CIENCIA DE LA COMPUTACIÓN

ÁLGEBRA ABSTRACTA

TRABAJO DE INVESTIGACIÓN

- **Apaza Coaquira, Aarón Misash**
- **Choque Mayta, Gabriel Santiago**
- **Condori Gonzales, Jean Carlo**
- **Romero Guillen, Geraldo Mark**
- **Vilca Campana, José Enrique**

CCOMP3-1

2021

Los alumnos declaran haber realizado el trabajo de acuerdo a las normas De la universidad Católica San Pablo

Resumen:

Se implementaron las siguientes soluciones:

- *Test de primalidad:*
    - *Fermat:* Es la prueba de primalidad probabilística más simple.
    - *Solovay-Strassen*:Fue la primera prueba de este tipo en popularizarse por la llegada de las claves públicas de la criptografía.
    - *Miller-Rabin: E*s la prueba de primalidad más utilizada, también conocida como la prueba de pseudoprimos más fuerte.
- *Números aleatorios:*
    - *Trivium: es un cifrado de flujo síncrono diseñado para proporcionar una compensación flexible entre la velocidad y el número de puertas en el hardware además genera hasta* 2^{64} *bits de salida a partir de una clave de 80 bits y un IV de 80 bits .*
    - *Cha Cha: es un cifrado de flujo basado suma de bit a bit XOR y rotaciones*

Criterios de Evaluación:

- Tiempo de ejecución respecto al número de bits.
- Medida de entropía , como nivel de información de predictibilidad y grado de uniformidad en la distribución.

Algoritmo de mejor desempeño:

- *Números aleatorios:*
    - *TRIVIUM*
- *Test de primalidad:*
    - *Miller-Rabin*

Introducción:

El presente trabajo de investigación está motivado por la necesidad de eficiencia y reducción de costo computacional en los algoritmos de aleatoriedad y test de primos. Por consiguiente, todos los algoritmos presentados tienen como objetivo satisfacer este menester.

Por cada algoritmo en este trabajo de investigación se ahondará en su respectiva explicación detallada, implementación en pseudocódigo, acercamiento ejemplar al contenido de las variables en la implementación de los algoritmos, convergencia y eficiencia respecto a bits y su fundamento matemático.

Test de primalidad tradicional

Definición

Se trata de revisar todos los valores desde 2 hasta la raíz cuadrada del número en busca de un divisor del número. Esto se basa en que los divisores de un número supongamos $n = x \,.\, y$ estos dos divisores no podrían ser mayores a la raíz de n Puesto que sino la multiplicación excedería al número, esto quiere decir que todos los divisores de un número se pueden juntar con un divisor menor o igual a la raíz y otro mayor o igual a la raíz, de esta forma solo se buscan divisores en el rango establecido.

Seguimiento:

$n = 47, i = 3$

$47 \; mod \; 2 \neq 0$ //se comprueba si el número es par

$i^2 \neq 47$ //el límite no debe de superar al número

$3^2 \neq 47$ //no lo supera por ende continua

$n \; mod \; i$ //comprobamos los divisores

$47 \; mod \; 3 \neq 0$ //no es divisor por ende continuamos

$i = i + 2$ //continuamos el iterador

$5^2 \neq 47$

$n \; mod \; i$ //comprobamos los divisores

$47 \; mod \; 5 \neq 0$ //no es divisor, siguiente número

$i = i + 2$

$7^2 > 47$ //el cuadrado es mayor al número significa que ya sobrepasó la raíz sin ningún divisor, por lo que 47 es primo

$\Rightarrow 47 \; es \; un \; primo$

Implementación en C++

```
bool esPrimo(ZZ n)
{
    for (ZZ i = ZZ(3); i * i <= n; i += 2)
    {
        cout << i << endl;
        if (module(n, i) == 0) return false;
    }
    return true;
}
```

Fermat primality test

Definición:

La prueba de primalidad probabilística más simple es la prueba de primalidad de Fermat, al ser probabilístico, este proporciona una respuesta correcta en la mayoría de los casos, sin embargo, no proporciona con exactitud la primalidad en algunos números compuestos.

El teorema de Fermat se implementa de la siguiente manera:

Si n es un número primo y a un número entero, $1 \leq a \leq n - 1$, entonces:

$$a^{n-1} mod\, n \equiv 1$$

esta equivalencia no es completamente cierta, pero la desigualdad $a^{n-1} mod\, n \neq 1$es suficiente para probar que n es un número compuesto[1].

Pseudo-Algoritmo:

$Input: An\ odd\ integer\ n \geq 3\ and\ a\ security\ parameter\ t \geq 1.$
$Output: an\ answer\ bool,\ "true"\ or\ "false"\ to\ the\ question: "Is\ n\ prime?"$
$for\ i \leftarrow 1\ to\ t\ do:$
    $Choose\ a\ random\ integer\ a,\ 2 \leq a \leq n - 2$
    $r \leftarrow a^{n-1} mod\, n$
    $If\ r \neq 1\ then\ return\ ("false").$
$return\ ("true")$

Seguimiento: $a^{n-1} mod\, n \equiv 1$

$Prime,\ n = 101,\ a = 2$ $\quad$ $Pseudoprime,\ n = 341 = 11 \times 31,\ a = 2$
$a^{n-1} mod\, n$ $\quad$ $a^{n-1} mod\, n$
$2^{100} mod\, 101 = 1 \leftarrow Correct$
$2^{340} mod\, 341 = 1 \leftarrow Incorrect,\ 341\ no\ es\ primo$

Implementación en C++:

```
bool Fermat(ZZ n,ZZ t){
   for(ZZ i = ZZ(1); i<=t;i++){
      ZZ a=RandomNum(n);
      ZZ r = left_to_right_binary(a,n-1,n);
      if (r!=1)   return false;
   }
```

```
    return true;
}
```

Solovay-Strassen primality test

Definición:

La prueba de primalidad probabilística de Solovay-Strassen fue la primera prueba de este tipo en popularizarse por la llegada de las claves públicas de la criptografía, en particular el criptosistema RSA.

