo de elegir aleatoriamente un número y ponerlo en una posición del LS es de orden constante. Aún así, se necesitaría en el mejor caso tomar *n* símbolos y colocarlos en cada fila *n* veces: eso implica n^2 operaciones (si no hay colisiones). Como este es un algoritmo que extrae símbolos aleatoriamente y los coloca en el LS de la manera en que van saliendo en cada fila, el tiempo de ejecución del algoritmo depende de cómo se hayan extraído los símbolos para cada caso en particular. Si la función pseudoaleatoria tiene alguna tendencia, y su valor esperado no es el valor *n/2*, toda la generación tenderá a ir siempre por casos parecidos. En nuestro contexto, el algoritmo en Python es un prototipo para luego implementar un algoritmo real que tendrá una función para extraer símbolos de 1 a n en forma aleatoria, tomando ruido de ambiente (humedad, luz, temperatura) como fuente de datos, o también de fuentes como [11]. Teniendo esto en cuenta, puede suponerse que el algoritmo se comportará más similarmente al caso promedio analizado en la sección anterior. Sin embargo, para n=16 se utilizará la cota del peor caso, como se mencionó anteriormente. Entonces, si el tratamiento de cada colisión (que son 2 o 3 instrucciones Python) llevara del orden de algunos picosegundos[2], la generación del LS tardaría 0,093928268313 segundos (ver **tabla 1**) para tratar todas las colisiones. En este punto no se está teniendo en cuenta que hay colisiones más costosas que otras. El código Python se compila y se optimiza, se optimizan también algunas operaciones del CPU y al ejecutar el programa reiteradas veces (10.000 por ejemplo) la memoria cache juega un rol muy importante, al acceder más rápido a secciones de memoria del programa (datos o código) que se ha utilizado recientemente. Además, se requieren 16×16 operaciones del orden de picosegundos para poner los elementos sin colisión que serían 0,000000000256 segundos (que es despreciable). Es decir, se tienen del orden de 0,09 segundos como cota superior para completar la generación de un LS de orden 16. En la práctica se observa que la generación de un LS de orden 16 es instantánea (siempre tarda del orden de 0,01 o 0,02 segundos como máximo[3]). También se debe tener en cuenta, que para nuestro propósito, que es generar LSs dentro del contexto de un protocolo criptográfico, se usará lenguaje C en lugar de Python. Los scripts utilizados

[2] Todos los tiempos de cómputo de este trabajo fueron calculados usando un procesador Intel Core 2 Duo T6400 @ 2.00 Ghz (con memoria cache de 2 Mb nivel 2 y 3 Gb de RAM).

[3] Tiempos medidos con la función *timeit()* de Python

en este trabajo son prototipos de los algoritmos que se diseñará e implementará en C, siendo C aproximadamente 23 veces más rápido que Python, por ser este último un lenguaje interpretado y de alto nivel.

Generación usando producto de LSs

Buscando en la literatura, se encontró en un trabajo de Koscielny [6] la noción de generar un LS multiplicando dos LSs más chicos. El producto de dos LSs de orden n_1 y n_2 respectivamente retornará un LS de orden $n_1 \times n_2$. La noción de producto es algo complicada, pero es fácilmente implementable en Python:

```
def multLS(ls1, ls2):
  n1 = len(ls1)
  n2 = len(ls2)
  result = [[0 for i in range(0,n1*n2)]
                for j in range(0,n1*n2)]

  for x in range (0, n1*n2):
    for y in range (0, n1*n2):
      result[x][y]=(n2*(ls1[x//n2][y//n2]))
                    +ls2[x%n2][y%n2]
  return result
```

Pseudocódigo 2: generación de un LS usando producto de dos LSs más pequeños

Lo que se tiene aquí es una función que toma dos LSs (ls_1 y ls_2), y retorna un LS que es el producto resultante. En el bucle principal se computa el valor de la posición *(x,y)* del producto como una función lineal sobre las coordenadas y tamaños de ls_1 y ls_2. El símbolo "//" denota la operación de división entera y "%" es la operación de módulo. Como puede verse, el cálculo del producto de dos LSs es de orden $O((n_1 \times n_2) \times (n_1 \times n_2)) = O((n_1 \times n_2)^2) = O(n^2)$ operaciones. Para generar un LS de orden n se deberían encontrar dos divisores de n, por ejemplo n_1 y n_2, tal que $n_1 \times n_2 = n$ y alguno de ellos se aproxime a $n//2$. Esto es porque es mejor (por ejemplo) multiplicar dos LSs de orden 16 para obtener uno de orden 256, que multiplicar uno de orden 2 y otro de orden 128.

