Generación de primos: una perspectiva computacional

R. Durán Díaz
Universidad de Alcalá,
Alcalá de Henares, España
e-mail: raul.duran@uah.es

L. Hernández Encinas
Instituto de Física Aplicada, CSIC,
Madrid, España
e-mail: luis@iec.csic.es

J. Muñoz Masqué
Instituto de Física Aplicada, CSIC,
Madrid, España
e-mail: jaime@iec.csic.es

***Resumen*—En este trabajo presentamos un resumen de los principales métodos utilizados para la generación de primos con particular énfasis en las optimizaciones desarrolladas para dispositivos móviles, que suelen disponer de prestaciones computacionales limitadas. Se presentan, además, los resultados de una implementación en Maple del método que consideramos más optimizado junto con resultados experimentales de su rendimiento.**

I. Introducción

En los últimos años hemos asistido a una expansión sin precedentes en el uso de la criptografía de clave pública, incluso en el ámbito de las relaciones institucionales, como es el caso del DNI electrónico y del pasaporte electrónico. Estos documentos y otros muchos necesitan utilizar números primos (generalmente de gran tamaño) como elementos básicos para generar las claves o para otros estadios del protocolo criptográfico. Resulta, pues, de interés revisar el estado del arte en lo que se refiere a los métodos para generar primos.

Es también un hecho notable la enorme difusión que han conseguido dispositivos móviles o portátiles, pequeños pero dotados de recursos computacionales más o menos amplios, según los casos. Desde teléfonos móviles o PDAs, con alta capacidad de cómputo y de memoria, hasta tarjetas inteligentes, generalmente mucho más modestas en uno y otro recursos, todos ellos son ejemplos de sistemas en donde encontramos con frecuencia capacidades criptográficas. Es necesario diseñar algoritmos de generación de primos apropiados a estos dispositivos, en cuanto a demanda de computación y de almacenamiento.

En este trabajo presentamos un resumen de los principales métodos utilizados para la generación de primos, en dos aplicaciones muy populares, GnuPG y OpenSSL, y en dispositivos móviles con prestaciones computacionales limitadas. Incluimos, además, resultados experimentales.

El trabajo se estructura así. La sección II repasa los algoritmos de primalidad deterministas y probabilísticos, con especial énfasis en el algoritmo de Miller-Rabin. Las secciones III y IV presentan y analizan datos experimentales sobre la generación de primos en dos ámbitos diferenciados. Por un lado se consideran las aplicaciones GnuPG y OpenSSL, sobre plataformas estándar; y por otro, se estudia el algoritmo de Joye-Paillier sobre plataformas móviles. Finalmente, las conclusiones se presentan en la sección V.

II. Algoritmos de detección de primalidad

Es sabido que decidir sobre la primalidad de un número es una tarea mucho más sencilla que factorizarlo, aunque los métodos de comprobación de primalidad implican gran carga computacional. Ordinariamente se basan en comprobar el cumplimiento, por parte del candidato a primo, de ciertas condiciones, cuya verificación implica su primalidad. Con frecuencia se recurre a métodos que exigen comprobaciones menos costosas, pero que, por contra, no dan con toda seguridad una respuesta correcta, es decir, pueden declarar que un número es primo cuando en realidad es compuesto.

Así pues, se llama *test de primalidad* (resp. *test de composición*) a un algoritmo que determina que un candidato es primo (resp. compuesto). Un test de primalidad decide con total seguridad acerca de la primalidad del candidato. Sin embargo, el test de composición sólo es capaz de determinar con toda seguridad que el candidato es compuesto. Por esta razón, a los primeros se les añade el calificativo de *deterministas*, mientras que los segundos se denominan *tests de primalidad probabilísticos*.

II-A. Algoritmos deterministas

El trabajo de Agrawal *et al.* ([1]) estableció que determinar la primalidad de un número admite un algoritmo de primalidad determinista de tiempo polinómico, es decir, el problema de la primalidad es de tipo **P**. Sin embargo, el interés de este algoritmo es más bien teórico, pues su tiempo de ejecución es de $\tilde{O}\left((\log n)^{21/2}\right)$ en el caso más pesimista, aunque puede mejorar a $\tilde{O}\left((\log n)^{6}\right)$ aceptando ciertas conjeturas habituales. La notación $\tilde{O}(x)$ significa $O(x\cdot \mathrm{pol}(\log x))$, donde $\mathrm{pol}(x)$ es un polinomio en x.