Según Euler:

Si n es un número impar primo, entonces:

$$a^{(n-1)/2} mod\ n \equiv (\frac{a}{n})$$

para todos los enteros a, donde: $mcd(a, n) = 1$

La operación $(\frac{a}{n})$utiliza el símbolo de Jacobi ,este es una función de la aritmética modular que toma dos argumentos y devuelve un valor entero comprendido en el intervalo [-1,1]. En esencia se puede considerar como una generalización del símbolo de Legendre[2].

$$\left(\frac{a}{n}\right) \begin{cases} 0 \ if\ a \equiv 0\ (mod\ n), \\ 1 \ if\ a \neq 0\ (mod\ n)\ y\ existe\ un\ entero\ x\colon\ a \equiv x^2 mod\ n \\ -1 \ if\ a \neq 0\ (mod\ n)\ y\ no\ existe\ x \end{cases}$$

Pseudo-Algoritmo:

$Input\colon An\ odd\ integer\ n \geq 3\ and\ a\ security\ parameter\ t \geq 1.$
$Output\colon an\ answer\ bool,\ "true"\ or\ "false"\ to\ the\ question\colon "Is\ n\ prime?"$
$for\ i \leftarrow 1\ to\ t\ do\colon$
    $Choose\ a\ random\ integer\ a,\ 2 \leq a \leq n-2$
    $r \leftarrow a^{(n-1)/2} mod\ n$
    $If\ r \neq 1\ and\ r \neq n-1\ then\ return\ ("false").$
    $s \leftarrow (\frac{a}{n}),\ using\ the\ Jacobi\ symbol$
    $If\ r\ mod\ n \neq s\ then\ return\ ("false").$

$return\ ("true")$

Seguimiento: $a^{(n-1)/2} mod\ n \equiv (\frac{a}{n})$

$Prime,\ n = 101,\ a = 2$	$Pseudoprime,\ n = 341 = 11 \times 31,\ a = 2$
$a^{(n-1)/2}\ mod\ n$	$a^{(n-1)/2}\ mod\ n$
$2^{50} mod\ 101 = 100, -1$	$2^{170} mod\ 341 = 1$
$s = (\frac{a}{n})\ using\ Jacobi\ symbol$	$s = (\frac{a}{n})\ using\ Jacobi\ symbol$
$s = (\frac{2}{101}) = -1$	$s = (\frac{2}{341}) = -1$
$2^{50} mod\ 101 \equiv (\frac{2}{101}) \equiv -1$	$2^{50} mod\ 101 \neq (\frac{2}{341})$

$Correct, 101\ es\ primo$

$1 \neq -1 \leftarrow Correct, 341\ no\ es\ primo$

Implementación en C++

```
bool Solovay_Strassen(ZZ n,ZZ t){
   for(ZZ i = ZZ(1); i<=t;i++){
      ZZ a = RandomNumber(ZZ(2),n-2);
      ZZ r = modPow(a,(n-1)/2,n);
      if (r!=1 && r!=n-1) return false;
      ZZ s = _Jacobi(a,n);
      if (module(r,n)!=s) return false;
   }
   return true;
}
```

Miller-Rabin primality test

Definición:

Miller-Rabin, es la prueba de primalidad más utilizada, también conocida como la prueba de pseudoprimos más fuerte[3].
Se basa en lo siguiente:

Sea n un número primo impar y sea

$$n - 1 = 2^{s}r,$$

donde r es impar y para cualquier entero a que cumpla:

$$mcd\,(a, n) = 1 \text{y también } a^{r} mod\, n \equiv 1 \text{o } a^{2^{j}r} mod\, n \equiv -1$$

para algún j , $0 \leq j \leq s - 1$.
Si se cumple lo anteriormente mencionado, n se considera un pseudoprimo fuerte en base a a[1].

Si $a^{r} mod\, n \neq 1$ y $a^{2^{j}r} mod\, n \neq 1$ para todo j, $0 \leq j \leq s - 1$, entonces a es un sólido testimonio que prueba definitivamente que $n$es un número compuesto[4].

Pseudo-Algoritmo:

$Input$: $An\ odd\ integer\ n \geq 3\ and\ a\ security\ parameter\ t \geq 1.$
$Output$: $an\ answer\ bool,\ "true"\ or\ "false"\ to\ the\ question$: $"Is\ n\ prime?"$
$Write\ n - 1 = 2^{s}r\ such\ that\ r\ is\ odd.$
$for\ i \leftarrow 1\ to\ t\ do$:
    $Choose\ a\ random\ integer\ a,\ 2 \leq a \leq n - 2$
    $y \leftarrow a^{r} mod\, n$
    $If\ y \neq 1\ and\ y \neq n - 1\ then\ do$:
        $j \leftarrow 1$
        $While\ j \leq s - 1\ and\ y \neq n - 1\ then\ do$:
            $y \leftarrow y^{2} mod\, n$
            $If\ y = 1\ then\ return\ ("false")$
            $j \leftarrow j + 1$
        $If\ y \neq n - 1\ then\ return\ ("false")$

$return\ ("true")$

Seguimiento:

$n = 101\,,\ a = 5$

$n - 1 = 2^{s}r$

$100 = 2^{2}25 \rightarrow s = 2,\ r = 25$

$a^{r} mod\, n \equiv 1\ or\ a^{2^{j}r} mod\, n \equiv -1\,,\ \ 0 \leq j \leq s - 1.$

$5^{25} mod\ 101 = 1 \quad \leftarrow Correct, 101\ es\ primo$

Implementación en C++

```
bool millerRabinTest(ZZ number, ZZ iter)
{
	ZZ d = number - 1;
	while (module(d,ZZ(2)) == 0)
		d >>= 1;
	for (ZZ i (0); i < iter; i++){
		ZZ a = RandomNumber(ZZ(2),number-2);
		ZZ x = modPow(a, d, number);
		if (x == 1  || x == number-1)
			return true;
		while (d != number-1)
		{
			x= module(x*x,number);
			d *= 2;
			if (x == 1)   return false;
			if (x == number-1)  return true;
		}
		return false;
	}
	return true;
}
```

Generadores de números pseudo aleatorios criptográficamente seguros (CSPRNG)

