Mejoras en los tiempos de ejecución de funciones criptográficas gracias al conocimiento de la arquitectura del ordenador

Pedro María Alcover y José M. García

Resumen– **Muchos algoritmos criptográficos descansan en la dificultad de factorizar enteros grandes. No se conocen algoritmos que logren factorizar en un tiempo polinómico. Cualquier avance que logre reducir el tiempo de este proceso es bien valorado. Si tenemos en cuenta las arquitecturas de los microprocesadores podemos obtener códigos, en la implementación de algoritmos de factorización, más eficientes. Presentamos una técnica de optimización basada en el análisis del comportamiento del código y su interacción con la máquina donde se ejecuta.**

Como resultado de la aplicación de nuestro proceso de optimización en una implementación de un algoritmo de factorización, se ha logrado una reducción de ciclos de reloj de una magnitud de 4.5. Hemos comparado nuestro código optimizado con algunas herramientas de cálculo simbólico, que ha resultado ser entre 1.5 y 3 veces más rápido.

Palabras clave– **Factorización de enteros, Optimización de código, Arquitectura del computador**

I. Introducción

El poder de computación se hace cada vez más accesible y más rápido. Y aunque en sus primeros compases la solución al reto de la factorización de enteros se consideró mera cuestión de tiempo y de esperar el avance de las tecnologías [14], posteriormente se ha visto cómo la tarea de factorizar un entero, producto de dos primos largos, no resulta en absoluto trivial aunque dispongamos de mucha potencia de cálculo; y que los algoritmos actualmente existentes no logran dar solución a este problema: al menos entre los algoritmos conocidos por la comunidad científica.

Han aparecido los sistemas criptográficos que basan su seguridad en esta, hasta la fecha, inhabilidad para factorizar. De todos es conocido el criptosistema de Clave Pública RSA [17] nacido en 1978 de la mano de Rivest, Shamir y Adleman, del MIT. Los tiempos necesarios para la factorización de enteros puestos en relación con el tamaño de esos números nos indican el grado de seguridad de este criptosistema. El último reto de factorización conocido y publicado se logró en agosto de 1999 [2]: se trataba de un entero de 512 bits (155 dígitos decimales), producto de dos primos largos. Como señala Lenstra en [9] y [10] para el presente año 2003, se considera una longitud de clave para RSA completamente segura una cuyo módulo, producto de dos primos largos, tenga 1068 bits. En la tabla que presenta, Lenstra recoge tamaños de claves hasta el año 2040: para ese año garantiza como clave segura para RSA una de 3214 bits.

El estudio de las técnicas de factorización de números de gran tamaño, y todo posible avance para lograr reducir tiempos en esos procesos, resulta de enorme interés para la criptografía. No es necesario señalar el interés matemático de la cuestión, que no puede darse por resuelta. El Teorema Fundamental de la Aritmética [18] afirma que la factorización de un compuesto es única excepto en el factor 1 (que interviene tantas veces como queramos) y en el orden de los factores. No se conoce ningún algoritmo de factorización que realice la operación en un tiempo polinomial.

Como señalan Schneier y Whiting [20] lo más importante en una implementación de cualquier algoritmo criptográfico, es la seguridad. Pero es necesario realizar la implementación de forma que se obtengan al final herramientas eficientes. Y es que es posible escribir programas en C, aparentemente eficientes, pero que posteriormente, al compilar, no logran utilizar de forma óptima los recursos que ofrecen las nuevas arquitecturas de los microprocesadores actuales. Y si bien es verdad que los compiladores han evolucionado y mejorado mucho en los últimos años, también es cierto que si tenemos en cuenta esas arquitecturas, podemos llegar a códigos ejecutables mucho más eficientes.

Los nuevos microprocesadores disponen de una serie de contadores internos, cuya lectura no es accesible mediante código en lenguaje C, pero a los que sí podemos llegar mediante algunas herramientas disponibles en el mercado, tanto de libre distribución como propietarias (como por ejemplo la Vtune [21], implementada por Intel).