Uniformidad de los LSs generados

Entropía de la función *choice()* de Python

A medida que se genera un LS de orden 16 con el algoritmo base, se van restringiendo cada vez más los valores que puede asignarse a cada posición, dado que éstos dependen de los generados en filas y columnas anteriores. Además, en un LS, los valores de *1* a *n* se repiten en cada fila y columna. Por lo tanto, para medir la entropía de la función *choice()* lo más apropiado es simular la generación de un byte aleatorio (caracter ASCII de 0 a 255) sin estar restringidos por otros valores, tal como si se tratara de la variable aleatoria de la posición *(1,1)*. Las demás variables aleatorias del LS serán menos tendenciosas que la de la posición *(1,1)* por estar más restringidas. Para estimar la entropía, se desarrolló un script que extrae un número aleatorio de 0 a 255, escribe el byte ASCII correspondiente en un archivo binario, y repite este proceso diez millones de veces: se genera un archivo con caracteres aleatorios de aproximadamente 10 megabytes. Los resultados arrojados por la herramienta *ENT*[4] son los siguientes:

1) La entropía de la función *choice()* de casi 8 bits por byte indica que el archivo es extremadamente denso en información (esencialmente aleatorio) y que la compresión del archivo no reduciría su tamaño (no hay muchas repeticiones de números).

2) El test χ^2 es muy sensible a los errores en los generadores de números pseudoaleatorios. El porcentaje que arroja este test se interpreta como el grado en que la secuencia testeada es "sospechosa" de ser no aleatoria. El porcentaje no es aceptable si se está cerca de los bordes (1% o 99%); el porcentaje 31,79 obtenido indica que la función es poco tendenciosa, pero no es ideal para usos en criptografía.

3) La media de los datos se acerca a la media esperada (de 127,5).

4) El cálculo del valor de π usando el método de "Monte Carlo" tiene muy poco error, indicando que los datos son aleatorios.

[4] La herramienta ENT puede descargarse de: http://www.fourmilab.ch/random/

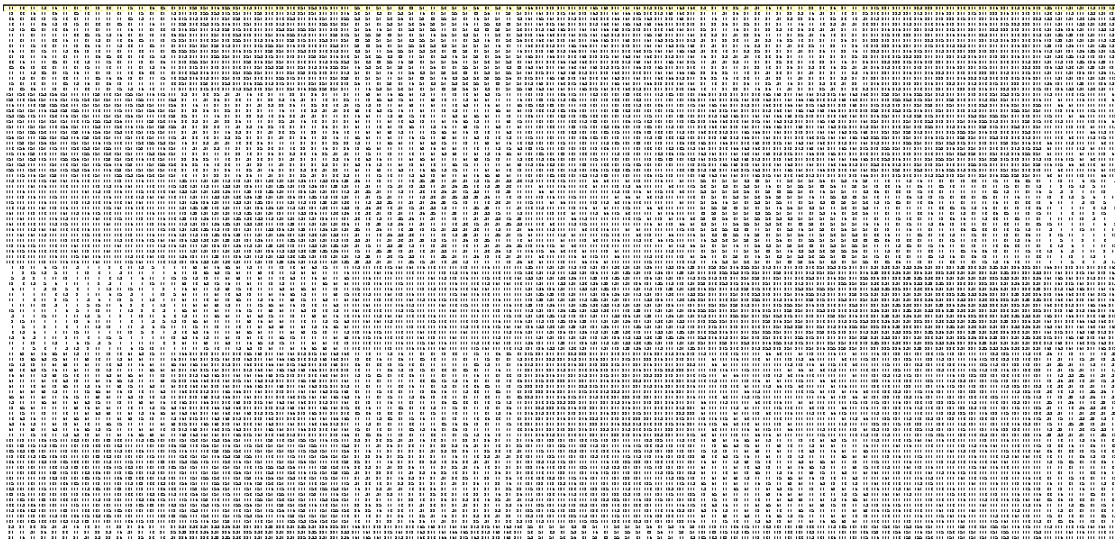

Figura 4: no-uniformidad del LS de orden 256

5) Y por último, el coeficiente de correlación de los datos (que significa en qué medida un byte depende de los anteriores) es muy cercano a 0 e indica correlación casi nula.