Este algoritmo fue rápidamente mejorado por otros autores, entre los que destacan Berrizbeitia ([2]) que baja ese tiempo a $\tilde{O}\left((\log n)^{4}\right)$ para aquellos números primos n para los cuales n^2-1 es divisible por una potencia de 2 cercana a $(\ln n)^2$. Cheng ([3], [4]) extendió ese resultado a una clase más amplia de números primos n, aquellos tales que $n-1$ es divisible por un primo $e \simeq (\log n)^2$. Bernstein ([5]) generaliza el resultado a valores $e \simeq (\log n)^2$ que dividan a n^d-1, con $d \in n^{o(1)}$ (en la práctica, lo interesante es el caso $d=1$). Independientemente, Avanzi y Mihailescu en [6] generalizaron totalmente el valor e.

Todos ellos son algoritmos de tipo aleatorio que proporcionan un tiempo de cómputo esperado de $\tilde{O}\left((\log n)^4\right)$.

Aunque obtener estos resultados ha supuesto un notable esfuerzo, el tiempo de computación de tales algoritmos los descarta de un uso práctico y hay que recurrir a los algoritmos probabilísticos.

II-B. Algoritmos probabilísticos: test de Miller-Rabin

El test de Miller-Rabin es el algoritmo universalmente utilizado para producir los llamados «primos industriales». Este algoritmo cae en la categoría de test de composición, por lo que existe una probabilidad no nula de que el algoritmo califique de «no compuesto» un número que realmente lo es. Sin embargo, el test permite reducir esa probabilidad a un valor tan pequeño como se desee con una eficiencia computacional alta.

El teorema de Fermat es la base de éste y de otros algoritmos probabilísticos de comprobación de primalidad y afirma que si n es primo y a es un entero tal que $\mathrm{mcd}(n,a)=1$, entonces $a^{n-1}\equiv 1 \pmod n$. Si se desea comprobar la primalidad de n y se encuentra una base prima con n en que el teorema de Fermat no se verifica, se puede asegurar que n es compuesto. Ahora bien, la recíproca no es cierta: aunque todas las posibles bases verifiquen el teorema, eso no garantiza que el número n sea primo. A pesar de ello, el test de Fermat en sí mismo es extraordinariamente útil por varias razones:

1. La condición del teorema es sólo necesaria, pero el número de excepciones (*pseudoprimos*) es muy bajo.
2. Desde el punto de vista computacional depende casi totalmente de la *exponenciación modular*.

Los números n que satisfacen la congruencia $a^{n-1}\equiv 1 \pmod n$ para toda base $a\in[2,n-1]$ tal que $\mathrm{mcd}(a,n)=1$ se conocen como *números de Carmichael*. Por desgracia, existen infinitos de ellos (ver [7]), pero esto no impide seguir usando el teorema de Fermat como se verá ahora.

Si n un número entero impar positivo y a otro entero, se escribe $n-1=2^s q$, con q otro entero impar. En estas condiciones, se dice que n es un *pseudoprimo robusto* en la base a si o bien $a^q\equiv 1 \pmod n$ o bien existe un e tal que $0\le e<s$ y $a^{2^e q}\equiv -1 \pmod n$.

Observación 1: Si p es un primo impar, es fácil ver que también p es un pseudoprimo robusto en cualquier base a tal que $\mathrm{mcd}(a,p)=1$. Recíprocamente, se puede probar (véase, por ejemplo, [8]) que si p no es primo, existen menos de $p/4$ bases a tales que $1<a<p$ para las cuales p es un pseudoprimo robusto en la base a.

Suponiendo que las probabilidades son independientes, es claro que si un candidato resulta pseudoprimo robusto para t bases aleatorias, la probabilidad de que sea realmente primo será[1] $1-2^{-2t}$. Con esto, Miller ([10]) y Rabin ([11]) desarrollaron el test que lleva su nombre y podemos enunciar como sigue.

[1]Esta probabilidad es demasiado pesimista, al no tener en cuenta la distribución de los primos. Puede verse un análisis más detallado en [9, §§4.48, 4.49], que proporciona valores adecuados para el parámetro de seguridad de acuerdo a la probabilidad deseada. En dicho análisis se demuestra también que el parámetro depende además de la longitud en bits del candidato analizado.

ENTRADA:	$n\in\mathbb{Z}$, parámetro de seguridad $t\in\mathbb{N}$.
SALIDA:	n es compuesto o primo probable con probabilidad $1-2^{-2t}$.

1. [Inicialización]
   Se eligen s, $q\in\mathbb{Z}$ tales que $n-1=2^s q$, q impar.
2. [Lazo]

```
while (t > 0)
{  Elegimos un entero a ∈ [2, n-1] aleatoriamente.
   b = a^q (mód n);
   if (b == 1  o bien  b == n-1) goto seguir;
   for (j ∈ [1, s-1])
   {  b = b^2 (mód n);
      if (b == n-1) goto seguir;
      if (b == 1) return "n es compuesto";
   }
   return "n es compuesto";
seguir:
   t = t - 1;
}
return "n es primo con probabilidad 1 - 2^(-2t)";
```

Figura 1. Algoritmo de Miller-Rabin

Algoritmo 2: Dados $n\in\mathbb{Z}$, candidato a comprobar, y $t\in\mathbb{N}$, parámetro de seguridad, este algoritmo determina si n es primo con una probabilidad de acierto de $1-2^{-2t}$. El algoritmo está descrito en la figura 1.