Nuestro objetivo, una vez definidas y creadas todas las funciones necesarias para trabajar con enteros largos (como requieren los procesos en criptografía), se ha centrado en la optimización de las implementaciones de una aplicación de factorización de enteros producto de dos primos largos, y reducir así los tiempos de ejecución. El proceso de mejora del código lo hemos podido realizar gracias al conocimiento de la arquitectura del ordenador y al estudio y análisis del comportamiento de nuestro código y su interacción con la máquina sobre la que hemos realizado todo nuestro trabajo.

Pedro Alcover es profesor del Departamento de Tecnología de la Información y las Comunicaciones de la Universidad Politécnica de Cartagena. e–mail: pedro.alcover@upct.es. José M. García es profesor del Departamento de Ingeniería y Tecnología de Computadores de la Universidad de Murcia. e–mail: jmgarcia@ditec.um.es

En el siguiente epígrafe recogemos una vista resumida y genérica de los algoritmos actuales de factorización. En el apartado 3 presentamos el protocolo de actuación que hemos definido para realizar la tarea de optimización de código. Mostramos después (apartado 4) algunos ejemplos de cómo hemos procedido para reducir tiempos en algunas de las funciones trabajadas y luego, en el apartado 5, recogemos resultados y valores de tiempos y otros parámetros que orientan la optimización y dan información de las mejoras reales introducidas. En el último epígrafe presentamos unas breves conclusiones de nuestro trabajo.

II. Fundamentos y trabajos relacionados

Existen tres algoritmos de factorización subexponenciales con un fundamento matemático semejante. Citados en orden histórico, y también en orden de eficiencia, estos algoritmos son el basado en la técnica de las fracciones continuas (CFRAC), introducido en 1975 por Morrison y Brillhart [11], el de la criba cuadrática (QS), introducida a principio de la década de los 80 por Pomerance [13], y el de la criba de campo numérico (NFS), introducida en los primeros años de la década de los 90 por Lenstra et al. [8]. Todos ellos se basan en una idea introducida por Fermat que pretende la búsqueda de una relación de la forma

$$x^2 \equiv y^2 (\mathrm{mod}\, N) \qquad (1)$$

siendo N el número que se desea factorizar. De la relación (1), se deduce que

$$x^2 - y^2 = k \cdot N ,$$

es decir, que

$$(x + y) \cdot (x - y) = k \cdot N , \qquad (2)$$

y buscando mediante el algoritmo de Euclides [1] el máximo común divisor de $(x + y)$ con N y de $(x - y)$ con N, tenemos un 50% de probabilidades de que el resultado sea un factor no trivial de N.

Los tres métodos señalados coinciden en dedicar el máximo de sus esfuerzos en lograr hallar una relación como la recogida en (1). Todos ellos procuran métodos de generación de relaciones o congruencias de la forma

$$P^2 \equiv C (\mathrm{mod}\, N) , \qquad (3)$$

que verifiquen que C es B-Suave, para un determinado valor B, es decir, que todos los divisores de C sean menores que un límite superior fijado B. Una vez halladas suficientes relaciones de esta forma (3) se buscan subconjuntos de esas relaciones, que llamaremos subconjuntos S, que verifiquen que el producto de todos los C de cada subconjunto resulte ser cuadrado perfecto. Para la búsqueda de los diferentes subconjuntos S se utiliza la técnica de eliminación gaussiana. Todo este proceso viene ampliamente documentado en [4] y [16].

El producto de todos los elementos de un subconjunto S nos llevará a una expresión de la forma

$$\prod_{Ci \in S} P_i^2 \equiv \prod_{Ci \in S} Ci (\mathrm{mod}\, N) \qquad (4)$$

donde al haber logrado que la parte derecha de la congruencia sea cuadrado perfecto, tendremos que la expresión (4) es análoga a la (1).

Ya hemos dicho que el reto de la factorización de enteros parece lejos de estar resuelto y va a requerir nuevos planteamientos en los algoritmos y nuevos campos de estudio en las matemáticas. En este sentido, la aparición del método de factorización de las curvas elípticas (ECM) (cfr. [6] y [7]) resultó un cambio de estilo, en el que se abandona el planteamiento de Fermat y se sale en busca de nuevas alternativas. De todas formas, hoy por hoy, este algoritmo no ha logrado resultados mejores que los alcanzados por NFS.