Vista parcial de un LS de 256 generado

En la **figura 4** puede observarse una vista parcial de un LS de orden 256 en un archivo de texto, con el tamaño de letra reducido a 6pts. En cada posición hay un número de 0 a 255. Es evidente a simple vista que si bien la generación de un LS de orden 16 es aproximadamente uniforme, al aplicar el producto para generar un LS de orden 256 se generan sub-cuadrados de orden 16 con valores parecidos (cercanos uno del otro), generando distintos tonos de grises en cada cuadrado. Los tonos más claros son números de 1 o 2 cifras y los más oscuros son de 3 cifras.

Histograma de las variables aleatorias

Para verificar si la distribución de los LSs generados es aproximadamente uniforme, se construyó un histograma (como en [5] u [8]) de alguna de las variables que hay en cada posición de un LS. Se utilizó el mismo criterio de antes: se tomó para analizar la variable aleatoria de la primera fila y columna, ya que ésta no está restringida por otros valores anteriores. Se ejecutó la función *choice()* para extraer un número aleatorio del intervalo [0,255] diez millones de veces. Se puede distinguir en la **figura 5** que no todos los intervalos de valores tienen la misma frecuencia: los intervalos de 96 a 112 y de 112 a 128 son menos probables que los demás valores. Para que la distribución sea uniforme todos los intervalos deberían dar en promedio el mismo valor, y en el histograma las barras indican cuales intervalos son más o menos probables. La diferencia de frecuencias entre el máximo valor de un intervalo y el mínimo es de aproximadamente 2750.

Conclusiones

Se ha expuesto una forma sencilla y eficiente de generar LSs de orden 256, desarrollando scripts Python. Esto se hizo partiendo de dos LSs de orden 16 y multiplicándolos (el producto es una traducción de una idea de Koscielny [6]).

Se han corrido pruebas de entropía sobre la función *choice()* de Python (generador de números pseudoaleatorios o PRG) y se ha graficado un histograma para estimar la distribución de probabilidades del PRG, limitando las opciones a números en el intervalo [0,255]. Se ha encontrado que los LSs de orden 256 generados no son verdaderamente aleatorios (no todos los LSs posibles son igualmente probables), ya que el PRG utilizado no tiene distribución uniforme (tiene tendencias).

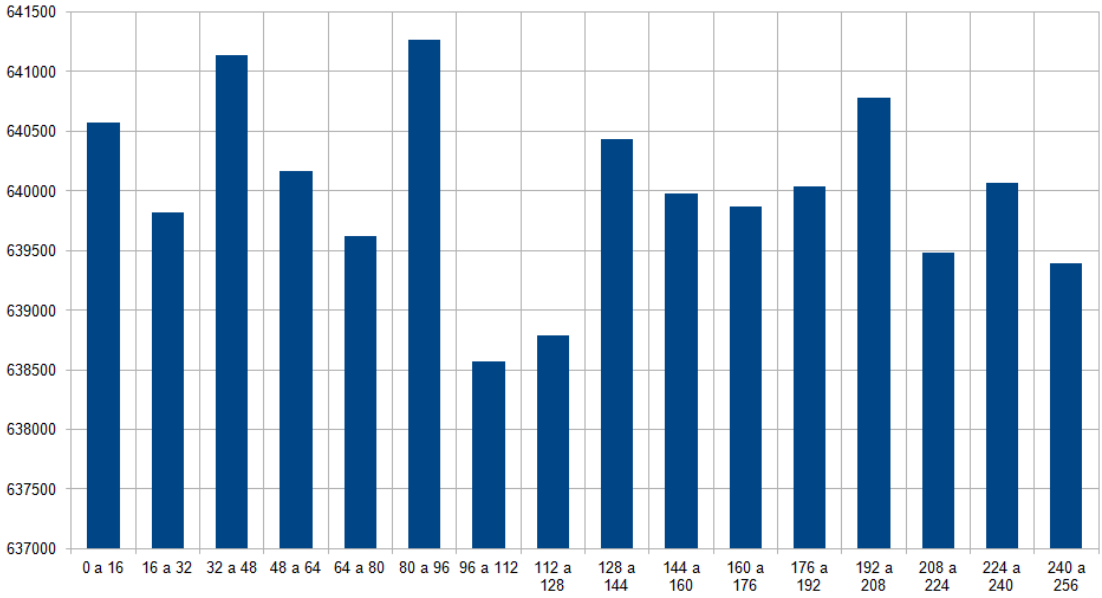

Figura 5: histograma de una variable aleatoria correspondiente a una posición del LS