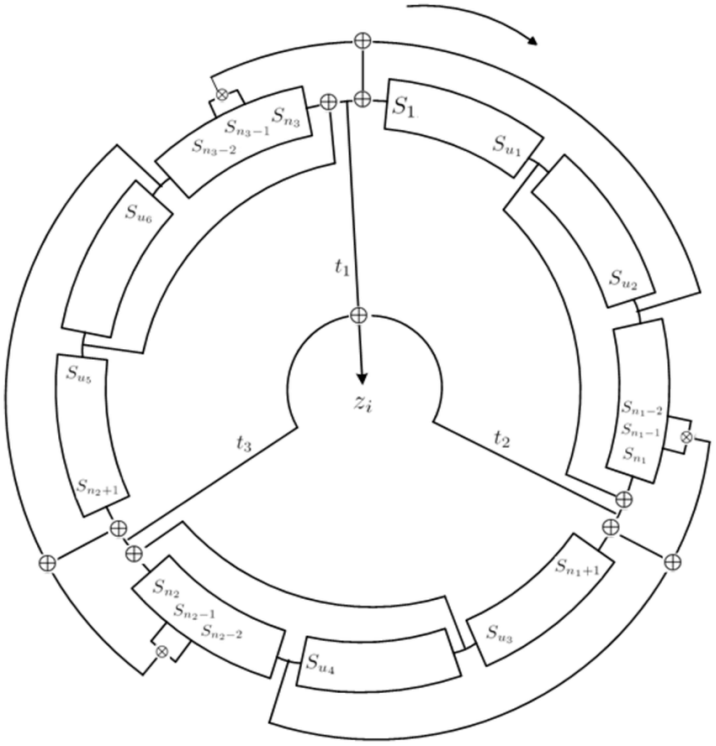
TRIVIUM

Definición

TRIVIUM es un algoritmo de encriptación simétrica creado por Christophe De Canniere y Bart Preneel para el proyecto eSTREAM el cual se llevó a cabo entre los años 2004 y 2008. El algoritmo TRIVIUM está diseñado especialmente para ser implementado fácilmente en hardware [5], es altamente paralelizable y de licencia libre . TRIVIUM es un algoritmo de cifrado de tipo Stream Cipher, esto quiere decir que es capaz de generar un keystream de tamaño 2^{64} bits a partir de una llave privada de 80 bits y de un vector inicial de 80 bits. El algoritmo de cifrado se divide en dos partes; la inicialización del estado interno y la generación del keystream. El estado interno está formado por 288 bits.

El proceso de inicialización del estado interno s consiste en los siguientes pasos:

- Se copia la llave secreta K a las primeras 80 localidades del estado interno y se asigna cero a las localidades de 81 a 93 : $(S_1, S_2, S_3,..., S_{93}) \leftarrow (K_1, K_2,..., K_{80}, 0,.., 0)$
- Se copia el vector inicial IV a las localidades 94 a 173 del estado interno y se asigna a cero a las localidades 174 a 177: $(S_{94}, S_{95}, S_{96},..., S_{177}) \leftarrow (IV_1, IV_2,..., IV_{80}, 0,.., 0)$
- Se asigna cero a las localidades 178 a 285 y uno a las localidades 286,287 y 288 del estado interno: $(S_{178}, S_{179}, S_{180},..., S_{288}) \leftarrow (0,... , 0, 1, 1, 1)$



Parámetros	
Key size	80 bit
IV size	80 bit
Internal state	288 bit

Pseudo-Algoritmo:

"+" y "*" representan a las operaciones XOR y AND respectivamente

function ***TRIVIUM***(N):

$(S_1, S_2, S_3, ..., S_{93}) \leftarrow (K_1, K_2, ..., K_{80}, 0, .., 0)$

$(S_{94}, S_{95}, S_{96}, ..., S_{177}) \leftarrow (IV_1, IV_2, ..., IV_{80}, 0, .., 0)$

$(S_{178}, S_{179}, S_{180}, ..., S_{288}) \leftarrow (0, ..., 0, 1, 1, 1)$

$for\ i\ =\ 1\ to\ 4\ *\ 288\ do$

$t_1 \leftarrow S_{66} + S_{91} * S_{92} + S_{93} + S_{171}$
$t_2 \leftarrow S_{162} + S_{175} * S_{176} + S_{177} + S_{264}$
$t_3 \leftarrow S_{243} + S_{286} * S_{287} + S_{288} + S_{69}$
$(S_1, S_2, \ldots, S_{93}) \leftarrow (t_3, S_1, \ldots, S_{92})$
$(S_{94}, S_{95}, \ldots, S_{177}) \leftarrow (t_1, S_{94}, \ldots, S_{176}$
$(S_{178}, S_{279}, \ldots, S_{288}) \leftarrow (t_2, S_{178}, \ldots, S_{278})$

end for

for i = 1 to N do

$t_1 \leftarrow S_{66} + S_{933}$
$t_2 \leftarrow S_{162} + S_{177}$
$t_3 \leftarrow S_{243} + S_{288}$
$Z_i \leftarrow t_1 + t_2 + t_3$
$t_1 \leftarrow t_1 + S_{91} * S_{92} + S_{171}$
$t_2 \leftarrow t_2 + S_{175} * S_{176} + S_{177} + S_{264}$
$t_3 \leftarrow t_3 + S_{286} * S_{287} + S_{288} + S_{69}$
$(S_1, S_2, \ldots, S_{93}) \leftarrow (t_3, S_1, \ldots, S_{92})$
$(S_{94}, S_{95}, \ldots, S_{177}) \leftarrow (t_1, S_{94}, \ldots, S_{176}$
$(S_{178}, S_{279}, \ldots, S_{288}) \leftarrow (t_2, S_{178}, \ldots, S_{278})$

