El Sistema RSA

José de Jesús Ángel Ángel
jesus@seguridata.com

SeguriDATA
www.seguridata.com

El Sistema RSA

José de Jesús Ángel Ángel
@seguridata.com.mx

1 Introducción

Desde su aparición en 1978 ([54]) el sistema de llave pública **RSA** (de **R**ivest,**S**hamir y **A**dleman) ha ganado gran popularidad, por una parte por la gran seguridad que ofrece al basar ésta en un problema matemático difícil de resolver que había dejado de tener interés en la comunidad mundial, cómo lo es el **P**roblema de la **F**actorización **E**ntera **PFE** y a causa del sistema **RSA** se ha retomado e incrementado su investigación. Por otra parte, aunque implementar **RSA** requiere de mucho cuidado en detalles que son necesarios, la idea de su funcionamiento es muy simple de entender, lo que lo hace muy popular principalmente en sectores donde no hay abundancia de matemáticas.
Este artículo está dedicado a resumir los últimos resultados paso a paso sobre la implementación del sistema **RSA**, principalmente en lo que se refiere a su seguridad y más concretamente en los puntos de mayor debilidad del sistema, éstos están resumidos en lo que llamamos resultados, que se reescriben en la última parte del artículo.
Respecto a la implementación de las técnicas de realizar aritmética con números de múltiple precisión o longitud "grande", se requiere de mayor espacio, aunque se dan las referencias necesarias para abordar con seriedad el tema.

El algoritmo **RSA** es usado esencialmente en:
a) Generación de llaves **RSA**
b) Cifrado del texo original m
c) Descifrado del texto cifrado c

Algoritmo RSA de llave pública

l.a) Generar dos primos *p,q*

2.a) Calcular $n = pq, \phi(n) = (p-1)(q-1)$

3.a) Elegir un entero $e,\ 1 < e < \phi, \quad (e, \phi) = 1$

4.a) Calcular $d = e^{-1} \bmod \phi$

5.a) La llave pública es (n, e) y la llave privada (d)

6.b) El usuario **A** calcula $c = m^e \bmod n$, con la llave pública (n, e)

7.b) **A** envía el mensaje cifrado *c* al destinatario **B**

8.c) **B** recobra el mensaje con la fórmula $m = c^d \bmod n$, con la llave privada d

Otra importante aplicación es la "Firma Digital" que consiste en lo siguiente:

Algoritmo de Firma Digital

A) Si el usuario **B** quiere firmar el mensaje m , se procede a calcular $s = m^d \bmod n$,

donde d es la llave privada de **B**

B) Para verificar la firma de **B** al mensaje m , se procede como sigue: *s* es la firma de

B $\Leftrightarrow s^e = m \bmod n$

Enseguida revisaremos temas relacionados con cada uno de los anteriores puntos y justificaremos de forma simple algunos de ellos, o en caso de que sea de mayor tratamiento, damos las referencias.

2 El tamaño de la llave pública n

Si podemos factorizar al número n , entonces podemos conocer a ϕ , y por lo tanto calcular a d a partir de e , es decir, romperemos el sistema.

El problema de calcular a d y factorizar a n son computacionalmente equivalentes, la elección de p y q debe ser de tal forma que la factorización de $n = pq$ sea computacionalmente "imposible". A continuación procedemos a encontrar el tamaño de los números primos p , q , para lograr el objetivo.

Si $n = 2^r$ entonces n esta en el extremo de ser fácilmente factorizable, es decir, n esta en la forma más simple de poder ser factorizable ya que n tiene muchos factores y estos son del menor tamaño posible. Entonces si n tiene un mínimo de factores y tales factores son grandes, n posiblemente estará en el otro extremo, el de ser "imposible" su factorización. Es decir si $n = pq$ donde p y q son números primos y de considerable tamaño, entonces la factorización de n debe ser difícil.
A continuación veremos que los números producto de dos primos son de los más difíciles de factorizar.

2.1 Criterios de factorización

Una forma de clasificar a los métodos de factorización es en dos grupos: 1. Los métodos de "propósito especial", quienes buscan propiedades especiales a los factores de n y 2. Los métodos de "propósito general" que dependen sólo del tamaño de n .
Entre los métodos de "propósito especial" están:

- El método del ensaño de divisores pequeños (conocido por la criba de Eratosthenes)

- El algoritmo $p-1$ de Pollard ([46])
- El método que usa curvas elípticas **(MCE)** ([36])
- Criba de campos numéricos especial **(CCNE**) ([33], [34])

Entre los métodos de "propósito general" están

- Criba cuadrática ([21], [33], [49])
- Criba de campos numéricos general **(CCNG)** ([33])

En general no existe una forma eficiente de factorizar a un número entero si no se conoce nada acerca de él, ya que algunos métodos son más eficientes en algunos casos que en otros, sin embargo podemos dar una estrategia para poder intentar factorizar un número entero n *:*

- Aplicar el método de ensañar divisores pequeños, hasta una cota c_0

- Aplicar el algoritmo de Pollard, esperando encontrar un factor r , tal que $c_0 < r < c_1$ donde c_1 es otra cota

- Aplicar el método con curvas elípticas, esperando encontrar un factor r_1 tal que $c_1 < r < c_2$
- Aplicar un método de propósito general

Con lo que esperamos optimizar el proceso de factorizar, aplicando el mejor método conforme a la forma del número.

Si ya se conoce una estrategia para factorizar al número entero, el parámetro más importante es el tiempo que tomará para lograr el objetivo. Enseguida procedemos a explicar una forma simple de medir el tiempo.

2.2 Cómo medir el tiempo en factorizar números enteros

La práctica ha permitido llegar a medir algunos algoritmos con la siguiente cota:

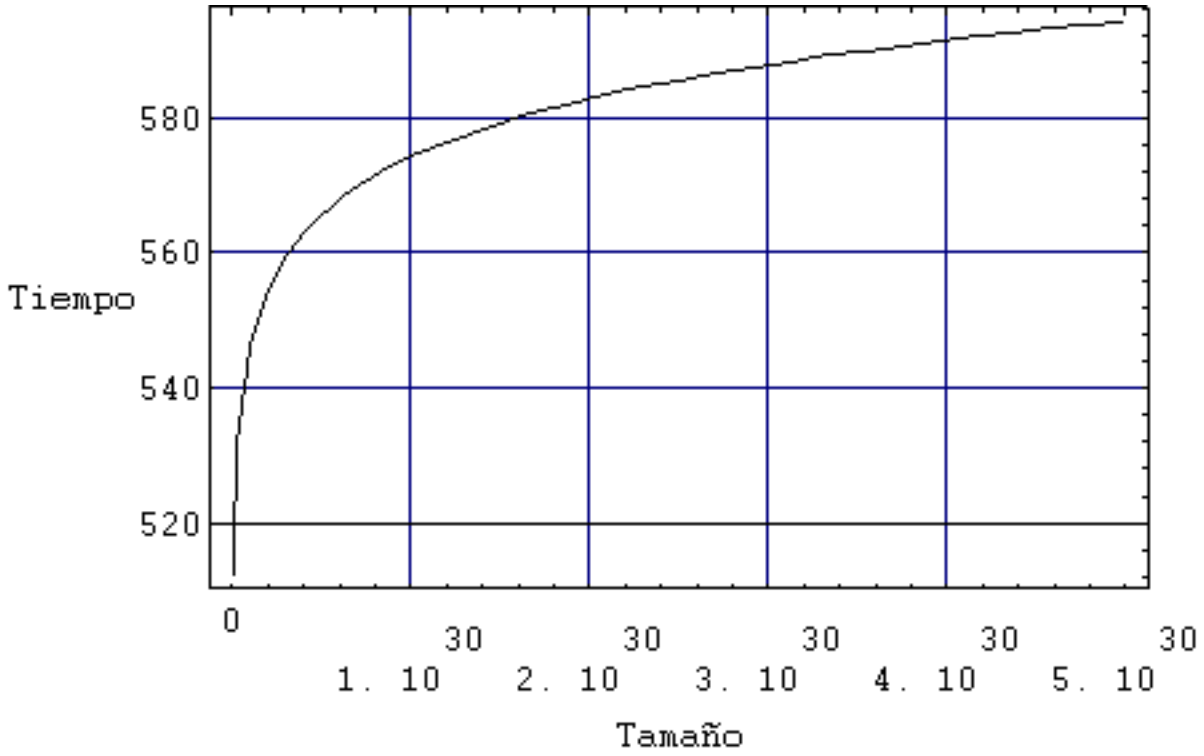
$$L[n,v,c] = O\left[\exp(c(\log n)^{v}(\log\log n)^{1-v})\right]$$

donde podemos suponer que la función como cota superior a considerar es:

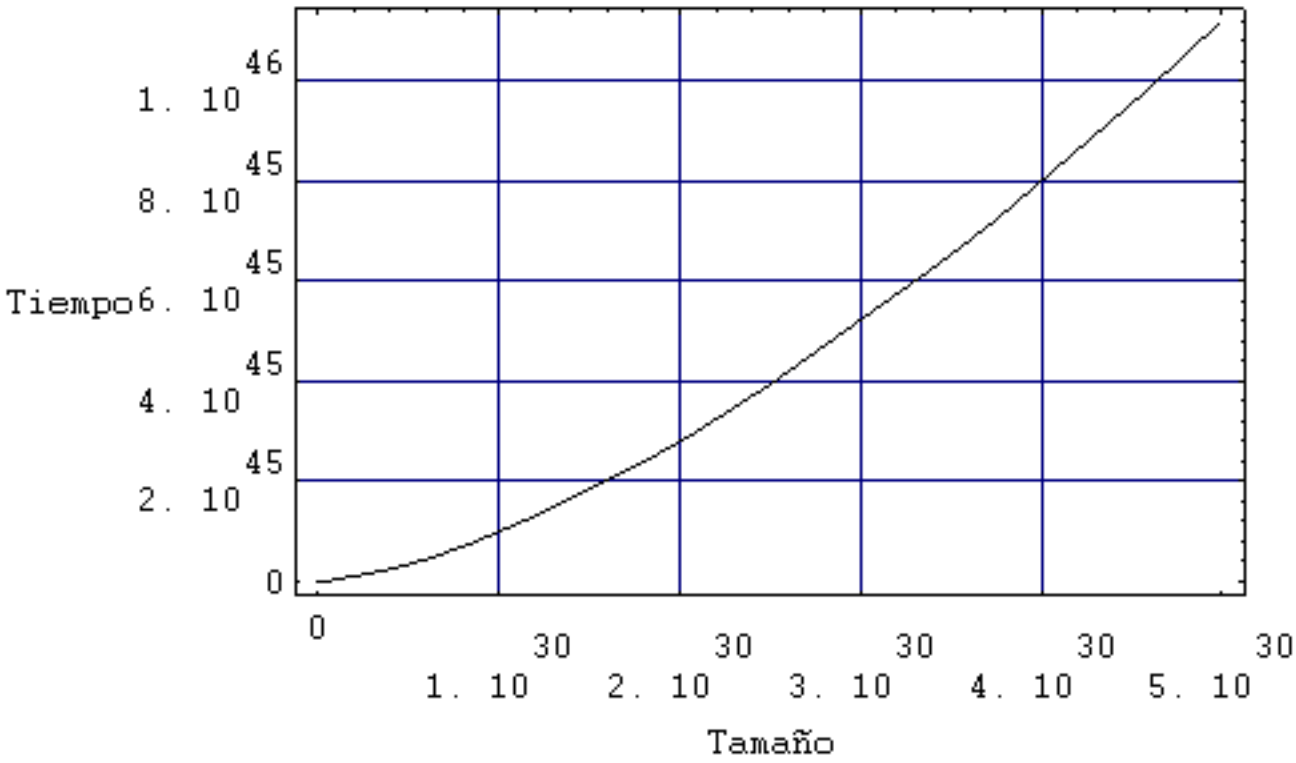
$$L[n,v,c] = \exp(c(\log n)^{v}(\lg\log n)^{1-v}$$

donde c es una constante, n el tamaño del número, v un número tal que $0 < v < 1$ y determina la "lentitud" del algoritmo en el sentido de que si $v = 1$, entonces L se convierte a $L[n,1,c] = \exp(c\log n)$, y si $v = 0$, entonces $L[n,0,c] = \exp(c\log\log n)$, cuyos comportamientos los podemos apreciar en las siguientes gráficas.

Esta gráfica es de la función: $L[n, 0, 1.5]$



Esta gráfica es de la función: $L[n, 1, 1.5]$



Si tenemos un algoritmo que corre a un tiempo de $L[n, 1, c]$ diremos que corre en **tiempo polinomial,** si el algoritmo tiene una complejidad de $L[n, 1, c]$, entonces diremos que corre a un **tiempo exponencial,** y si $0 < v < 1$, diremos que el algoritmo corre a un **tiempo subexponencial.**

Ejemplos del tiempo que corren algunos algoritmos conocidos son:

Algoritmo: Criba de campos numéricos general

Tiempo: $L[n, 1/3, c_0 + o(1)], c_0 = (\frac{64}{9})^{1/3}$

Algoritmo: Criba de campos numéricos especial

Tiempo: $L[n, 1/3, c_1 + 0(1)], c_1 = (\frac{32}{9})^{1/3}$

Algoritmo: Criba cuadrática

Tiempo: $L[n, 1/2, 1 + o(1)]$

Algoritmo : Método de fracciones continuas

Tiempo: $L[n, 1/2, c_2 + o(1)]$, donde $c_2 = \sqrt{2}$

Algoritmo : Método de Factorización con Curvas Elípticas

Tiempo: $L[p, 1/2, \sqrt{2}]$, para encontrar el factor p y si $n = pq$, el tiempo es $L[n, 1/2, 1]$

2.2.1 Tiempo de factorización en la práctica

Hoy en día, parece ser que el problema de factorización ha progresado enormemente, este gran impulso lo ha provocado en gran parte la criptografía y más concretamente el sistema **RSA**. Enseguida nos proponemos a dar los datos que se tienen del crecimiento sobre la factorización de números enteros y estimar cual sería el tiempo en que se tardará la humanidad en factorizar un número entero considerado hoy imposible de factorizar.
Aunque se muestre que un algoritmo corre a tiempo subexponencial o más aún a tiempo polinomial, el tiempo de calendario (tiempo real en meses, días o minutos) puede variar mucho dependiendo de la entrada del algoritmo y del equipo de cómputo con que se cuenta. Existe una forma de medir el potencial de cómputo y de esta forma determinar cuál sería el tiempo en calendario que llevaría factorizar un número entero "grande" y cuándo podría llevarse a cabo esto.

La notación "mips" significa millones de instrucciones por segundo ([43]) y es tomada como medida estándar para registrar la potencia de cómputo con que se cuenta. Un mips tiene como origen la potencia de cómputo que realizaba una DEC VAX 11/780. Un mipsy significa el número de instrucciones que se pueden realizar en un año con un poder de cómputo de un mips, es decir 31 536 x 10^9 instrucciones.

Algunos datos interesantes son los mips que han sido utilizados para factorizar algunos números enteros.

Año	mipsy	Longitud
1994	0.001	45d
1984	0.1	71d
1984	5000	129d

Tabla 1

donde nd significa un número n entero de d dígitos.

La primera estimación que haremos es sobre el poder de cómputo que se puede conseguir en todo el mundo, el cual se estima que en 1994 era de 3x 10^8 mipsy, de los cuales un 10% estaba conectado a Internet y como se puede observar en la tabla 1 se utilizo un 0.0033% total para factorizar un número de 129d. Se estimó poder reunir 100 veces más poder de cómputo, es decir, usar un 0.33% del total en un trabajo de factorización.

Hagamos una estimación del poder disponible estimado en todo el mundo usando la predicción conocida por "ley de Moore". Gordon Moore co-fundador de Intel, predice en 1965 que la rapidez con que crecen los microprocesadores se duplica en 18 meses. Es decir, si en 1994 el poder de cómputo en el mundo era de 3x 10^8 mips, entonces en 3 años (1997) seria de 1.2x 10^9 mips, lo cual no está muy lejos de la realidad.