Es importante ahora tratar del tiempo de ejecución de este algoritmo, por estar en el corazón de los generadores de primos utilizados en la práctica. De la observación del algoritmo, en la figura 1 queda claro que el tiempo de ejecución viene dominado por el necesario para llevar a cabo la operación de exponenciación modular. Existen muchos algoritmos para la exponenciación (véanse, por ejemplo, [12, §1.2], [13, Cap. 9], [14, §9.3]) que requieren ordinariamente $O(\log n)$ multiplicaciones en el grupo. Por tanto es fundamental tratar de elegir el método de multiplicar que minimice el tiempo de computación.

Los algoritmos de multiplicación más sencillos requieren $O\left((\log n)^2\right)$ operaciones básicas, entendiendo por tales las que involucran dígitos de un tamaño máximo, digamos B, en bits. Existen otros métodos, sin embargo, que permiten reducir este número.

1. Método de Karatsuba ([15]). Divide los números a multiplicar en dos partes más pequeñas, reduciendo 4 multiplicaciones a 3. Así, el número de operaciones básicas pasa a ser $O\left((\log n)^{\log 3}\right)\simeq O\left((\log n)^{1,585}\right)$.
2. Método de Toom-Cook ([14, §9.5]). Divide los números a multiplicar en k partes más pequeñas (si $k=2$, se convierte en Karatsuba). Cuando, por ejemplo, $k=3$, el número de multiplicaciones pasa de 9 a 5, con lo que el número de operaciones básicas es $O\left((\log n)^{\log 5/\log 3}\right)\simeq O\left((\log n)^{1,465}\right)$.

3. Métodos que involucran transformadas de Fourier, como el de Schönhage y Strassen ([16]) que necesita un número de operaciones de $O\left(n \log n \log^2 n\right)$.

Todos los métodos citados necesitan realizar aparte las reducciones modulares, lo que implica costosas divisiones. Ello se evita usando la reducción de Montgomery ([17], [18]), un ingenioso método de realizar la reducción modular, que evita las divisiones. Este método resulta de mucho interés para los dispositivos móviles por su economía, pero ha sido objeto de intensos ataques de canal lateral como el *timing attack* (véase, por ejemplo, [19]). Se concluye con la siguiente

Proposición 3: El tiempo de ejecución esperado para el algoritmo de Miller-Rabin es $O\left((\log n)^{2+\varepsilon}\right)$.
El valor de ε depende del método de multiplicación empleado.

III. Generadores en GnuPG y OpenSSL

En esta sección estudiamos los generadores utilizados en dos importantes aplicaciones del mundo del software abierto: GnuPG y OpenSSL. Hemos elegido estas dos aplicaciones por acogerse a las licencias de tipo *GNU General Public License*.

GnuPG es la implementación hecha por GNU de OpenPGP, tal como está definido en el RFC4880. Entre las operaciones soportadas está, naturalmente, la generación de claves, que necesita como base una primitiva de generación de números primos. Obviamente, los algoritmos usados para la generación impactan en la calidad de los primos generados y, por tanto, en la seguridad básica del sistema.

OpenSSL es un proyecto para desarrollar en código abierto una implementación de los estándares Secure Sockets Layer (SSL) y Transport Layer Security (TLS), que constituyen la base más usada actualmente para la comunicación segura a través de internet. Protocolos como `https`, para navegación web con autenticación y cifrado, aplicaciones como `ssh` para comunicación serie (terminal serie) cifrada y autenticada, son clientes de tales estándares. También es interesante en este caso estudiar y cualificar la calidad de las primitivas de generación de primos, base para la generación de las claves.

III-A. Generación en GnuPG

GnuPG usa GMP, *GNU Multiple Precision Arithmetic Library* ([20]) como biblioteca de multiprecisión[2], modificada ligeramente en cuanto al modo de almacenar los datos.

La generación de números primos está centralizada en una sola función denominada `gen_prime` y contenida en el fichero `cipher/primegen.c` de la distribución.

El proceso de generación implica dos fases: la generación de un número aleatorio candidato y la comprobación de primalidad mediante tests sucesivamente más fiables y costosos computacionalmente. En todo el proceso, se utiliza una lista estática de números primos hasta e incluyendo 4999, que se prepara antes de comenzar el proceso.