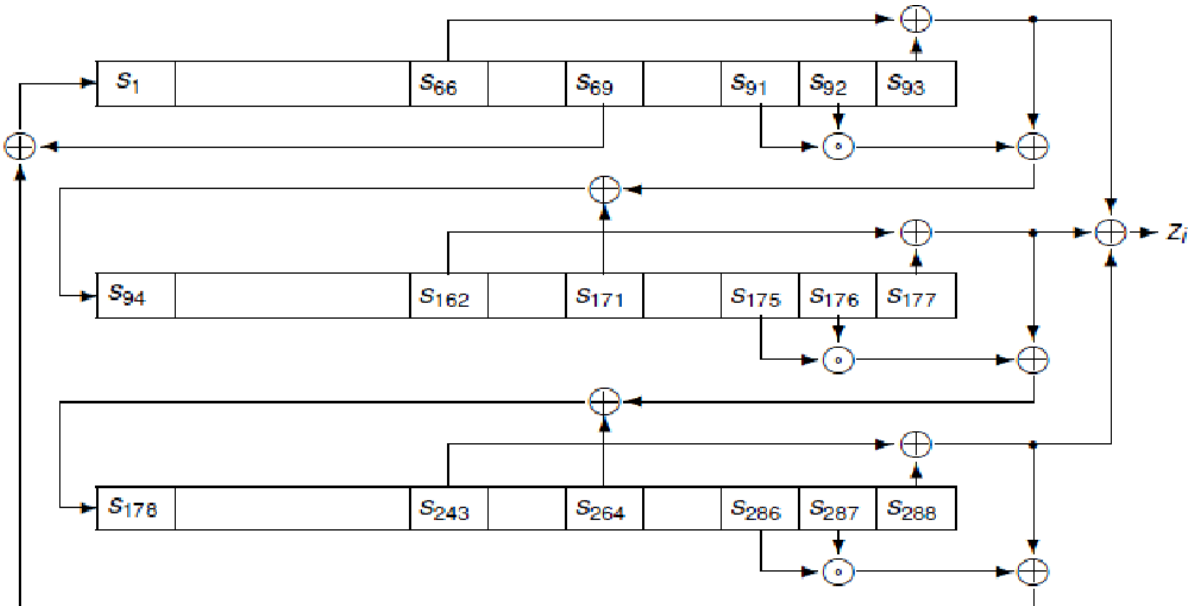
end for

return Z

end function

Seguimiento :

TRIVIUM es un diseño orientado al hardware que se centra en la flexibilidad .Pretende ser compacto en entornos con restricciones en el recuento de puertas, energéticamente eficiente en plataformas con recursos de energía limitados y rápido en aplicaciones que requieren cifrado de alta velocidad.
En el corazón de Trivium hay tres registros de desplazamiento, A, B y C. Las longitudes de los registros son 93, 84 y 111, respectivamente. La suma XOR de las tres salidas de registro forma el flujo de claves si. Una característica específica del cifrado es que la salida de cada registro está conectada a la entrada de otro registro. Por tanto, los registros están dispuestos en forma de círculo. Se puede considerar que el cifrado consta de un registro circular con una longitud total de 93 + 84 + 111 = 288. Cada uno de los tres registros tiene una estructura similar a la que se describe a continuación.



Implementación en C++

```
class TRIVIUM
{

private:
    int s[288];
    int IV[80], key[80];
    int t1, t2, t3;
    int bits;

    void generate_Key_and_IV()
    {
        for (int i = 0; i < 80; i++)
        {
            IV[i] = rand() % 2;
```

```
            key[i] = rand() % 2;
        }
    }

    void rotate()
    {
        for (int j = 93; j >= 1; --j)
            s[j] = s[j - 1];
        for (int j = 177; j >= 94; --j)
            s[j] = s[j - 1];
        for (int j = 288; j >= 178; --j)
            s[j] = s[j - 1];
        s[0] = t3;
        s[93] = t1;
        s[177] = t2;
    }

    void init()
    {
        generate_Key_and_IV();

        for (int i = 0; i < 288; i++)
            s[i] = 0;

        for (int i = 0; i < 80; ++i)
        {
            s[i] = key[i];
            s[i + 93] = IV[i];
        }
        for (int i = 285; i < 288; ++i)
            s[i] = 1;

        //rotate
        for (int i = 1; i < 1152; i++)
        {
        t1 = ((s[65] ^ (s[90] & s[91])) ^ s[92]) ^ s[170];
        t2 = ((s[161] ^ (s[174] & s[175])) ^ s[176]) ^ s[263];
```

```
        t3 = ((s[242] ^ (s[285] & s[286])) ^ s[287]) ^ s[68];
            rotate();
        }
    }

    ZZ keyStream()
    {
        ZZ z(0);
        ZZ base10(0);

        for (int i = 0; i < bits; i++)
        {
            t1 = s[65] ^ s[92];
            t2 = s[161] ^ s[176];
            t3 = s[242] ^ s[287];

            z = t1 ^ t2 ^ t3;

            t1 ^= s[90] ^ s[91] & s[170];
            t2 ^= s[174] ^ s[175] & s[263];
            t3 ^= s[285] ^ s[286] & s[68];
            rotate();
            base10 += z << i;
        }
        return base10;
    }

public:
    Trivium(int bits)
    {
        this->bits = bits;
        init();
    }

    ZZ PseudoRandom()
    {
        ZZ n;
```

```
        do
        {
            n = keyStream();
        } while (countBits(n) != bits);

        return n;
    }
    int countBits(ZZ n)
    {
        int count = 0;
        while (n != 0)
        {
            count++;
            n >>= 1;
        }
        return count;
    }
};
```

Blum Blum Shub

Definición

Blum Blum Shub (BBS) es un generador pseudoaleatorio de números propuesto por Lenore Blum, Manuel Blum y Michael Shub en 1986.

El algoritmo BBS es:

$x_{n+1} = (x_n)^2 mod\ M$

donde:

$p \equiv q \equiv 3\ en\ Z_4$ y con un pequeño $mcd((p-3)/2, (q-3)/2)$

$M = pq$

$mcd(M, s) = 1$

$s \neq 1\ or\ 0$

$x_{-1} = s$

[6]

Pseudo algoritmo

INPUT : m, s

OUTPUT : un número pseudo aleatorio.

FUNCTION BBS

1. s = s*s mod m
2. return s

s es la variable donde está la semilla, puede estar en un scope superior(no recomendado el global) o ser pasado por referencia, al igual que m.