En la siguiente tabla mostramos la estimación de los próximos años:

Año	mipsy
1996	3×10^8
95-96	6×10^8
1997	1.2×10^9
98-99	2.4×10^9
2000	4.8×10^9
01-02	9.6×10^9
2003	1.92×10^{10}
04-05	3.84×10^{10}
2006	7.68×10^{10}

Año	mipsy
07-08	1.53×10^{11}
2009	3.07×10^{11}
10-11	6.14×10^{11}
2012	1.2×10^{12}
13-14	4.9×10^{12}
2015	9.8×10^{12}
16-17	2×10^{12}
2018	2×10^{13}
1920	4×10^{13}

Año	mipsy
2021	8×10^{13}
22-23	1.6×10^{14}
2028	10^{15}
2034	10^{16}
2038	10^{17}
2043	10^{18}
2049	10^{19}
2053	10^{20}
2058	10^{21}

Para poder determinar cuándo se podría contar con el poder necesario para factorizar un número entero de tamaño conocido debemos de saber qué poder de cómputo es necesario utilizar para tal factorización.
Esto lo estimaremos de la siguiente manera: partimos de que sabemos cuántos mipsy son utilizados para factorizar un número entero, por ejemplo un número de 129 d ~ 429 bits se puede factorizar con 1000 my, entonces para poder factorizar un número de 512 bits necesitaremos de T_{512} , donde T_{512} lo obtenemos de la siguiente fórmula:

$$T_n = T_{n-1} \bullet L[n] / L[n-1]$$

donde T_{n-1} es el poder de cómputo requerido para un número de menos bits y $L[n], L[n-1]$ son las respectivas complejidades que toma la criba de campos numéricos general, así por ejemplo para saber el poder de cómputo requerido para factorizar un número entero de 512 bits procedemos como sigue:

$$T_{512} = 1000 \bullet L[2^{512}] / L[2^{429}] = 29369.2 \sim 3 \times 10^4$$

Enseguida mostramos la estimación del poder de cómputo requerido para factorizar un número entero de la longitud respectiva pensando en usar la criba de campos numéricos general, ya que el número entero es aleatorio.

Dígitos	Bits	mipsy	Aprox.
129	429		1000
154	512	29369.2	3×10^4
192	640	2.990×10^6	3×10^6
231	768	1.800×10^8	2×10^8
269	896	7.331×10^9	7×10^9
308	1024	2.198×10^{11}	2×10^{11}
346	1152	5.150×10^{12}	5×10^{12}
385	1280	9.835×10^{13}	9×10^{13}
423	1408	1.580×10^{15}	2×10^{15}
462	1536	2.189×10^{16}	2×10^{16}
500	1664	2.666×10^{17}	3×10^{17}
539	1792	2.898×10^{18}	3×10^{18}
577	1920	2.849×10^{19}	3×10^{19}
616	2048	2.558×10^{20}	3×10^{20}

Con estos datos podemos; de un modo más fundamentado, estimar la seguridad de las llaves utilizadas en **RSA**, es decir, el poder usar una llave de longitud predeterminada sin arriesgar la seguridad, al menos evitando el ataque de la factorización (que es el ataque más seguro conocido hasta hoy).

A partir de las observaciones anteriores mostramos en la siguiente tabla el año donde es posible pensar que la longitud de la llave señalada puede ser rota:

Año límite	Longitud de llave
1997	768
2003	896
2012	1024
2018	1152
2022	1280
2034	1408
2038	1536
2043	1664
2049	1794
2053	1920
2058	2048

El año límite se refiere al año en donde el sistema debe comenzar a evaluar el cambio de la longitud de la llave a una longitud superior.

La anterior estimación no contempla varios aspectos que son muy difíciles de predecir como: el descubrir un método brillante que factorice números enteros en tiempos reducidos y colapsé al sistema **RSA,** el descubrir un nuevo tipo de tecnología que aumente la rapidez exponencialmente de las computadoras por lo que reduciría a gran escala el tiempo de factorización, u otro suceso que reduzca a gran escala el tiempo de factorización. Es prudente decir que la anterior estimación fue tomada suponiendo que el crecimiento tanto de rapidez de las computadoras como de la eficacia de los algoritmos, se comportará como hasta hoy; que pareciera no ser de otra forma. Así mismo no se considera el descubrimiento de otro ataque a **RSA** que no tenga que ver con la factorización, pero elimine el interés de su desarrollo así como también factores no previstos.

Resultado 1: La longitud de las llaves que se usan en el sistema **RSA** en el año de 1998, deben de ser de 1024 bits, donde los primos tengan casi la misma longitud, esto proporciona una seguridad de al menos 15 años.

Lo anterior nos sugiere que todas las etapas del sistema **RSA** deben de considerar el tamaño de 1024 bits, enseguida damos el detalle de alguna de ellas.

3. Generación de números primos de tamaño 1024 bits

Esta parte la dedicaremos a revisar el método que tendrá como salida números primos de una longitud que permita a la llave ser de 1024 bits, de hecho, los mismos algoritmos pueden ser usados para generar primos de mayor longitud con el respectivo gasto de tiempo.
Antes de entrar de lleno a la discusión sobre el cómo verificar que un número es primo, hablaremos brevemente de la generación de números pseudoaleatorios, cosa de no menor importancia, ya que de esto dependen fuertemente varios aspectos de seguridad en el sistema.

3.1. Generación de números pseudoaleatorios

La principal característica de un número aleatorio, es que su probabilidad de ser elegido es la misma probabilidad que tiene cualquier otro número dentro de un intervalo, además la elección debe ser independiente. Sin embargo, existe una enorme discusión sobre la aleatoriedad real, es decir, cómo podemos estar seguros o cómo se puede demostrar que algún dispositivo es aleatorio, en realidad poco podríamos conseguir si nos restringimos al concepto puro de aleatorio. Aunque gran parte de la probabilidad aplicada está basada en la existencia de eventos aleatorios, por ejemplo, si se quiere encontrar la preferencia de algún producto o partido político se procede a realizar una encuesta, la cual no se realiza a toda la población sino que se elige una muestra de tamaño considerable para que el intervalo de confianza sea satisfactorio, pues bien, la herramienta estadística debe de suponer que la muestra tomada es aleatoria, de otra forma los resultados estarán muy lejos de la realidad, para lo cual existen métodos que permiten acercarse a que la muestra considerada sea lo más aleatoria posible evitando las características más comunes de no aleatoriedad. De esta forma podemos llegar al concepto de pseudoaleatoriedad.

Diremos entonces que un dispositivo genera números pseudoaleatorios si en lo posible evita las principales características de no aleatoriedad, por ejemplo, si nuestro dispositivo generara una cadena de 10 dígitos binarios, y en circunstancias normales nos da una salida de a=0000000001, entonces podríamos pensar que existe una mayor probabilidad de que el dispositivo genera al 0 que al 1, por lo que tener una mayor cantidad de ceros que de unos ó una mayor cantidad de unos que de ceros será una característica de no aleatoriedad. Se pueden encontrar varias características de éstas y tratar de evitarlas (ver por ejemplo [40]), existe también una prueba que detecta la mayor parte de estas características llamado el método Universal de Maurer [38], una explicación muy entendible de esto se puede encontrar en [3].

La forma práctica de generar números pseudoaleatorios para propósitos criptográficos es la siguiente: podemos decir que tenemos dos etapas y una prueba, las dos etapas consisten en primero generar un primer número pseudoaleatorio que llamaremos semilla y la segunda una cadena de números pseudoaleatorios que provienen de una función de un sólo sentido, en la práctica se usan **MD5, SHA-1,** la función **RSA**, etc., la prueba consiste en verificar que la salida del dispositivo sea pseudoaleatoria con una prueba como la de Maurer.

Como resultado de estas consideraciones podemos concluir lo siguiente:

Resultado 2: Para generar la semilla de un dispositivo pseudoaleatorio basado en software se deben de usar eventos que estén muy alejados de la intervención humana; por ejemplo, el sistema de reloj, el teclado, el ratón, el contenido de buffers de entrada y salida, o una combinación de los anteriores.

Resultado 3: Para generar la cadena pseudoaleatoria se debe hacer uso de funciones de un sólo sentido como funciones Hash: **MD5, SHA-1,,** la función **RSA,** la función logaritmo en curvas elípticas, etc.

Resultado 4: Para probar que un dispositivo genera números pseudoaleatorios, éste debe de pasar una prueba que detecte las propiedades más conocidas de no aleatoriedad, por ejemplo la prueba de Maurer.

3.2. Pruebas de primacidad

En esta sección hablaremos de la forma para probar que un número de 1024 bits de longitud es primo o compuesto, en el transcurso de las pruebas se efectuarán operaciones sobre números de esta longitud y se entenderá que el problema para efectuar dichas operaciones está solucionado, algunos algoritmos al respecto se pueden encontrar en ([40], [42], [52], [60], [61]).

En la literatura sobre el tema existe una gran variedad de términos que se relacionan con números primos, nosotros nos restringimos sólo aquellos que tienen importancia en la criptografía.