[2]Las bibliotecas de multiprecisión recogen rutinas que permiten utilizar datos de precisión arbitraria (por ejemplo, enteros de tamaño arbitrariamente grande, o números de coma flotante con precisión arbitrariamente pequeña) para lenguajes de programación estándar, como C o C++. En particular, GMP es una biblioteca de código abierto, avalada por una amplísima base instalada.

1. Fase de generación del candidato:
   a) Genera un número aleatorio. Para ello utiliza los recursos del sistema operativo; en particular, para los sistemas tipo Unix/Linux, el dispositivo `urandom`.
   b) Asegura que los bits más y menos significativos sean '1'. Así garantiza un determinado número de bits y que el candidato es impar.
2. Fase de comprobación de primalidad:
   a) Realiza un test de divisiones sobre la tabla de primos bajos.
   b) Realiza un test de Fermat.
   c) Realiza un test de Miller-Rabin, con un parámetro de seguridad fijado en 5 para todos los candidatos.

Si en alguno de los tests de la fase 2 el candidato resulta ser compuesto, se repiten los tests sobre el siguiente impar, hasta probar un total de 10.000. Si todos los candidatos resultan compuestos, se vuelve a la fase 1, se genera un nuevo número aleatorio y se repite todo el proceso.

El algoritmo de exponenciación usado es el clásico de «potencia cuadrada y producto repetidos» ([12, §1.2]) junto con una reducción por simple división con resto. La multiplicación usa el método de Karatsuba si resulta más eficiente.

III-B. Generación en OpenSSL

OpenSSL utiliza una biblioteca propia, incluida en el propio paquete, como biblioteca de multiprecisión. Esta biblioteca se denomina BIGNUM y está escrita en lenguaje C.

En este caso, la generación de primos está centralizada en la función `BN_generate_prime_ex`, contenida en el fichero `crypto/bn/bn_prime.c`. Los clientes de esta función son, por ejemplo, el generador de parámetros para intercambio de claves de Diffie-Hellman, o la generación de claves para RSA.

En este caso, también la generación de primos comporta dos fases: la generación de un número aleatorio y la comprobación posterior mediante un algoritmo de primalidad.

1. Fase de generación del candidato:
   a) Genera un número aleatorio, utilizando funciones de la propia biblioteca, basadas *grosso modo* en la familia MD de funciones resumen (*hash*).
   b) Dentro de la misma fase, comprueba también que el número aleatorio así generado no contenga ningún factor común con alguno de los primos de una tabla de primos estáticamente generada. Dicha tabla está prefijada en el fichero `crypto/bn/bn_prime.h` y alberga los 2048 primeros primos, desde 2 hasta e incluyendo 17863.
2. Fase de comprobación de primalidad:
   a) Obtiene un parámetro de seguridad dependiente del tamaño del candidato, con el fin de garantizar una probabilidad menor de 2^{-80} para cualquier número de bits que tenga el candidato. Específicamente, se elige 3 para los candidatos de 1024 bits, 6 para los de 512 bits y 12 para los de 256 bits (véase [9, Tabla 4.4]).

b) Se realiza un test de Miller-Rabin con el parámetro de seguridad seleccionado en el punto anterior.

Como dato de interés, la biblioteca utiliza la exponenciación de Montgomery, implementada en la función `BN_mod_exp_mont`.

III-C. Resultados experimentales

En esta sección presentamos los resultados experimentales acerca de los tiempos de computación y el número de llamadas al algoritmo de primalidad que han sido necesarios para obtener primos de diversas longitudes utilizando para ello los generadores respectivos de GnuPG y de OpenSSL.

La metodología de trabajo ha sido la misma para las medidas correspondientes a ambas aplicaciones. A continuación, resumimos los pasos dados para las mediciones.

1. Aislar las rutinas que generan los primos en cada aplicación.
2. Crear un programa principal como *envoltorio* capaz de generar un primo de la longitud deseada.
3. Armar una batería de tests que generen 200 números primos de longitudes desde 100 hasta 1000 bits, con saltos de 100 bits, registrando en cada caso los datos de tiempo de ejecución y número de llamadas al algoritmo de primalidad.

Las longitudes se han seleccionado teniendo en cuenta los valores que pueden ser de interés hoy en día. Ello no obstante, los resultados parecen ser fácilmente extrapolables. Para la ejecución de los programas, se ha utilizado una plataforma de tipo Intel Pentium M, a 1,60 gHz, con un tamaño de caché de 2048 kbytes y 1 gbyte de memoria RAM.