Implementación en C++

```
class BBS
{
private:
    ZZ p, q, m, s;
public:
    BBS(ZZ p, ZZ q, ZZ seed)
    {
        this->p = p;
        this->q = q;
        this->s = seed;
        this->m = p * q;
    }
    ZZ getRandNum()
    {
        ZZ nextRandNum = mod((this->s * this->s), this->m);
        this->s = nextRandNum;
        return nextRandNum;
    }
};
```

[7]

Seguimiento:

$p = 11, q = 23, M = 253$
$mcd(M, 3) = 1, s = 3$
$x_{-1} = 3$
$x_{n+1} = (x_n)^2 mod\ M$

$x_0 = (x_{-1})^2 mod\ M$

$x_0 = (3)^2 mod\ 253$

$x_0 = 9$

$x_1 = (9)^2 mod\ 253$

$x_1 = 81$

$x_2 = (81)^2 mod\ 253$

$x_2 = 6561\ mod\ 253$

$x_2 = 236\ mod\ 253$

$x_3 = (236)^2 mod\ 253$

$x_3 = 55696\ mod\ 253$

$x_3 = 36\ mod\ 253$

$x_4 = (36)^2 mod\ 253$

$x_4 = 1296\ mod\ 253$

$x_4 = 31\ mod\ 253$

$x_5 = (31)^2 mod\ 253$

$x_5 = 961\ mod\ 253$

$x_5 = 202\ mod\ 253$

$x_0, x_1, x_2, \dots, x_5 = 9, 81, 236, 36, 31, 202$

Chacha20

Definición:

En 2008 Bernstein publicó una modificación al algoritmo de salsa20. Su nombre viene, además de dar juego con el algoritmo de origen, de que los bits en cada una de las celdas parecieran bailar.[12]

Algoritmo:

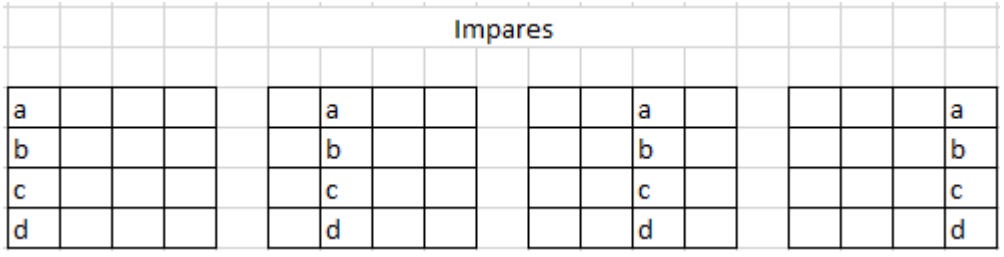
- estado inicial
    - Una matriz de 4 x 4, donde cada celda es de 32 bits.
    - La primera fila son constantes, la constante convencional es el string “expand 32-byte k”
    - las siguientes dos filas son la llave, que en el estándar aes su equivalente sería una de 256 bits.
    - La última fila está compuesta por un contador, o *block number* , que nos permite saltar a la mitad de un archivo si queremos; y 3 nonce, un número u otra variable que solo usaremos una vez.

Initial state of ChaCha

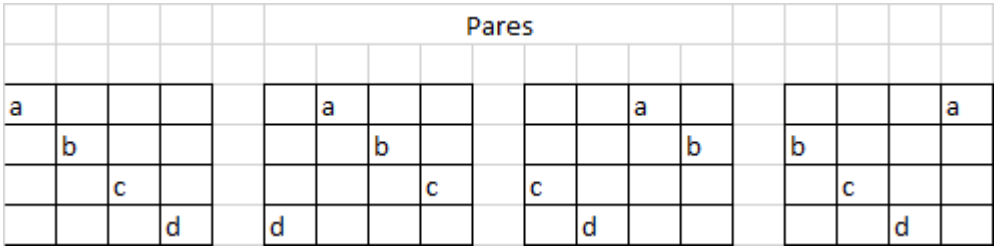
"expa"	"nd 3"	"2-by"	"te k"
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Nonce	Nonce	Nonce

Tenemos nuestra matriz original, la pasamos por las operaciones add-rotate-xor, y la matriz resultado la sumamos con la matriz inicial. Cada uno de los valores en las celdas es el número pseudo aleatorio resultante.

Operaciones add-rotate-xor.
En las iteraciones impares operamos las columnas,

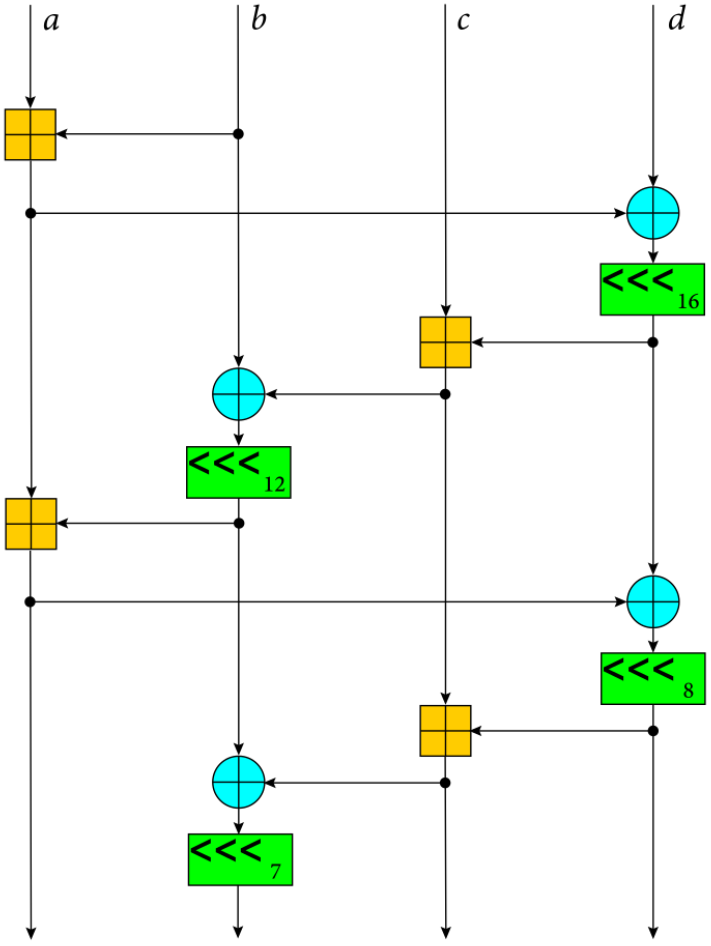


en las iteraciones pares las diagonales, comenzando por la gran diagonal, continuando por la mitad superior.