Comenzaremos a dar los pasos a seguir para probar que un número es primo en los algoritmos criptográficos que hacen uso de ellos ([9], [10]).

Prueba de primacidad con propósitos criptográficos

Paso 1: Genera un número impar aleatorio m
Paso 2: Este número se somete a la prueba de la división.
Paso 3: Si el número pasa la prueba del paso 2, se somete a una prueba probabilística de primacidad.
Paso 4: Si el número pasa la prueba del paso 3, se declara a este número como "primo".
Paso 5: Si el número m no pasa 3, se considera a $m + 2$, y regresamos al paso 2.

Llamaremos a los números que pasan la anterior prueba pseudoprimos. En lo que sigue nos dedicaremos a proporcionar cómo se debe de hacer para que la prueba sea eficiente y confiable.

Justificación: paso2

Esta prueba consiste en dividir el número entero m por números primos pequeños que se mantiene en un arreglo de longitud constante. Primero justificaremos la aplicación de esta prueba.

Esta prueba es conocida como la criba de Eratosthenes, que consiste en dividir a m por cada número primo menor a $\sqrt{m}$, y si resulta que ningún primo de estos es factor de m, entonces m es primo. Es obvio observar que como m es de considerable longitud, entonces esta prueba es inaplicable.

Sin embargo probaremos que ayuda considerablemente a la prueba total. Aunque el algoritmo solicita de entrada que el número m sea impar, podemos suponer que no lo sabemos, para propósitos de explicación la única característica de *m* es que es aleatorio. La justificación final se basa en los siguientes hechos:

La probabilidad de que un número m sea dividido por 2 es de $\frac{1}{2}$, ya que la mitad de los números menores a m son pares y la otra mitad son impares. La probabilidad de que un número m sea divido por 3 es de $\frac{1}{3}$. Otra forma de verlo es que existen 3 clases de equivalencia módulo 3 y la probabilidad de que m esté en la clase del cero es $\frac{1}{3}$. Análogamente la probabilidad de que m sea divisible por el número primo es $p < \sqrt{m}$ es $\frac{1}{p}$.

De lo anterior podemos observar que si a un número entero aleatorio lo sometemos a la división de 2, hay una posibilidad del 50% que m tenga el factor 2, así como una probalbilidad de más del 30% que m tenga el factor 3, es decir, que hay una probabilidad de $\frac{1}{2}\left(1-\frac{1}{3}\right)=\frac{1}{3}$, o el 33.3% que m no tenga como factor ni al 2 ni al 3, o dicho de otra forma, que se descarta el 66.7% de los números generados al someterlos por la prueba de la división con los primos 2 y 3, por lo tanto, ya que el gasto en tiempo de cómputo es casi despreciable en la prueba de la división, es muy recomendable esta prueba para aligerar en gran porcentaje el consumo de tiempo con la prueba probabilística de primacidad.

En el caso concreto de generadores de números aleatorios criptográficos es muy común que el número generado sea impar, por lo tanto, la prueba da comienzo en el 3 en la siguiente tabla. Damos valores de $r(B)$ donde B es el número de primos con que cuenta la prueba de la división, empezando en 3, y $r(B)$ es probabilidad de que m no sea divisible por ningún primo entre $\{P_i = 3, 5, 7, \ldots, P_B\}$, es decir:

$$r(B) = \prod_{i=1}^{B} (1 - \frac{1}{P_i})$$

por lo tanto existe un $(1 - r(B))$ de porcentaje de números aleatorios impares descartados.

B	r(B)
100	0.177
200	0.157
300	0.147
400	0.141
500	0.136
600	0.133
700	0.130
800	0.128
900	0.126
1000	0.124

B	r(B)
1100	0.123
1200	0.122
1300	0.120
1400	0.119
1500	0.118
1600	0.117
1700	0.117
1800	0.116
1900	0.115
2000	0.114

B	r(B)
2100	0.114
2200	0.113
2300	0.113
2400	0.112
2500	0.112
2600	0.111
2700	0.111
2800	0.110
2900	0.110
3000	0.109

La forma de poder calcular el mejor B, o en su defecto una cota b, tal que $p_1 < b \; \forall i < B$, la podemos encontrar en [37] y consiste esencialmente en calcular $b = \frac{D}{E}$, donde D es el tiempo que tarda la máquina con el algoritmo de exponenciación que se usa y E es el tiempo en que se tarda la máquina en intentar una división entre m y un primo considerado pequeño. Actualmente en la práctica, se usa la tecnología de software y de hardware un B entre 1000 y 2000.

Resultado 5: Es necesario someter a los números aleatorios impares generados en aplicaciones criptográficas (como en el sistema **RSA**) a la prueba de la división con primos "pequeños", ya que esto permite eliminar entre un 80% y 90% de los números generados y así someter a otras pruebas más lentas un menor número de candidatos a primos.

Justificación: paso 3

Después de que un número m generado aleatoriamente pasa la prueba de la división con pequeños primos, se le somete a otra prueba más rigurosa, por lo tanto más lenta, sin embargo el candidato ya tiene una ligera aproximación para ser un probable primo. Antes de revisar los algoritmos más conocidos sobre pruebas de primacidad, veamos qué tantos candidatos a primo podemos hechar mano, esto nos permitirá entre otras cosa calcular la probabilidad de eventos que nos son de mucha utilidad.

Si la longitud de la llave n es de 1024 bits, entonces los números primos p, q deben de ser de 512 bits, es decir, estamos hablando de aproximadamente primos de 154 dígitos del orden de 10^{154}, si suponemos por el teorema del número primo ([39]) que hay x / lnx números primos en [2,x], entonces tenemos aproximadamente ($2^{512} / ln 2^{512}$) - ($2^{508} / ln 2^{508}$) = 3.54 x 10^{151} números primos de 512b. Para dar una idea del orden que estamos considerando podemos comparar este número con el dato que aparece en [31], donde se estima que el número aproximado de moléculas en todo nuestro universo es de 10^{80}, es decir, estamos hablando de un número que podemos considerar "denso" o "infinito" en la práctica de hoy en día.

Por lo tanto, el porcentaje de números primos que tenemos aquí es de (3.54 x 10151) / (6.2849 x 10153) = 0.0056325 donde 6.2849 x 10^{153} = (2^{512} - 2^{508}) / 2, , los números de longitud 512b no considera aquéllos que tienen menos de un dígito hexadecimal (4 bits), además son impares y como $0.005632 \approx \frac{1}{177}$ podemos suponer que en 177 intentos encontraremos a un número pseudoprimo.

Sin embargo, ya que el número generado aleatoriamente, además de ser impar, se somete a la prueba de la división por primos pequeños y obtenemos los siguientes resultados:

B	$\frac{\pi(2^{512})}{2^{512}\gamma(B)}$	$\sim \frac{1}{N}$
100	0.03183	$\frac{1}{32}$
200	0.03589	$\frac{1}{28}$
500	0.04143	$\frac{1}{24}$
1000	0.04544	$\frac{1}{22}$
1500	0.04775	$\frac{1}{21}$
2000	0.04934	$\frac{1}{21}$
2500	0.05031	$\frac{1}{20}$
3000	0.05170	$\frac{1}{19}$

Si en la tabla de primos pequeños es de 100, sólo hay que probar con la prueba probabilística 32 para poder encontrar un primo, de la misma forma, si en la tabla tenemos 1000 números primos sólo hay que probar a 22 para obtener el primo requerido, por lo que es recomendable elegir una buena cota B para mejorar la eficacia al generar números primos.

Una de las partes más sensibles del sistema criptográfico **RSA** es el algoritmo probabilístico de números primos, de hecho, una buena elección de éste garantiza en gran parte el buen funcionamiento del sistema. En lo siguiente abundaremos particularmente en la prueba de Fermat y la de Miller-Rabin, aunque mencionaremos algunas otras. Tendremos especial atención en justificar que el uso de la prueba de Miller-Rabin con 7 testigos alcanza los niveles comerciales más exigentes de exactitud. Incluso justificaremos que la prueba de Fermat con testigo 2 es aceptable a niveles menos pretenciosos pero de igual forma es considerablemente eficaz y es usado en software libre o de muestra ([27]).