Los resultados experimentales se pueden ver en las figuras 2-3. La figura 2 representa el tiempo de ejecución necesario, medido en milisegundos y representado logarítmicamente, frente al número de bits requerido. Hemos representado el promedio de tiempos resultado de la batería de tests ejecutada para cada una de las aplicaciones, de modo que se pueda ver una comparación entre ambas al golpe de vista. Los resultados son muy similares si bien se ve que OpenSSL consigue una

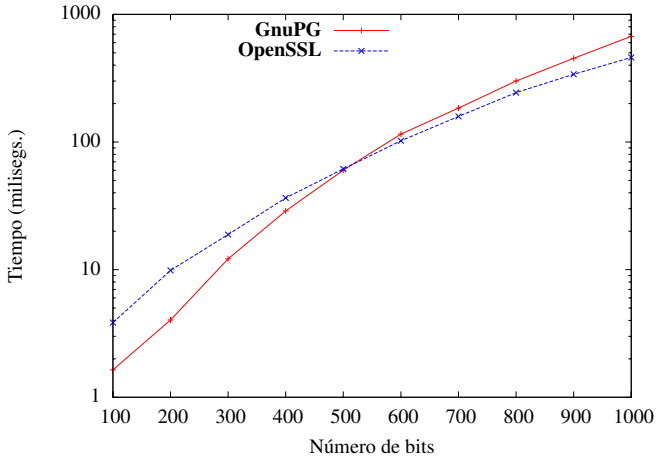

Figura 2. Tiempo de ejecución promediado frente a número de bits

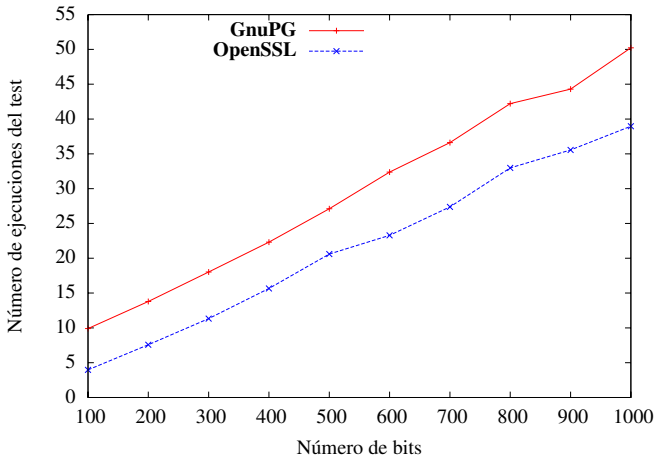

Figura 3. Número promediado de invocaciones al algoritmo de primalidad

pendiente más suave lo que le favorece para el cómputo de números primos más grandes.

La figura 3 representa el promedio de invocaciones al algoritmo de primalidad. Aquí claramente la ventaja es para OpenSSL, que consigue obtener primos con menos invocaciones. Ello se debe a que los candidatos presentados por OpenSSL al algoritmo son de «mejor calidad», es decir, tienen más probabilidades de ser realmente primos. Sin duda esto explica su buen comportamiento en términos de tiempo de ejecución, tal como se acaba de ver en la figura anterior.

IV. Generación de primos en dispositivos móviles

Después de repasar los métodos habituales de generación de primos en plataformas estándar, nos centramos en el problema de hacer lo mismo sobre dispositivos móviles, tales como tarjetas inteligentes, PDAs, etc. Este tipo de dispositivos se caracteriza por su limitación tanto en capacidad de proceso (que es nula en algunos casos) como en capacidad de almacenamiento. Ello hace inviables los métodos habituales y hay que recurrir a métodos especializados que permitan generar primos eficientemente en este tipo de plataformas. Tal es el caso del método de Joye y Paillier ([21], [22]).

La propuesta citada es capaz de producir primos q uniformemente distribuidos en un intervalo prefijado, $[q_{min}, q_{max}]$, donde q_{min} y q_{max} son dos enteros arbitrarios, $q_{min} < q_{max}$. Se supone que el dispositivo está dotado de un generador de números aleatorios y de una función de comprobación de primalidad T. El objetivo es maximizar la velocidad de generación, básicamente reduciendo el número de aplicaciones del test T, que es lo más costoso computacionalmente. Veamos a continuación las distintas fases del algoritmo.

Selección de los parámetros del sistema. Tomamos $0 < \varepsilon \leq 1$ como un parámetro de calidad (por ejemplo, $\varepsilon = 10^{-3}$) y sea φ la función de Euler. Se elige un un conjunto de primos y se calcula $\Pi = \prod_i p_i$, tal que existan enteros t, v, w, que satisfacen

(P1) $1 - \varepsilon < \frac{w\Pi - 1}{q_{max} - q_{min}} \leq 1$;
(P2) $v\Pi + t \geq q_{min}$;

ENTRADA: $v, w, t \in \mathbb{Z}$, y $a \in \mathbb{Z}_m^* \setminus \{1\}$.
SALIDA: primo aleatorio en el intervalo $[q_{min}, q_{max}]$.