No es el objeto de este artículo presentar una descripción pormenorizada de estos algoritmos, ni de su implementación. A lo largo de estas líneas podremos hacer referencia a diferentes funciones implementadas: el único interés singular que pueden tener es el de ser cauce para mostrar diferentes formas de optimizar un determinado código según las pautas de actuación que presentamos más adelante. Hemos trabajado sobre una implementación del método CFRAC, pero todo lo que mostraremos en estas líneas resulta válido para una optimización de cualquiera de las tres implementaciones de los tres algoritmos presentados.

Respecto a las nuevas arquitecturas de los microprocesadores, el uso de la segmentación y la ejecución superescalar supone una mejora importante. Pero, como se afirma en [20], aunque las arquitecturas superescalares ofrecen la posibilidad de incrementar de forma sustancial la velocidad, también incrementan de forma dramática las penalizaciones por una mala implementación de los algoritmos. Y cuando el grado de ejecución superescalar aumenta, las reglas de optimización son cada vez más complejas.

Disponer de una guía general de optimización y realizar una exploración de la implementación del software, puede ayudar en el futuro a mejorar la velocidad de ejecución de los algoritmos criptográficos. Como afirma Clapp [3], el incremento de la eficiencia de los procesadores, su segmentación y el aumento del paralelismo, puede orientar hacia el diseño de nuevos algoritmos de cifrado o de criptoanálisis.

Otro dos factores de importancia decisiva en la eficiencia en la implementación de los algoritmos son la cantidad de accesos a memoria requeridos y el número de instrucciones de salto que deberá ejecutar y predecir el microprocesador. Una implementación será más eficiente cuando, al margen de otras consideraciones, se logre reducir al máximo el número de referencias a memoria de datos, y, por tanto, la cantidad de fallos en el acceso a la memoria caché en sus diferentes niveles. Y de la misma manera, también ganaremos velocidad cuanto menores sean las instrucciones de salto ejecutadas, y especialmente, las instrucciones de salto equivocadamente predichas.

III. Metodología de trabajo

El procedimiento que hemos seguido ha sido el siguiente:

En primera lugar, ejecutar repetidas veces el programa implementado para la factorización de un compuesto de dos primos grandes. Todo el proceso de optimización ha sido realizado con un número prefijado, de 100 bits, producto de dos primos de 50 bits cada uno:

```
8 4218BCE3 04AFD040 00D35DB9 =
2C7E3 25E2BDE1 * 2F90B F8A9AAD9
```

El objetivo de esta reiterada ejecución es determinar cuáles son las funciones que consumen mayor cantidad

de tiempo. Como sugiere la Ley de Amdhal, convendrá centrar los procesos de optimización en esas funciones.

Para cada función a optimizar hemos seguido un protocolo de actuación. Este protocolo puede sintetizarse en los siguientes pasos:

1. Búsqueda de un algoritmo, distinto al empleado, que reduzca el tiempo de ejecución. Es decir, búsqueda de algoritmos mejores a los utilizados en una primera implementación.
2. Procurar eliminar instrucciones. Una vez tenemos el algoritmo seleccionado, lograr su ejecución con menos instrucciones supondrá siempre una mejora en la eficiencia de su ejecución. Se trata de reducir instrucciones sin variar el planteamiento del algoritmo seleccionado. De lo contrario estaríamos en un intento de mejora como el señalado en n.1; y hay que tener en cuenta que no toda reducción de instrucciones es conveniente: a veces puede lograrse al precio de aumentarlos ciclos por instrucción (índice CPI), lo que puede llevar finalmente a un incremento del tiempo de ejecución. Nos referimos específicamente a la reducción de instrucciones por simple eliminación. Aunque este paso puede parecer trivial, la realidad es que a medida que el código a optimizar sufre diferentes modificaciones, orientadas a su mejora en la eficiencia, aparecen de hecho instrucciones residuales prescindibles. También se logra reducir instrucciones mediante técnicas de reuso, creando nuevas variables que almacenen un valor repetidamente calculado.
3. Procurar reducir las instrucciones de salto: todo lo que reduzca estas instrucciones favorecerá el índice CPI. Para lograr este objetivo, una medida es la de desenrollar los bucles, procurando un número suficiente de instrucciones dentro del bucle. Con esta tarea se pretende lograr que la ventana de instrucciones se mantenga siempre llena de instrucciones a ejecutar. Nos hemos propuesto lograr bucles de, al menos, 15 instrucciones.
4. Evitar dependencias de datos: la solución se centra en separar las instrucciones que sufren este tipo de dependencia de datos: insertar código entre dos sentencias con variables con dependencias RAW; y realizar segmentación software.
5. Optimizar los accesos a memoria: el número de accesos a memoria, y los fallos de acceso a memoria. Para optimizar estos accesos tenemos una colección de técnicas: prefetching, la mezcla de arrays, el intercambio de bucles, y la fusión de bucles.