3.3 Prueba de primacidad de Fermat

Quizá la forma más conocida de sospechar que un número es primo, es aplicar la identidad conocida por el Teorema pequeño de Fermat y consiste en la siguiente afirmación:

Si n es un número primo y $1 \leq a \leq n-1$ entonces $a^{n-1} \equiv 1 (\mathrm{mod}\, n)$, en efecto esto lo cumple cualquier número primo, es decir, si encontramos a un número a y un número n tal que no cumpla la igualdad, entonces n es compuesto, sin embargo, la afirmación recíproca no se cumple, es decir, existen números que cumplen la igualdad y no son primos. En resumen estrictamente hablando, el método de Fermat no da el 100% de certeza para poder encontrar números primos.

Uno de los métodos que se ha estudiado ampliamente ha sido el de Fermat, su simplicidad lo justifica y aunque no se recomienda su uso en aplicaciones donde se requiere la mejor exigencia de seguridad, este método es usado en aplicaciones con un mínimo de pretensiones. A pesar de lo anterior en las siguientes líneas damos algunos datos interesantes sobre el mismo.

En [53] da a conocer un experimento sobre la generación de números aleatorios y las consecuencias que se obtuvieron al someter a estos números a la prueba de Fermat con testigo 2, en comparación con una prueba teóricamente más fuerte (la de Miller-Rabin), de 718 millones de números aleatorios de 256 bits, 43 741 404 pasaron la prueba de la división con primos pequeños con cota 10^4. De éstos 4 058 000 dieron válida la igualdad de Fermat y aún más, todos los últimos se sometieron a la prueba de Miller-Rabin con 8 iteraciones y ninguno reprobó.

Aunque para lo anterior aún no existe una completa justificación, hay varios resultados que apoyan en gran medida los resultados obtenidos. Por ejemplo si un número n, cumple la igualdad de Fermat para un a dado, siendo n compuesto, a n se le llama pseudoprimo base a. En [47] y [48] se estudia y conjetura que el número de estos pseudoprimos menores a n no pasan de la cota $L(n) = \exp\left(\frac{\log n \log_3 n}{\log_2 n}\right)$, por ejemplo en el caso anterior $L(2^{256}) = 3.9 \times 10^{52}$, en otras palabras hay una probabilidad de $\frac{1}{10^{11}}$ de encontrar un pseudoprimo. Además si $n \longrightarrow \infty$, entonces la probabilidad de aceptar a un número compuesto como primo converge a cero ([19]), es decir, mientras el número a considerar sea más grande, la probabilidad de falla disminuye.

Un problema teóricamente más serio son aquellos números que pasan la prueba de Fermat con cualquier testigo *a,* siendo *n* compuesto. A esta clase de números se les conoce con el nombre de números de Carmichael, la principal falla que se tiene en criptografía con esta clase de números es que si *n* fuera de Carmichael sirve para encriptar y desencriptar mensajes, ya que cada operación de este tipo es equivalente a una prueba de Fermat, y por lo tanto tendríamos manejando llaves **RSA** susceptibles para ser factorizables más fácilmente. Hoy en día se sabe que existe una cantidad infinita de primos de Carmichael, pero además también se sabe que en comparación a los primos, es una cantidad despreciable ([2], [47]). Por otro lado, aunque existen algunas sugerencias para detectar estos números de Carmichael ([45]), la mejor forma de evitarlos es implementar una prueba de primacidad probabilística más dura sin costo de programación considerable como lo es la prueba de Miller-Rabin.

Resultado 6: La prueba probabilística de Fermat es muy simple de implementar y la probabilidad de fallar es del orden de $1/10^{25}$ en comparación en la generación de primos de tipo **RSA** actual. Por lo tanto puede ser utilizada en dispositivos donde el costo de riesgo no se considere prioritario como programas de muestra, experimentales, etc., sin embargo en dispositivos que exijan requerimientos de seguridad aceptables para propósitos comerciales, financieros o de seguridad nacional es muy recomendable usar una prueba más dura como la de Miller-Rabin.

3.4. Prueba de primacidad de Miller-Rabin

La forma más simple de derivar la prueba de Miller-Rabin a partir de la prueba de Fermat es considerando cómo se efectúa la potencia $a^{n-1} \bmod n$, es decir, con $n - 1 = 2^r s$,

$n-1=2^r s$, si $a^s \equiv 1 \bmod n$ ó $a^{2^j s} \equiv -1 \bmod n$ para algún j, $0 \le j \le s-1$,

entonces $a^{n-1} \equiv 1 \bmod n$. Dicho de otra forma si a pasa la prueba de Miller-Rabin, a pasa la prueba de Fermat, cosa que al contrario no es válida, concretamente a causa de los primos de Carmichael.

Sin duda, la prueba de Miller-Rabin es la más popular para generar números primos de gran longitud en los sistemas criptográficos, esto es porque además de que su implementación no requiere de mucho costo, este método proporciona una gran probabilidad de certeza. En [29] se obtiene una cota superior para la probabilidad de que el método confirme a un número como primo siendo éste compuesto, a esta probabilidad de error la denotaremos como $P_{k,t}$, donde k es el número de bits del número que tratamos de probar su primacidad, y t es el número de iteraciones que son usadas en el algoritmo de Miller-Rabin para obtener la probabilidad $P_{k,t}$. Mientras pasa el tiempo estas cotas se han ido mejorando, de tal modo que a partir de la fórmula tomada de [13]

$$P_{k,t} \le (\pi(2^k) - \pi(2^{k-1}))^{-1} \sum_{n \in M_1} a(n)^t$$

donde $\pi(x)$ denota el número de primos menores a x, la suma es sobre los enteros compuestos y $\alpha(n) = \frac{S(n)}{(n-1)}$, con $S(n)$ el número de números enteros $a \in [1, n-1]$ tales que $a^u \equiv 1 \bmod n$ ó $a^{2^i u} \equiv -1 \bmod n$ para algún $i < s$, siendo $n - 1 = 2^s u$, usando que $\pi(2^k) - \pi(2^{k-1}) > (0.71867)\frac{2^k}{k}$ para $k \geq 21$, demostrada en [29] y usando resultados de [13] obtenemos algunos valores de $P_{k,t}$ en la siguiente tabla:

k\t	1	2	3	4	5	6	7	8	9	10
300	19	33	44	53	60	67	73	78	83	88
350	28	38	48	58	66	73	80	86	91	97
400	37	46	55	63	72	80	87	93	99	105
450	46	54	62	70	78	85	93	100	106	112
500	56	63	70	78	85	92	99	106	113	119
550	65	72	79	86	93	100	107	113	119	126
600	75	82	88	95	102	108	115	127	127	133

Donde $P_{k,t} \leq (\frac{1}{2^j})$ siendo j el valor correspondiente de la tabla.

Lo anterior nos lleva a nuestro siguiente resultado:

Resultado 7: Para generar números primos de tamaño 512 bits necesarios para generar llaves de 1024 bits, es recomendable usar el algoritmo de Miller-Rabin con 7 iteraciones, ya que por una parte evita los posibles errores sobre números de Carmichael y por otra, la probabilidad de que declare a un número compuesto como primo del orden de $1/2^{100}$.

De las pruebas probabilísticas que existen como la de Solovay-Strassen, se ha podido mostrar que el conjunto de primos que pasan la prueba está contenido en el conjunto de primos que pasa la prueba de Miller-Rabin, por lo que es recomendable no usarlas. Por lo que se refiere a las pruebas determinísticas, aunque éstas son más efectivas son mucho más lentas, por lo que sólo con una muy buena y creativa implementación podrían usarse para propósitos criptográficos.

4. Nota sobre la implementación

En los procesos tanto de generación de llaves, encriptamiento y desencriptamiento se realizan operaciones de suma, multiplicación, exponenciación, obtener inversos multiplicativos, etc., esto quiere decir, en términos de implementación y del tamaño de los números de 512 bits al menos, que es necesario diseñar la aritmética para tales números.

El primer problema con que nos encontramos es el cómo representar a números tan grandes ya que por ejemplo, la computadora PC actual sólo nos permite manejar números enteros de tamaño de 32 bits. La solución ha sido desarrollada ya desde hace varios años y le nombran números de múltiple precisión o de longitud grande. Actualmente, existen varios métodos que permiten realizar aritmética de múltiple precisión tanto implementados en software como en hardware ([26]), de esto depende en gran parte la rapidez del sistema por lo que es necesario encontrar las mejores formas de efectuar la aritmética de estos números, por ejemplo, se tiene una de las justificaciones que el sistema **RSA** use como exponente e a la constante $F_4 = 2^{2^4} + 1,$ ya que este número en su representación base 16, que es la más común para este tipo de uso, es 10001, lo que hace que la operación de exponenciación en cada encripción se efectúe lo más rápido posible. Más acerca del tema lo podemos encontrar en ([31],[40],[41]).