1. [Inicialización]
   Calcular $\ell = v\Pi$, $m = w\Pi$.
   Seleccionar aleatoriamente $k \in \mathbb{Z}_m^*$.
   $q = [(k-t) \pmod m] + t + \ell$;
2. [Lazo]
   `while (`$\mathsf{T}(q)$ `== false)`
   { $k = k \cdot a \pmod m$;
   $q = [(k-t) \pmod m] + t + \ell$;
   }
   `return` q;

Figura 4. Generación de primos de Joye-Paillier

(P3) $(v+w)\Pi + t - 1 \leq q_{max}$;
(P4) el cociente $\varphi(\Pi)/\Pi$ es lo más pequeño posible.

Los primos generados están en el intervalo $[v\Pi + t, (v+w)\Pi + t - 1] \subseteq [q_{min}, q_{max}]$. De acuerdo a (P1), cuanto más pequeño sea ε, tanto mejores resultados se obtienen. En (P4), minimizar el cociente $\varphi(\Pi)/\Pi$ garantiza que Π maximiza el número de primos distintos lo más pequeños posible. Dados $(q_{min}, q_{max}, \varepsilon)$, calcular los valores de (Π, v, w, t) que satisfagan las propiedades (P1)-(P4) es experimentalmente sencillo.

Generación de primos. El algoritmo se puede ver en la figura 4. Es de destacar que el primer paso necesita un valor aleatorio $k \in \mathbb{Z}_m^*$, por lo que hace falta un algoritmo que permita la selección computacionalmente simple de unidades. Este algoritmo se presenta en la figura 5. También es importante destacar que tanto a como k están en el grupo de las unidades $\mathbb{Z}_m^*$, por lo que siempre son coprimos con Π. Aquí radica la genialidad del algoritmo: es capaz de generar candidatos a primos que excluyen, *por construcción*, una lista tan grande como se quiera y pueda de factores primos (pequeños). Con esto disminuye proporcionalmente el número de veces que se ha de invocar el test de primalidad antes de obtener un primo.

Generación de unidades. La clave para el buen comportamiento del algoritmo es la posibilidad de generar fácilmente elementos $k \in \mathbb{Z}_m^*$. Para ello, se utilizan los siguientes resultados (demostrados o referenciados en [21]).

ENTRADA: m y $\lambda(m)$, con λ función de Carmichael.
SALIDA: unidad aleatoria $k \in \mathbb{Z}_m^*$.

1. [Inicialización]
   Seleccionar aleatoriamente $k \in [1, m]$.
   $u = (1 - k^{\lambda(m)}) \pmod m$;
2. [Lazo]
   `while` $(u \neq 0)$
   { Seleccionar aleatoriamente $r \in [1, m]$.
   $k = k + ru \pmod m$;
   $u = (1 - k^{\lambda(m)}) \pmod m$;
   }
   `return` k;

Figura 5. Generación de unidades de Joye-Paillier

Proposición 4: Sean $m > 1$ y $k \in \mathbb{Z}_m$. Entonces, $k \in \mathbb{Z}_m^*$ si y sólo si $k^{\lambda(m)} \equiv 1 \pmod m$, donde λ es la función de Carmichael.

Recordemos que la función de Carmichael de un número n se define como el entero más pequeño, $\lambda(n)$ tal que $a^{\lambda(n)} \equiv 1 \pmod n$ para todo $a \in \mathbb{Z}_n$ tal que $\mathrm{mcd}(a, n) = 1$. Si $n = p_1^{s_1} \cdots p_t^{s_t}$, entonces se puede calcular recursivamente como $\lambda(n) = \mathrm{mcm}(\lambda(p_1^{s_1}), \ldots, \lambda(p_t^{s_t}))$. A su vez, si $p \geq 3$, o $s \leq 2$, $\lambda(p^s) = \varphi(p^s) = p^{s-1}(p-1)$, y $\lambda(2^s) = 2^{s-2}$.

Proposición 5: Sean $k, r \in \mathbb{Z}_m$, tales que $\mathrm{mcd}(r, k, m) = 1$. Entonces $k + r(1 - k^{\lambda(m)}) \pmod m \in \mathbb{Z}_m^*$.

Está claro que calcular la función de Carmichael de m, $\lambda(m)$, es fácil porque, por construcción, sabemos perfectamente la factorización de m.

IV-A. Resultados experimentales

Para obtener resultados de utilización del algoritmo de Joye-Paillier no hemos podido contar con dispositivos físicos, por lo que hemos recurrido a su implementación en el sistema Maple de álgebra simbólica, que es muy popular y proporciona todas las primitivas necesarias para ello. No adjuntamos el código por falta de espacio.