Cada nueva mejora introducida se estudia de forma aislada: cada mejora individual dirigida a un solo objetivo del protocolo de actuación genera una nueva versión de la función a optimizar, diferente de la versión previa únicamente en esa mejora individual. Si la reducción del número de instrucciones o de los ciclos de reloj invertidos resulta ser mayor que 3% (ya veremos cómo lograr conocer estos valores), entonces la mejora queda como definitiva en el proceso, y se continúa el protocolo en el siguiente paso, buscando una nueva versión de la función a partir de la actual ya modificada y validada.

El objetivo en todo momento es doble: reducir el número de instrucciones, y reducir el valor del índice CPI.

IV. Propuestas desarrolladas

En este apartado vamos a aplicar el protocolo presentado, mostrando algunos ejemplos de cada uno de los cinco pasos descritos.

Es de todos conocido la baja eficiencia del método de ordenación de la burbuja frente a otros algoritmos, como por ejemplo el método desarrollado por C. Hoare [5] e implementado mediante la función `qsort()` de la biblioteca `stdlib.h` [19]. Como ejemplo del primer paso del protocolo (búsqueda de otro algoritmo más eficiente), hemos analizado el comportamiento de ambos algoritmos y tenemos que la función resulta ser 50 veces más rápida y ejecuta 70 veces menos instrucciones que el algoritmo implementado por el método de la burbuja.

Un ejemplo de cómo se pueden reducir instrucciones lo tenemos si estudiamos el comportamiento de los algoritmos que emplean el tipo de dato `unsigned long long int`. Hemos comprobado que se logra un incremento notable en la eficiencia si se evita el uso de esas variables definidas en el compilador `gcc` de Linux como enteras de 64 bits. Buena muestra de ello la tenemos en una función implementada, encargada de ordenar un vector de valores enteros consecutivos (desde 1 hasta 1300 en nuestro caso), que se empleará como vector de índices, en función de los valores de otros 1300 enteros almacenados en otro vector de tipo `unsigned long long int`. Si, haciendo un replanteamiento del proceso que nos permita prescindir de variables de ese tamaño, logramos cambiar el tipo de dato del vector de valores sobre los que se realiza la ordenación del vector de índices por un tipo estándar de C: entero sin signo de 32 bits (`unsigned long int`), y estudiamos de forma comparada el comportamiento de las dos implementaciones, observamos que el uso de variables enteras de 64 bits (no estándares en ANSI C) con operadores relacionales duplica el número de instrucciones a ejecutar. Hemos observado que el uso de este tipo de dato de 64 bits duplica también el número de las instrucciones de salto y aumenta en un factor 2.3 las referencias a memoria de datos. Hemos llegado a resultados similares cuando los operadores empleados con estas variables han sido los aritméticos, como por ejemplo (son los que hemos probado) la suma o el cociente. El uso de la variable entera de 32 bits opera a una velocidad 1.75 veces superior a la de 64 bits.