Resultado 8: La implementación del sistema **RSA** debe de hacerse con aritmética de múltiple precisión y usarse los algoritmos más rápidos.

5. PKSC #1

En los puntos anteriores comentamos brevemente el ataque más importante que se tiene en estudio, que es la factorización de números enteros y mencionamos algo muy somero sobre la implementación. Estas quizá sean las partes más estudiadas y desarrolladas en lo que se refiere al sistema **RSA**, sin embargo, hay mas aspectos importantes que es necesario entender y evitar cuando se requiera implementar el sistema **RSA**. A lo largo de la existencia de**RSA** se han podido desarrollar una buena cantidad de ataques, algunos son imprácticos, es decir, las condiciones que piden para que puedan ser efectivos en muchos casos no se pueden tener en la realidad, así como también algunas recomendaciones para una buena implementación.

La razón del título de esta sección es que el **PKSC # l** (Public Key Standar Cryptosystems) reúne las reglas que hay que tener en cuenta para que una implementación primero sea inmune a los diferentes ataques a que puede ser objeto, y por otro lado a que las diferentes implementaciones de **RSA** sean compatibles. Como el **PKSC # 1** existen otros estándares ([7], [25]) que vigilan lo mismo, y que pueden ser adoptados con los mismos propósitos.

Enseguida mencionaremos algunos de estos puntos.

5.1 Ataque de Wiener

Este ataque debido a Wiener ([64]) muestra cómo recobrar la llave privada d a partir de los parámetros públicos e y n. Este ataque hace uso de fracciones continuas y sólo es eficiente cuando d es pequeño en el sentido de que cumple $d < n^{1/4}$, es decir, si n es del orden de 1024 bits, entonces d debería de ser menor a 2^{256}, lo que lo hace impráctico para el valor que aquí consideramos. El motivo por el cual este ataque es anulado, es que se debe a que el número e sugerido es el número 4 de Fermat denotado por F_4, es decir:

$$F_4 = 2^{2^4} + 1 = 2^{16} + 1 = 65537_{10} = 10001_{16}$$

En este caso e tiene tamaño de 16 bits y como d es el número tal que $ed = 1 \bmod \phi(n)$ y además $\phi(n) = (p-1)(q-1)$ tiene tamaño del orden de 1024 bits aproximadamente, entonces d tiene tamaño del orden de 1008 bits. Lo que quiere decir que con el exponente sugerido en **PKCS #1** se evita este ataque.

5.2 El exponente de encripción *e* pequeño

Este ataque ([24]) hace referencia al tamaño del exponente e, éste esencialmente pide que un mensaje sea encriptado con diferentes llaves públicas pero con el mismo exponente e, entonces se puede recobrar a m usando el Teorema Chino del Residuo, más aún, se puede pedir que los mensajes estén relacionados de alguna forma conocida. Este ataque queda totalmente anulado al aplicar el**PKCS #1,** ya que éste pide que los bloques de texto original m sean ensuciados, es decir, como en general el sistema **RSA** se usa para encriptar llaves privadas de sistemas como el**TripleDES** o para firmar documentos que antes se les fue aplicada una función **Hash** (que reduce la longitud del mensaje m a una longitud constante). Entonces el mensaje m llega a tener alrededor de 160 bits, los restantes bits para completar un bloque que sea el que se encripte y tome el papel de m se rellena con bits aleatorios, a este proceso es al que llamaremos ensuciar el mensaje. Otros ataques que piden que el exponente e sea pequeño como por ejemplo $e = 3$ han sido publicados, como el de Franklin y Reiter ([20], [11]) sin embargo, también llegan a ser imprácticos ensuciando el mensaje.

5.3 Un ataque de texto original elegido

Si suponemos que un atacante quiere encontrar el contenido de un mensaje encriptado $c = m^e \bmod n,$ mandado al usuario A, y este usuario pudiera desencriptar mensajes para este atacante, entonces el atacante puede seleccionar un número $x \in Z_n^*$ y calcular $\bar{c} = cx^e \bmod n,$ y pedir a A, que desencripte este mensaje, o sea que calcule $\bar{m} = \bar{c}^d \bmod n,$ entonces como $\bar{m} = \bar{c}^d = c^d x^{ed} = mx \bmod n,$ el atacante puede recuperar fácilmente a m . Este ataque es también superado al imponer una estructura a los mensajes que se quieran desencriptar. En el caso del **PKCS #1**, sería muy difícil que el mensaje $\bar{c}$ tuviera esa estructura, ya que esto significa que el arreglo que representa a $\bar{c}$ tenga 0s exactamente donde dice el**PKCS #1,** el desarrollo de este tipo de ataques se puede encontrar en ([8], [14], [18]).

5.4 Ataque de la superencripción

No es muy difícil ver que si tenemos un texto encriptado $c = m^e \bmod n,$ lo podemos considerar como un elemento de Z_n^* ya que $(c, n) = 1$ con gran probabilidad. Por lo tanto debe existir un número k , tal que $c^k = m^{ek} = 1 \bmod n$, es decir, $m^{ek+1} = m \bmod n,$ sin embargo, esto es impráctico, más aún es equivalente a factorizar que el encontrar k es encontrar el orden de Z_n^* que a su vez es lo mismo que factorizar al número n , por lo que no es un ataque viable.

5. 5 Los primos p,q muy cercanos

Otra forma de pretender romper el sistema **RSA,** es tratar de factorizar a la llave pública n extrayendo la raíz cuadrada de n y enseñar como posibles factores primos a los primos cerca de la raíz. Es claro que esto sólo es efectivo si los factores primos de n estuvieran cerca, es decir, $p - q$ fuera pequeño. No es difícil probar que si el sistema **RSA** genera a los números primos aleatoriamente, la probabilidad de que esto suceda es despreciable para propósitos criptográficos, concretamente: para que pudiéramos considerar viable este tipo de ataque sería necesario que $p - q$ fuera mucho menor que $\sqrt{n},$ esto con el fin de compararlo con la complejidad de la criba de Eratosthenes, ya que $\sqrt{n} = 2^{512},$ y como la

probabilidad de que al menos dos números tengan la mitad de bits iguales es de $\frac{1}{2^{512}}$. Entonces el ataque es efectivo con una probabilidad de $\frac{1}{2^{k}}$ con k mucho mayor a 512, lo que hace al ataque no viable.

6.- Si el tamaño del texto original es pequeño

Una posibilidad más que se tiene para romper el sistema es si consideran mensajes de encripción pequeños, es decir, si $m^{e} < n$. Aunque esto también se puede tener cuando el mensaje sea pequeño y el exponente grande, pero por rapidez de la implementación se requiere que e sea pequeño, entonces esto lo tenemos solo cuando m es pequeño. Este ataque se efectúa simplemente extrayendo la raíz e-ésima. De la misma forma es evitado cuando ensuciamos el mensaje.

Resultado 9: Una gran parte de ataques y recomendaciones de implementación al sistema **RSA**son evitados y adaptadas respectivamente al asumir un estándar de implementación como el **PKCS #1**.

7 Los primos duros

Por último hablaremos de los llamados primos duros y su utilidad en el sistema **RSA.** Un primo p es duro si $p-1$ tiene un factor primo r grande, $0<r<1$ que tenga un factor primo grande y que $r-1$ tenga un factor primo grande, para propósitos criptográficos podemos considerar que un primo grande es de alrededor de 100 bits.

El solicitar que los primos para el **RSA** sean duros, pretende evitar que el producto $n = pq$ pueda ser factorizado por métodos que piden factores pequeños en $p \pm 1$ y $r-1$, como el método de Pollar y Williams, sin embargo, primos duros no evitan el Método con Curvas Elípticas el cual se considera que éste puede ser limitado su éxito con la longitud de los primos grandes y su aleatoriedad. Respecto a los primos duros existen dos opiniones que no difieren considerablemente para propósitos criptográficos.

La primera que sustenta **RSA Data Security** es que no son necesarios los primos duros ([4]), ya que éstos no evitan el **MCE,** y el considerar a primos grandes y aleatorios evita en la mayoría de métodos sobre factorización más potentes y por lo tanto en gran probabilidad a los métodos de Pollar y Williams. La segunda opinión es que en efecto, los argumentos de la primera opinión son acertados sin embargo el añadir condiciones a la generación de los primos para que éstos sean duros no ofrece ninguna dificultad ni tiempo extra considerable ([23], [58]), así que es sugerida la utilización de primos duros.