La metodología ha sido similar a la utilizada para las aplicaciones GnuPG y OpenSSL: el programa implementado en Maple es capaz de generar un primo de la longitud pedida, reportando el tiempo necesario para ello y el número de llamadas al algoritmo de primalidad. Es importante notar que los sistemas de cómputo simbólico se ejecutan a una velocidad relativamente lenta, por lo que los resultados de tiempo de computación son interesantes sólo desde un punto de vista relativo. Con la ayuda de este programa, hemos realizado una batería de tests para generar primos en el intervalo aproximado desde 100 a 1000 bits, en pasos de aproximadamente 100 bits. Para cada longitud se generan 500 primos y se registra el tiempo de computación y el número de llamadas al algoritmo de primalidad necesarios para obtenerlos. Finalmente se calcula el promedio de ambas medidas.

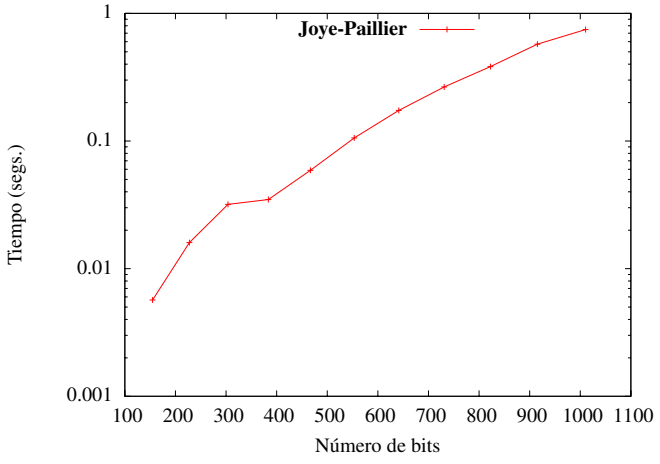

Figura 6. Tiempo de ejecución promediado frente a número de bits

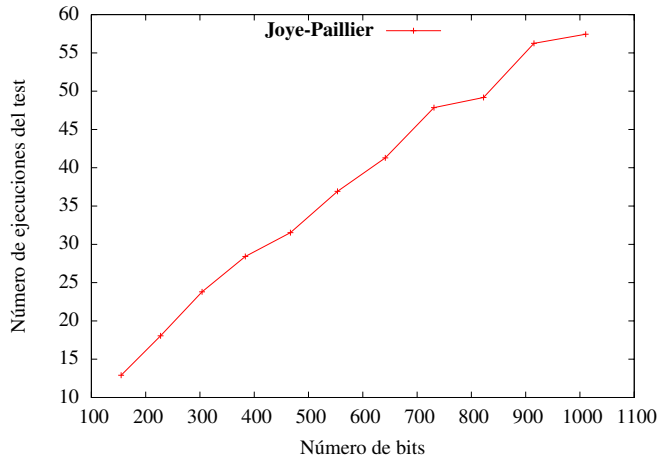

Figura 7. Número promediado de invocaciones al algoritmo de primalidad

Los resultados pueden verse en las figuras 6-7. En la figura 6 se presenta el tiempo medio de computación. Se observa que, en términos relativos, el comportamiento es relativamente similar al que se ha obtenido para las aplicaciones GnuPG y OpenSSL. Como ya se ha indicado, no fue posible realizar estas mediciones sobre dispositivos móviles reales, por lo que nos hemos de limitar a señalar el aspecto de la curva. No obstante pensamos que no es atrevido esperar un comportamiento similar.

En la figura 7 se puede observar que, en general, el número de llamadas al algoritmo de primalidad resulta ser más alto para todo el rango de tamaño en bits si se compara con los resultados obtenidos en la figura 3. Esto resulta en detrimento de este algoritmo que precisamente buscaba minimizar el número de tales llamadas. Ello parece indicar que el buen comportamiento del algoritmo resulta muy dependiente de una correcta elección de los parámetros del sistema.

V. Conclusiones

En este trabajo hemos presentado un resumen de los principales métodos utilizados para la generación de primos en diferentes plataformas. Se ha analizado experimentalmente la eficiencia de los métodos utilizados por dos aplicaciones muy populares, GnuPG y OpenSSL, así como por el algoritmo de Joye-Paillier. Los primeros pueden considerarse de carácter general, mientras que el último está especialmente diseñado para tarjetas criptográficas con limitada capacidad de cómputo.

Hemos diseñado programas específicos que aíslan el proceso de generación de primos para cada una de ellas y presentamos resultados experimentales acerca del tiempo necesario y número de invocaciones al algoritmo de primalidad en cada caso, considerando la generación de primos con distintas longitudes.