con las variables a, b, c, d establecidas. Procedemos con la siguiente sucesión de operaciones, como lo indica el diagrama de flujo.

adición binaria o XOR: ⊕
32-bit addition mod 2^{32}: ⊞
rotación de distancia constante: <<<



El uso de estos tres operadores evita la posibilidad de ataques de tiempo en el software, ya que no importa el input o lo que procesen por estos operadores, siempre se tardará igual.[8][9]

Pseudocódigo:

XOR y OR son operadores a nivel binario
el operador << es desplazamiento a la izquierda
el operador >> es desplazamiento a la derecha

ROTL(a, b)

1. (((a) << (b)) OR ((a) >> (32 - (b))))

QR(a, b, c, d)

1. a = a + b,
2. d = d XOR a,
3. d = ROTL(d, 16),
4. c = c + d,
5. b = b XOR c,
6. b = ROTL(b, 12),
7. a = a + b,
8. d = d XOR a,
9. d = ROTL(d, 8),
10. c = c + d
11. b = b XOR c,
12. b = ROTL(b, 7))

1. LOOP 10 veces. Los valores en las funciones son los índices de la matriz inicial
   1.1. // Odd round
   1.2. QR(state, 0, 4, 8, 12) // 1st column
   1.3. QR(state, 1, 5, 9, 13) // 2nd column
   1.4. QR(state, 2, 6, 10, 14) // 3rd column
   1.5. QR(state, 3, 7, 11, 15) // 4th column
   // Even round
   1.6. QR(state, 0, 5, 10, 15) // diagonal 1 (main diagonal)
   1.7. QR(state, 1, 6, 11, 12) // diagonal 2
   1.8. QR(state, 2, 7, 8, 13) // diagonal 3
   1.9. QR(state, 3, 4, 9, 14) // diagonal 4

   // Tras iterar 10 veces aquel bloque de QuarterRounds(QR). sume la matriz resultante con la matriz inicial.
2. Final matrix = Initial Matrix + result matrix

[10][11]

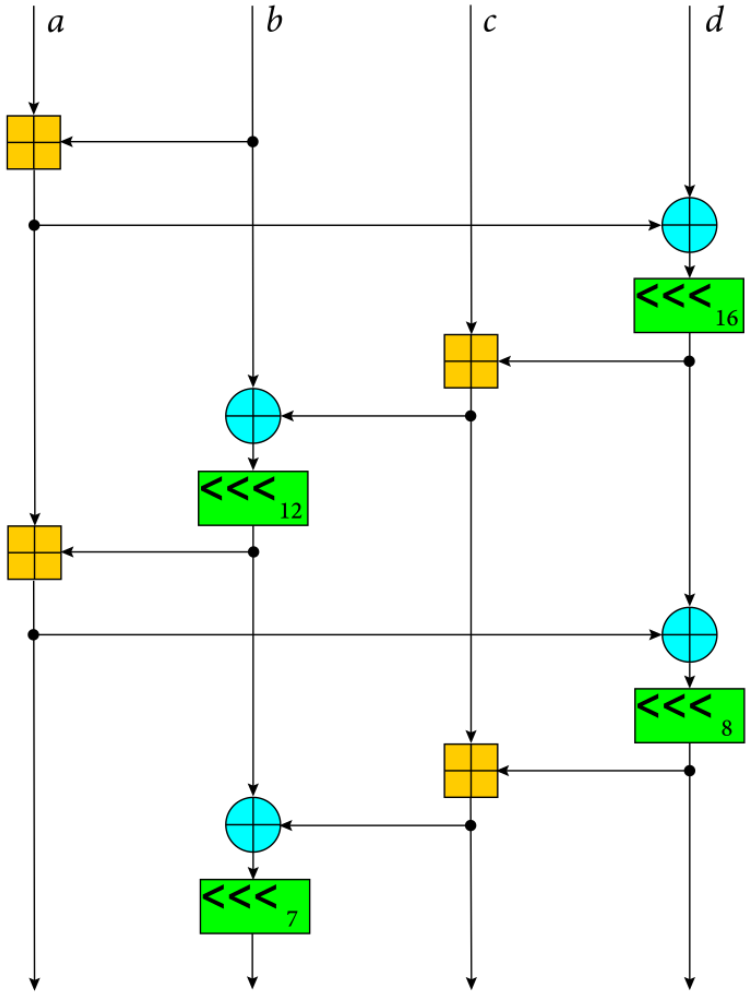
Seguimiento:

nonce = counter = 11
key = 1
const [0][0] = 42827 = 1010011101001011

const [0][1] = 6558 = 0001100110011110
const [0][2] = 56124 = 1101101100111100
const [0][3] = 4185 = 0001000001011001
Nuestra matriz inicial es esta

	0	1	2	3
0	1010011101001010	1100110011110	1101101100111100	0001000001011001
1	0000000000000001	0000000000000001	0000000000000001	0000000000000001
2	0000000000000001	0000000000000001	0000000000000001	0000000000000001
3	0000000000001011	0000000000001011	0000000000001011	0000000000001011

en cada iteración se aplica el siguiente diagrama de flujo



resultado final:

	0	1	2	3
0	2035711850	291759761	898797097	3533627726
1	2723371319	462505721	2344030699	4195144790
2	527304851	3774393083	3520247395	709185862
3	3508653707	3902631038	3901445067	1157652586

Implementación en C++

```
#define ROTL(a, b) (((a) << (b)) | ((a) >> (32 - (b))))
#define QR(a, b, c, d) (               \
    a += b, d ^= a, d = ROTL(d, 16), \
    c += d, b ^= c, b = ROTL(b, 12), \
    a += b, d ^= a, d = ROTL(d, 8),  \
    c += d, b ^= c, b = ROTL(b, 7))

void chacha_block(ZZ out[16], ZZ const in[16])
{
    int i;
    ZZ x[16];

    for (i = 0; i < 16; ++i)
        x[i] = in[i];
    // 10 loops × 2 rounds/loop = 20 rounds
    for (i = 0; i < 20; i += 2)
    {
        // Odd round
        QR(x[0], x[4], x[8], x[12]);  // column 0
        QR(x[1], x[5], x[9], x[13]);  // column 1
        QR(x[2], x[6], x[10], x[14]); // column 2
        QR(x[3], x[7], x[11], x[15]); // column 3
        // Even round
        QR(x[0], x[5], x[10], x[15]); // diagonal 1 (main
diagonal)
        QR(x[1], x[6], x[11], x[12]); // diagonal 2
        QR(x[2], x[7], x[8], x[13]);  // diagonal 3
        QR(x[3], x[4], x[9], x[14]);  // diagonal 4
```

```
    }
    for (i = 0; i < 16; ++i)
    {
        out[i] = x[i] + in[i];
        cout << out[i] << endl;
    }
    cout << endl;
}
```