7 Notas finales

Para terminar este artículo comentaremos otros aspectos importantes sobre el sistema **RSA**. El sistema **RSA** ha sido uno de los más estudiados hasta el momento y por lo tanto se considera que es uno de los más seguros, ya que ha podido superar todo tipo de controversia, así que es por hoy uno de los sistemas criptográficos de llave pública más usados en la industria, en el comercio, en los gobiernos, en la milicia y en general en toda actividad que requiera que su información tenga un alto grado de seguridad criptográfica. Es bueno decir que hasta hoy se han desarrollado una gran cantidad de sistemas de llave pública con el fin de substituir, generalizar o simplemente competir con **RSA,** sólo que no han tenido gran éxito, en principio deben de pasar un riguroso criptoanálisis por parte de la comunidad criptográfica y después se someten a la competencia comercial, la prueba es en general proporcionar al menos la misma seguridad de los sistemas existentes con al menos la misma facilidad de implementación y después que basen su seguridad en problemas muy duros. Hasta hoy solo los sistemas basados en el **P**roblema del **L**ogaritmo **D**iscreto **E**líptico **(PLDE) (www.certicom.com)** han podido competir exitosamente con el sistema **RSA**, incluso son más prometedores que **RSA** ya que con sólo llaves de 160 bits proporcionan la misma seguridad que **RSA**. Existe otro tipo de sistemas que basan su seguridad en el **PFE,** sin embargo, se ha demostrado que son equivalentes a **RSA** por lo que se desechan, es decir, necesitan la misma o mayor longitud de las llaves que **RSA**.

En el sitio de la compañía **RSA** [56] se puede encontrar buena parte de información sobre la actualización del sistema, así como otro tipo de información relevante. Es importante mencionar que actualmente se esta actualizando el **PKCS #1**, y se espera a fin de año tener el documento nuevo.

Esperamos que en este documento se encuentre información valiosa que pueda fundamentar algunas de las características importantes del sistema **RSA** que son dadas por obvias en otros documentos. Resumimos y finalizamos con una bibliografía que no pretende ser completa pero si es suficiente para justificar aquellos resultados que se mencionaron a lo largo del artículo.

Resultados

Resultado 1: La longitud de las llaves que se usan en el sistema **RSA**, en el año de 1998, deben de ser de 1024 bits, donde los primos tengan casi la misma longitud, esto proporciona una seguridad de al menos 15 años.

Resultado 2: Para generar la semilla de un dispositivo pseudoaleatorio basado en software se deben de usar eventos que estén muy alejados de la intervención humana, por ejemplo, el sistema de reloj, el teclado, el ratón, el contenido de buffers de entrada y salida, o una combinación de los anteriores.

Resultado 3: Para generar la cadena pseudoaleatoria se debe de hacer uso de funciones de un solo sentido como las funciones Hash: **MD5, SHA-1,...,** la función **RSA**, la función logaritmo en curvas elípticas, etc.

Resultado 4: Para probar que un dispositivo genera números pseudoaleatorios, éste debe de pasar una prueba que detecte las propiedades más conocidas de no aleatoriedad, por ejemplo la prueba de Maurer.

Resultado 5: Es necesario someter a los números aleatorios impares generados en aplicaciones criptográficas (como en el sistema **RSA**) a la prueba de la división con primos "pequeños", ya que esto permite eliminar entre un 80% y 90% de los números generados y así someter a otras pruebas más lentas un menor número de candidatos a primo.

Resultado 6: La prueba probabilística de Fermat es muy simple de implementar y la probabilidad de fallar es del orden de $1/10^{25}$, en la generación de primos de tipo **RSA** actual. Por lo tanto puede ser utilizada en dispositivos donde el costo de riesgo no se considere prioritario, como programas de muestra, experimentales, etc., sin embargo en dispositivos que exijan requerimientos de seguridad aceptables para propósitos comerciales, financieros o de seguridad nacional es muy recomendable usar una prueba más dura como la de Miller-Rabin.

Resultado 7: Para generar números primos de tamaño 512 bits necesarios para generar llaves de 1024 bits, es recomendable usar el algoritmo de Miller-Rabin con 7 iteraciones ya que por una parte evita los posibles errores sobre números de Carmichael y por otro la probabilidad de que declare a un número compuesto como primo es del orden de $1/2^{100}$

Resultado 8: La implementación del sistema **RSA** debe de hacerse con aritmética de múltiple precisión y usar algoritmos más rápidos.

Resultado 9: Una gran parte de ataques y recomendaciones de implementación al sistema **RSA** son evitados y adaptados respectivamente, al asumir un estándar de implementación como el **PKCS #1,** en su última versión.

Bibliografía

1 W. Alexi, B. Chor, O. Goldreich, C. P. Schonorr, **RSA and RABIN Functions: Certain Parts are as Hard as the Whole,** *SIAM J. Comput. Vol.* 17 *No. 2, pp.* 194-209, 1988.

2 W.R. Alford, A. Granville, C. Pomerance, **There are Infinitely Many Carmichael Numbers,** *Annals of Mathematics Vol,* 140, *pp.* 703-722, 1994.

3 J.J. Angel A., **Generación de Números Pseudoaleatorios usados en Sistemas Criptográficos,** *CryptoNotas Vol.* 1, *No.* 1, 1998.

4 An RSA Laboratories Seminar, **RSA Key Generation and Strong Primes** *RSA Laboratories, RSA Data Security, Inc.,* 100 *Marine Parkway City, CA* 94065, 1995.

5 F. Arnault, **Rabin-Miller Primality Test: Composite Numbers Which Pass It,** *Mathematics of Computation Vol.* 64, *No.* 209, *pp.* 355-361, 1995.

6 D. Atkins, M.GraA; A.K. Lenstra, P.C. Leyland, **The Magic Words are Squeamish Ossifrage,** *Adv. in Crypto. ASIACRYPT94, LNCS* 917, pp.263-277, 1994.

7 M. Bellare, P. Rowaway, Optimal Asymetric Encryption, Adv. in Crypto, EUROCRYPT94, pp. 92-111, 1995.

8 D. Bleichenbacher, M.Joye, J. Quisquater, A **New and Optimal Chosen-Message Attack on RSA-Type Cryptosystems,** *Information and Communications Security, LNCS* 1334, *pp,* 302-313, 1997.

9 J. Brandt, I. Damgard, **On the Generation of Problable Primes by Incremental Search,** *Adv, in Crypto. CRYPTO92, LNCS* 740, *pp* 358-370.

10 J. Brandt, I. Damgard, P. Landrock, **Speeding up Prime Number Generation,** *Adv, in Crypto. ASL4CRYPP91, LNCS* 739, *pp.* 440-449, 1993.

11 D. Coppersmith, M. Franklin, J. Patarin, M. Reiter, **Low-Exponent RSA with Related Messages,** *Adv. in Crypto. EUROCRFPP96, LNCS* 1070, *pp,* 1-9, 1996.

12 J. Cowie, B. Dodson, R.M. Elkenbracht-Huizing, A.K. Lenstra, P. L. Montgomery, J. Zayer, **A World Wide Number Field Sieve Factoring Record: On to 512 Bits,** *Adv. in Crypto. ASIACRYPP96, pp.* 382-394, 1996.

13 I. Damgárd, P. Landrock, C. Pomerance, **Average Case Error Kstimates for the Strong Problable Prime Test,** *Mathematics of Computation Vol.* 61, *No. 203, pp.* 177-194, 1993.

14 G.I. Davida, **Chosen Signature Cryptanalysis of the RSA Public Key Cryptosystem,** *Thechnical Report TR-CS-82-2, Department of Electrical Engineering and Computer Science, University of Wisconsin, Mitwaukee,* 87, 1982.

15 J.M. DeLaurentis, A **Further Weakness in the Common Modulus Protocol for the RSA Crypto Algorithm,** *Cryptologia Vol.* 8, *pp* 253-259, 1984.

16 D.E. Denning, **Digital Signatures with RSA and others Public-Key Cryptosystems,** *Communications of the ACM Vol. 27, pp.* 388-392, 1984.

17 T. Denny, B. Dodson, A,K. Lenstra, M.S. Manasse, **On the Factorization of RSA-120,** *Adv, in Crypto. CRYPTO93, LNCS 773, pp.* 166-174, 1994.