Los procesos de generación de primos, en todos los casos, pueden considerarse muy eficientes y los algoritmos empleados garantizan una alta calidad en el proceso de generación de primos. Este resultado es especialmente destacable en el caso del OpenSSL.

Aunque no ha sido posible realizar una implementación sobre dispositivos reales, los resultados experimentales aquí presentados parecen indicar que el algoritmo de Joye-Paillier supone una optimización sobre tarjetas criptográficas, tanto en tiempo como en recursos, en línea con las aplicaciones GnuPG y OpenSSL, destinadas a plataformas estándares.

Agradecimientos

Los autores agradecen a los revisores sus sugerencias para la mejora de este trabajo que ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación mediante el proyecto TEC2009-13964-C04-02, y el Ministerio de Industria, Turismo y Comercio, en colaboración con CDTI y Telefónica I+D mediante el proyecto Segur@ CENIT-2007 2004.

Referencias

[1] M. Agrawal, N. Kayal y N. Saxena, “PRIMES Is in P”, en *Ann. of Math.*, vol. 160, no. 2, pp. 781–793, 2004.
[2] P. Berrizbeitia, “Sharpening ‘Primes is in P’ for a large family of numbers”, en *Math. Comp.*, vol. 74, no. 252, pp. 2043–2059, 2005.
[3] Q. Cheng, “Primality proving via one round in ECPP and one iteration in AKS”, en *Lecture Notes in Comput. Sci.*, vol. 2729, pp. 338–348, 2003.
[4] ——, “Primality proving via one round in ECPP and one iteration in AKS”, en *J. Cryptology*, vol. 20, no. 3, pp. 375–387, 2007.
[5] D. J. Bernstein, “Proving primality in essentially quartic random time”, en *Math. Comp.*, vol. 76, pp. 389–403, 2003.
[6] P. Mihăilescu y R. Avanzi, “Efficient ‘quasi’-deterministic primality test improving AKS”, preprint, http://caccioppoli.mac.rub.de/website/papers/aks-mab.pdf
[7] W. Alford, A. Granville y C. Pomerance, “There are infinitely many Carmichael numbers”, en *Ann. of Math.*, vol. 140, pp. 703–722, 1994.
[8] D. Knuth, *The Art of Computer Programming*. Reading, MA, USA: Addison-Wesley Publishing Co., 1968, 1980, 2nd edition, vol. 2 - Seminumerical Algorithms.
[9] A. Menezes, P. van Oorschot y S. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1997.
[10] G. Miller, “Riemann’s hypothesis and tests for primality”, en *J. Comput. System Sci.*, vol. 13, pp. 300–317, 1976.
[11] M. Rabin, “Probabilistic algorithms for testing primality”, en *J. Number Theory*, vol. 12, pp. 128–138, 1980.
[12] H. Cohen, *A Course in Computational Algebraic Number Theory*. Berlin: Springer, 1993.
[13] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen y F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, H. Cohen, G. Frey y C. Doche, Eds. Boca Raton, FL, USA: Chapman & Hall/CRC, 2005.
[14] R. Crandall y C. Pomerance, *Prime Numbers. A Computational Perspective*. New York: Springer, 2001.
[15] A. Karatsuba, “The complexity of computations”, en *Proc. Steklov Inst. Math.*, vol. 211, pp. 169–183, January 1995.
[16] A. Schönhage y V. Strassen, “Schnelle Multiplikation großer Zahlen”, en *Computing (Arch. Elektron. Rechnen)*, vol. 7, pp. 281–292, 1971.
[17] P. L. Montgomery, “Modular multiplication without trial division”, en *Math. Comp.*, vol. 44, no. 170, pp. 519–521.
[18] Ç. K. Koç, T. Acar y B. S. Kaliski, Jr., “Analyzing and comparing Montgomery multiplication algorithms”, en *IEEE Micro*, vol. 16, no. 3, pp. 26–33, June 1996.
[19] H. Sato, D. Schepers y T. Takagi, “Exact analysis of Montgomery multiplication”, en *Lecture Notes in Comput. Sci.*, vol. 3348, pp. 290–304, 2004.
[20] T. Granlun, “The GNU multiprecision arithmetic library”, http://gmplib.org, 2010.
[21] M. Joye y P. Paillier, “Fast generation of prime numbers on portable devices: An update”, en *Lecture Notes in Comput. Sci.*, vol. 4249, pp. 160–173, 2006.
[22] M. Joye, P. Paillier y S. Vaudenay, “Efficient generation of prime numbers”, en *Lecture Notes in Comput. Sci.*, vol. 1965, pp. 340–354, 2000.