[13]

Análisis de Algoritmos

Características del procesador y sistema operativo:

- AMD Ryzen 5 3400G with Radeon Vega Graphics (8 CPUs), ~3.7Ghz
- Windows 10 Pro 64 bits
- 16384MB RAM

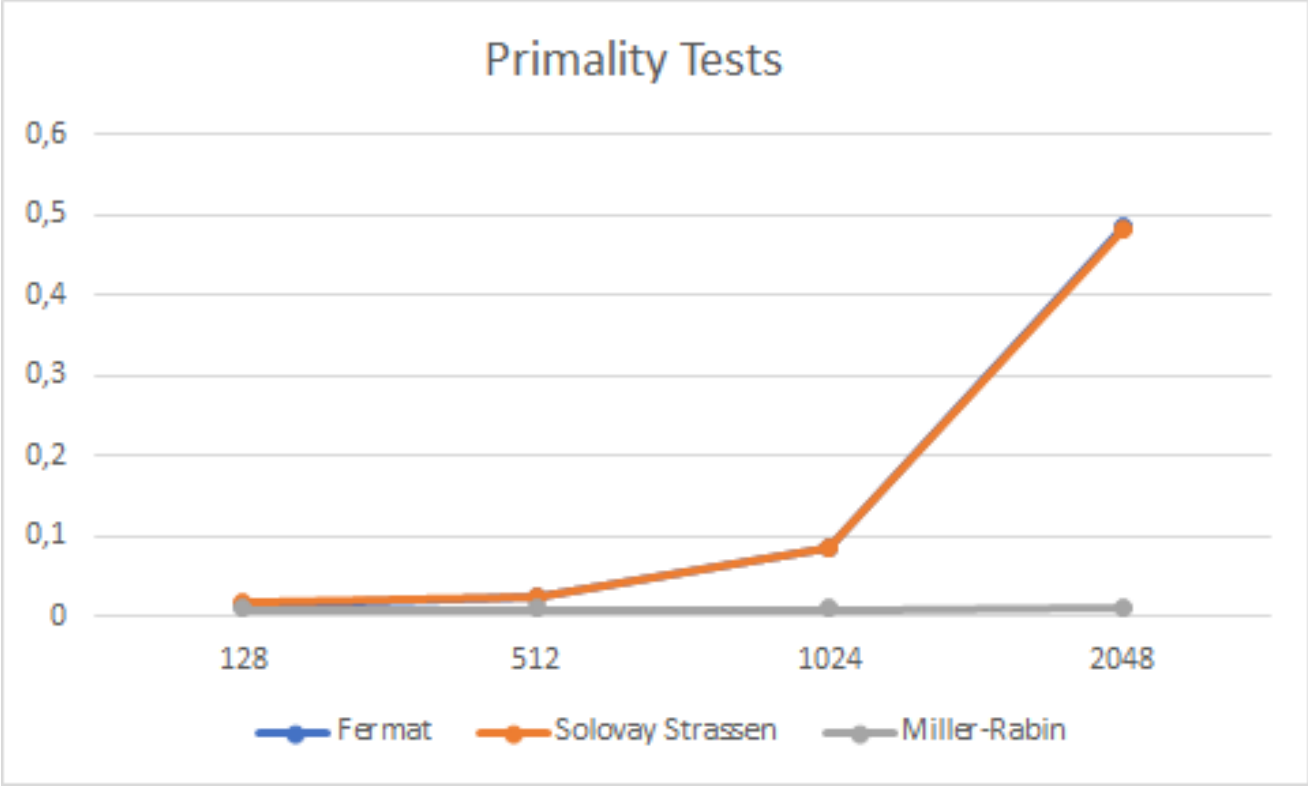
Tiempo de ejecución vs. Nro. de Bits

128 bits											
	Intentos										Promedio
	1	2	3	4	5	6	7	8	9	10	
millerRabinTest	0,01	0,011	0,01	0,01	0,013	0,011	0,01	0,01	0,01	0,011	0,0106
Fermat	0,012	0,016	0,012	0,017	0,015	0,015	0,017	0,013	0,018	0,015	0,015
Solovay_Strassen	0,017	0,017	0,018	0,019	0,017	0,016	0,019	0,02	0,018	0,016	0,0177

512 bits											
	Intentos										Promedio
	1	2	3	4	5	6	7	8	9	10	
millerRabinTest	0,011	0,011	0,012	0,009	0,01	0,011	0.009	0,012	0.01	0,01	0,0105
Fermat	0,027	0,022	0,024	0,029	0,023	0,025	0,027	0,028	0,029	0,029	0,0263
Solovay_Strassen	0,024	0,026	0,028	0,027	0,025	0,023	0,025	0,029	0,023	0,023	0,0253

1024 bits											
	Intentos										Promedio
	1	2	3	4	5	6	7	8	9	10	
millerRabinTest	0,01	0,012	0,009	0,012	0,012	0,01	0,01	0,01	0,01	0,01	0,0105
Fermat	0,082	0,087	0,082	0,087	0,09	0,082	0,092	0,082	0,083	0,083	0,085
Solovay_Strassen	0,09	0,087	0,081	0,081	0,089	0,087	0,086	0,086	0,091	0,086	0,0864

2048 bits											
Intentos											Promedio
	1	2	3	4	5	6	7	8	9	10	
millerRabinTest	0,012	0,01	0,014	0,01	0,012	0,009	0,013	0,009	0,017	0,013	0,0119
Fermat	0,508	0,484	0,481	0,481	0,49	0,464	0,505	0,488	0,487	0,476	0,4864
Solovay_Strassen	0,487	0,472	0,486	0,492	0,49	0,48	0,481	0,469	0,483	0,477	0,4817