18 Y. Desmedt, A.M. Odlyzko, A **Chosen Text Attack on the RSA Cryptosystem and Some Discrete Logarithm Schemes,** *Adv. in Crypto. CREPO85, LNCS* 218, *pp.* 516-522, 1986.

19 P. Erdos, C. Pomerance, **On The Number of False Witnesses for a Composite Number,** *Mathematical of Computation Vol.* 46, *No.* 173, *pp.* 259-279, 1986.

20 M.Franklin, M.Reiter, **A Linear Protocol Failure for RSA with Exponent Three,** *Rump Session of Crypto95, Santa Barbara, CA.*

21 J. Gerver, **Factoring Large Numbers with a Quadratic Sieve,** *Mathematics of Computation, Vol.* 41, *pp.* 287-294, 1983.

22 J.S. Greenfield, **Distributed Programming Paradigms with Cryptography Applications,** *LNCS* 970, 1994.

23 J. Gordon, **Strong Primes are Easy to Find,** *Adv. in Crypto. EUROCRFPT84, LNCS* 209, *pp.* 216-223, 1984.

24 J. Hastad, **Solving Simultaneous Modular Equations of Low Degree,** *SL4M Journal on Computing,* 17(2) *pp.* 336-341, 1988.

25 ISO/IEC, **International Standar 9796:** *Information Technology, Security Techniques: Digital Signature Scheme Giving Message Recovery,* 1991.

26 K. Iwamura, T. Matsumoto, H. Imai, **High-speed Implementation Methods for RSA Scheme,** *Adv. in Crypto. EUROCRYPT92, I.NCS* 658, *pp.* 221-238.

27 B.S. Kaliski, **How RSAs Toolkits Generate Primes,** *Tech. Report* 003-903028-100-000-000, *RSA Laboratories, Aedwood City, CA* 1994.

28 B.S. Kaliski, M. Robshaw, **The Secure Use of RSA,** *CryptoBytes Vol.* 1, *No. 3,* 1995, *pp.* 7-13.

29 S.H. Kim, C. Pomerance, **The Probability that a Random Probable Prime is Composite,** *Mathematics of Computation Vol,* 53, *No.* **188,** *pp* 721-741, 1989.

30 N. Koblitz, **A Course in Number Theory and Cryptography,** *Springer Verlag GTM* l l4, 1987.

31 C.K. Koc, **High Speed RSA Implementation,** *RSA Laboratories, RSA* ***Data Security, Inc.,*** 100 *Marine Parkway City, CA* 94065, 1995.

32 P.C. Kocher, **Timing Attacks on Implementations of Diffie-Hellman,** RSA, **DSS, and Others Systems,** *Adv. in Crypto. CRYPTO96, LNCS* l l09, *pp.* 104-113, 1996. *Mathematics of Computation Vol.* 53, *No.* **1SS,** *pp.* 721-741, 1989.

33 A.K. Lenstra, H.W. Lenstra Jr., (eds.) **The Development of the Number Field Sieve,** *LNM* 1554, *Springer Verlag, Berlin,* 1993.

34 A.K. Lenstra, H.W. Lenstra Jr., M.S. Manasse, J.M. Pollar, **The Factorization of the Ninth Fermat Number,** *Mathematics of Computations Vol.* 61, *pp.* 319-349, 1993.

35 H.W. Lenstra Jr., C. Pomerance, A **Rigorous Time Bound for Factoring Integers,** *Journal of the American Mathematical Society, Vol. 5,* pp.483-516, 1992.

36 H.W. Lenstra Jr., **Factoring Integers with Elliptic Curves,** *Ann. ofMath. Vol.* 126, *pp.* 649-673.

37 U.M. Maurer, **Fast Generation of Secure RSA-Muduli with Almost Maxima Diversity,** *Adv. in Crypto, EUROCAYPT89, pp.* 636-647, 1990.

38 U.M. Maurer, A UniveRSAl Statistical **Test for Random Bit Generators,** *Adv. in Cryp. CRIPTO90 LNCS* 537, *pp* 409-420, 1991.

39 G.L. Miller, **Riemanns Hypothesis and Tests for Primality,** *Journal of Computer and Systems Sciences Vol.13 pp.* 300-317, 1976.

40 A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, **Handbook of Applied Cryptography,** *CRC Press, Inc.Boca Raton Florida* 1997.

41 C. Mitchell, A. Selby, **Algorithms for Software Implementation of RSA,** *IEE Procrrdings, Vol.* 136 *No.3,* 1989.

42 L. Monier, **Evaluation and Comparison of Two Kfficient Probabilistic Primality Testing Algorithms,** *Theoretical Computer Science Vol.* 12, *pp.* 97-108, 1980.

43 A.M. Odlyzko, **The Future of Integer Factorization,** *July* 11, 1995.

44 A.M. Odlyzko, **On the Complexity of Computing Discrete Logarithms and Faetoring Integers,** *Bell Laboratories, Murray Hill, New Jersey* 07974

45 R.G.E. Pinch, **On Using Carmichael Numbers for Public Key Encryption Systems,** *Cryptography and Coding, LNCS* 1355, *pp* 265-269, 1997.

46 J.M. Pollard, **Theorems on Factorization and Primality Testing,** *Proceedings of the Cambridge Philosophical Society,* 76, *pp* 512-528, 1974.

47 C. Pomerance, J.L. Selfüdge, S.S. Wagstaff Jr., **The Pseudoprimes to 25x10,** *Mathematics of Computation, Vol.* 35 *No.* 151, 1980, *pp.* 1003-1026.

48 C. Pomerance, **On the Distribution of Pseudoprimes,** *Mathematics of Computation Vol. 37 No.* 156, 1981, *pp.* 587-593.

49 C. Pomerance, **The Quadratic Sieve Factoring Algorithm,** *Adv. in Crypto. EUROCRF7"* 84, *LNCS* 209, *pp.* 169-182.

50 C. Pomerance, **Factoring,** *Proceedings of Symposia in Applied Mathematics Vol.* 42, 1990.

51 C. Pomerance, **The Number Field Sieve,** *Proceedings of Symposia **in** Applied Mathematics Vol.* 48, 1994.

52 P. Ribenboim, **The Little Book of Big Primes,** *Springer Verlag* 1991

53 R.L. Rivest, **Finding Four Million Large Random Primes,** *Adv. in Crypto CRYPTO90, LNCS* 537, 1990, *pp.* 625-626.

54 R.L.Rivest,A.Shamir,L.Adleman, **A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,** *Communication of the ACM Vol.* 21 *No. 2,* 1978, *pp.* 120-126.

55 **RSA** Laboratories, **The Public-Key Cryptography Standards,** 1993.

56 **RSAhttp://www.RSA.com** Data Security,

57 Y. Sakai, K. Sakurai, H. Ishizuka, **On Weak RSA-Keys Produced from Pretty Good Privacy**, *Information and Communications Security, LNCS* 1334, *pp.* 314.324, 1997.

58 R.D. Silverman, **Fast Generation of Random, Strong RSA Primes,** *CryptoBytes Vol. 3, No.* 1, *pp.* 9-13, 1997.

59 G.J. Simmons, **A Weak Privacy Protocol Using the RSA Crypto Algorithm,** *Cryptologia Vol. 7, pp.* 180-182, 1983.

60 R. Solovay, V. Strassen, **A Fast Monte-Carlo Test for Primality,** *SL4M Journal on Computing, Vol.* 6, pp.84-85, 1977.

61 D.R. Stinson, Cryptography, **Theory aad Practice, *CRC Pess,*** *Inc, Boca Raton Florida* 33431, 1995.

62 T. Takagi, **Fast RSA-Type Cryptosystems Using N-Adic Expansion,** *Adv. in Crypto. CRYPTO97, LNCS* 1294, *pp.* 372-384, 1997.

63 S.A. Vanstone, R.J. Zuccherato, **Short RSA Key and Their Generation,** *Dept. of Combinatorics and Optimization University of JYaterloo, Ontario N2L, Canada,* 1995.

64 M. J. Wiener, **Cryptanalysis of Short RSA Secret Exponents,** *IEEE Transaction on Information Theory, Vol.* 36 *No.* 3, *pp.* 553-558, 1990.

65 H.C. Williams, **A Modification of the RSA Public-Key Encryption Procedure,** *IEEE Transactions on Information Theory, Vol. IT-26 No.*