Como solución, se podría partir de LSs generados y usarlos como punto de partida para aplicar una serie de perturbaciones (movimientos +/-1), como se muestra en [4], para obtener LSs más uniformemente distribuidos.

Trabajo futuro

Queda pendiente explorar otras opciones para obtener LSs más uniformes y en forma eficiente, como implementar el algoritmo de Jackobson y Matthews [4].

Es claro que ningún LS o algoritmo de encriptación es suficiente para establecer una comunicación segura entre dos puntos. Se necesitan otros componentes para completar el esquema de seguridad que se propone diseñar. En este esquema de comunicación, se necesitan mecanismos de seguridad para obtener integridad, confidencialidad y autenticación, además de prevenir otras formas de ataques más avanzados que pueden poner en riesgo la comunicación.

Agradecimientos

A Ariel Iván Ruiz Mateos por hacerme descubrir el fascinante mundo de la seguridad informática; a mi directora de tesis, Claudia Pons, por sus sabias sugerencias y consejos a través de los años. A mi madre, Marta Sagastume y a Alejandra, mi futura esposa, por la paciencia y la revisión del manuscrito; a Alberto Maltz por la orientación en temas de matemática y a Esteban Hurtado por la ayuda con los gráficos en Excel.

Referencias

[1] Wolfram Corporation. Wolfram web site. http://www.wolframalpha.com/, May 2012.

[2] Steve Gibson. *Off the grid* (online). https://www.grc.com/offthegrid.htm, Mayo de 2012.

[3] Danilo Gligoroski, Svein J. Knapskog, y Suzana Andova. Cryptcoding-encryption and error-correction coding in a single step. In Hamid R. Arabnia and Selim Aissi, editors, Security and Management, pages 145–151. CSREA Press, 2006.

[4] Mark T. Jacobson y Peter Matthews. Generating uniformly distributed random Latin squares. J. Combin. Des., 4(6):405–437, 1996.

[5] Gareth J. Janacek y Mark Lemmon Close. Mathematics for Computer Scientists. Ventus Publishing ApS, 2011.

[6] Czeslaw Koscielny. Generating quasigroups for cryptographic applications. Int. J. Appl. Math. Comput. Sci., 12(4):559–569, 2002.

[7] Brendan D. Mckay y Ian M.Wanless. On the number of latin squares. Ann. Combin, 9:335–344, 2005.

[8] Albert R. Meyer. Mathematics for Computer Science. Massachusets Institute of Technology, 2010.

[9] NerdBurrow.com. Proof of hall's theorem. http://www.nerdburrow.com/hall/, Mayo de 2012.

[10] Design Theory Organization. Latin squares definition and examples (online). http://designtheory.org/library/encyc/latinsq/g/, Octubre de 2004.

[11] Random.org. Random.org web site. http://www.random.org, Mayo de 2012.
[12] Van Lint, J.H. y Wilson, R.R.M. A Course in Combinatorics. Cambridge University Press, 2001. ISBN: 9780521803403.
[13] Schneier, Bruce. Applied Cryptography: Protocols, Algorithms, and Source Code in C. 1996, John Wiley and Sons, Inc.
[14] Schneier, Bruce, Ferguson, Niels, y Kohno, Tadayoshi. Cryptography Engineering, Design Principles and Practical Applications.
[15] One-time pad from Wikipedia. http://www.scn.org/~bh162/one-time_pad_encryption.html. Septiembre de 2013.

Datos de Contacto:
Lic. Ignacio Gallego Sagastume.
Facultad de Informática, Universidad Nacional de La Plata.
Dirección postal particular: Calle 5 N°1013, entre 521 y 522, Tolosa.
CP: 1906, La Plata, Buenos Aires, Argentina.
E-mail: ignaciogallego@gmail.com o bien igallego@intrabytes.com