Test de Entropía vs. Nro. de Bits

128 bits											
	Intentos - Entropía										
	1	2	3	4	5	6	7	8	9	10	Promedio
Trivium test	0.995593	0.998414	1	0.998414	0.995593	1	0.999295	0.993651	0.993651	1	0.9974611
Cha Cha	0.99984	0.999905	0.999499	0.996923	0.794815	0.862055	0.806327	0.872302	0.998437	0.998988	0.9329091

512 bits											
	Intentos - Entropía										
	1	2	3	4	5	6	7	8	9	10	Promedio
Trivium test	0.999295	0.998899	0.996431	0.999901	0.997842	0.999956	0.995593	0.994666	0.999295	0.999989	0.9981867
Cha Cha	0.994766	0.999907	0.830496	0.830051	0.762606	0.862046	0.998341	0.998642	0.999037	0.999816	0.9275708

1024 bits											
	Intentos - Entropía										
	1	2	3	4	5	6	7	8	9	10	Promedio
Trivium test	1	0.999989	0.997685	0.999931	0.998899	0.999205	1	0.999956	0.999824	0.99718	0.9992669
Cha Cha	0.802719	0.80815	0.834008	0.811723	0.999941	0.996449	0.998371	0.999376	0.872548	0.869655	0.899294

2048 bits											
	Intentos - Entropía										
	1	2	3	4	5	6	7	8	9	10	Promedio
Trivium test	0.998899	0.999824	0.999975	0.999989	0.999535	0.999381	0.999917	0.999901	0.999989	0.999975	0.9997385
Cha Cha	0.794739	0.811278	0.999996	0.999037	0.999624	0.999865	0.823922	0.846935	0.842156	0.813494	0.8931046

Conclusión:

Respecto a los test de primalidad, con una gran diferencia, se concluye que el test con mejor desempeño es la prueba de Miller-Rabin, debido a su optimización de tiempo al reducir el exponente de la exponenciación modular utilizando la fórmula $n - 1 = 2^{s}r$ y a tener una mayor precisión debido a 2 fuertes condiciones para encontrar los primos :

$$a^{r} mod\, n \equiv 1, a^{2^{j}r} mod\, n \equiv -1$$

Además aumentar sus probabilidades al analizar casos especiales como los números pares ,este algoritmo tiene una implementación binaria para la división y multiplicación por 2 , siendo más eficiente un corrimiento de bits.

Al realizar el test de entropía para los Generadores de números pseudo aleatorios criptográficamente seguros (CSPRNG) , se observa que que el algoritmo TRIVIUM tienen como medida promedio de entropía a 0.99 y llegando a 1 algunas pruebas , entonces se puede interpretar que su distribución tiende a ser uniforme , caso contrario con las pruebas al algoritmo CHA CHA , en cual tiende a tener una medida más sesgada . La razón por la cual es importante que nuestro generador de números aleatorios , es debido a que existe una relación proporcional entre la medida de entropía y el grado de predictibilidad de la secuencia de bits.

También se concluye que TRIVIUM tiene mayor eficiencia debido a su implementación binaria , porque es un algoritmo enfocado a la eficiencia energética en hardware ; llegando a tener ejecuciones rápidas ; Según [14] trivium tiene la posibilidad de paralelizar sus operaciones , por lo que es capaz de generar hasta 64 bits por ciclo.

Se concluye que TRIVIUM tiene una implementación razonablemente eficiente respecto al número de bits, siendo uno de los cifrados de flujo más prometedores seleccionados por el proyecto STREAM .

REFERENCIAS:

[1] Handbook of Applied Cryptography, Menezes, Oorschot, Vanstone. CRC Press, New York, fifth edition (2001).Capítulo 4: Public key parameters. Random search for probable primes. Página 145. Handbook of Applied Cryptography (uwaterloo.ca)

[2] Jacobi Symbol ,2019 by Bruce Ikenaga jacobi-symbol.dvi (millersville.edu)

[3] Probabilidad, Números Primos y Factorización de Enteros. Implementaciones en Java y VBA para Excel. Revista digital Matemática, Educación e Internet.
Probabilidad_Primos_Factorizacion.pdf (tec.ac.cr)

[4] Improving the Speed and Accuracy of the Miller-Rabin Primality Test by Shyam Narayanan , MIT PRIMES-USA.
Narayanan.pdf (mit.edu)

[5] Jafarpour, A., Mahdlo, A., Akbari, A., Kianfar, K., "Grain and Trivium ciphers implementation algorithm in FPGA chip and AVR micro controller," 2011 IEEE

[6]Blum, L., Blum, M., & Shub, M. (1986). A Simple Unpredictable Pseudo-Random Number Generator. *SIAM Journal on Computing*, *15*(2), 364–383.
https://doi.org/10.1137/0215025

[7]BBS. (2020). [Blum Blum Shub c++ implementation]. OverStruck.

https://github.com/OverStruck/blum-blum-shub-prbg

[8]Haran, B. (2021, February 19). *Chacha cipher - Computerphile* [Video]. YouTube.

https://www.youtube.com/watch?v=UeIpq-C-GSA&feature=youtu.be

[9]Techopedia. (2014, April 21). *Nonce*. Techopedia.Com.

https://www.techopedia.com/definition/10297/nonce

[10]Google. (2015, May). *ChaCha20 and poly1305 for IETF protocols*. Datatracker.Ietf.Org.

https://datatracker.ietf.org/doc/html/rfc7539

[11]Manullang, I. T. (2020). *The implementation of XChaCha20-Poly1305 inMQTT protocol*. Informatika.Stei.Itb.Ac.Id.
https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Makalah-UAS/Makalah-UAS-Kripto-2020%20(20).pdf

[12]Tezcan, C. (2021, March 30). *Network Security 3.6: ChaCha20* [Video]. YouTube.

https://www.youtube.com/watch?v=Y5qvduBewUA&feature=youtu.be

[13]Salsa20. (2021, May 17). In *Wikipedia*. https://en.wikipedia.org/wiki/Salsa20

[14] Gun, I , Nava .o (2018). Implementación del algoritmo de cifrado trivium en un sistema embebido.