Un tercer ejemplo de optimización, que ha ofrecido la reducción de las instrucciones de salto lo tenemos en la sustitución de una sentencia `while` por una secuencia limitada de sentencias `if-else`. Tenemos definida una función cuyo cometido es determinar, dentro de una variable `unsigned long int`, en qué posición se encuentra el bit igual a 1 más significativo. En la primera versión teníamos un código semejante al que sigue:

```
if(!n) bits = 0;
else{
while(!(n & Test))
{   Test >>= 1;
    bits--; }}
```

donde `Test` es una variable `unsigned long int` inicializada al valor `0x80000000`, `bits` está inicializada al valor `32`, y `n` es la variable de la que deseamos conocer la posición del bit más significativo.

La modificación que hemos realizado sobre la función alarga en mucho su código. El cambio consiste en sustituir una estructura `while` en 32 `if` y 32 `else` concatenados y anidados hasta un nivel de 5. En la nueva versión se realiza la operación `AND` a nivel de bit entre la variable `n` y `0xFFFF0000`; Si resulta un valor verdadero (distinto de cero), repetimos la operación con `AND` con `0xFF000000`, y en caso contrario lo hacemos con `0x0000FF00`; y así sucesivamente. El código queda de la forma:

```
if(!n) bits=0;
else{
if(0xFFFF0000&n){
if(0xFF000000&n){
if(0xF0000000&n){
if(0xC0000000&n)bits=(0x80000000&n)?32:31;
else bits=(0x20000000&n)?30:29;}
else{
if(0x0C000000&n)bits=(0x08000000&n)?28:27;
else bits=(0x02000000&n)?26:25;}}
else{
if(0x00F00000&n){
if(0x00C00000&n)bits=(0x00800000&n)?24:23;
else bits=(0x00200000&n)?22:21;}
else{
if(0x000C0000&n)bits=(0x00080000&n)?20:19;
else bits=(0x00020000&n)?18:17;}}}
else{
if(0x0000FF00&n){
if(0x0000F000&n){
if(0x0000C000&n)bits=(0x00008000&n)?16:15;
else bits=(0x00002000&n)?14:13;}
else{
if(0x00000C00&n)bits=(0x00000800&n)?12:11;
else bits=(0x00000200&n)?10:9;}}
else{
if(0x000000F0&n){
if(0x000000C0&n)bits=(0x00000080&n)?8:7;
else bits=(0x00000020&n)?6:5;}
else{
if(0x0000000C&n)bits=(0x00000008&n)?4:3;
else bits=(0x00000002&n)?2:1;}}}}
```

Con el nuevo procedimiento toda ejecución de este algoritmo exige invariablemente 5 evaluaciones de sentencias condicionales; en la primera versión el número de evaluaciones de salto dependía de la posición del bit 1 más significativo en la variable `n` que, por término medio, debería ser de 16. Con este cambio, a pesar de aumentar notablemente el número de líneas del programa, queda sustancialmente reducido el número de instrucciones ejecutadas. Las instrucciones de salto tomadas y ejecutadas se han reducido en una proporción de 3.3 veces. Más significativa es la reducción de referencias a memoria de datos, que resulta ser del orden de 7.8. Como resultado de todo esto, los ciclos de reloj se dividen a casi un tercio y la reducción de las instrucciones totales ejecutadas es del orden de 3.1.

También hemos obtenido algunas mejoras interesantes al intentar eliminar o reducir las dependencias de datos. Podemos mostrar un ejemplo sencillo con el siguiente código:

```
for(;i<l;){
C=suma>>Byte4;
suma=C+(UINT8)*(a->N+i)+(UINT8)*(b->N+i);
*(c->N+i++)=(UINT4)suma;
C=suma>>Byte4;
suma=C+(UINT8)*(a->N+i)+(UINT8)*(b->N+i);
*(c->N+i++)=(UINT4)suma;}
```

No es el objeto ahora explicar la utilidad de este proceso que utilizamos en la definición de una de nuestras funciones. `Byte4` es una macro que vale `32`. `suma` es una variable de tipo `unsigned long long int` (que hemos redefinido con el nombre `UINT8`). Los campos de las variables `a` y `b` son del tipo `unsigned long int`. Si a este código le hacemos una simple modificación:

```
for(;i<l;){
C=suma1>>Byte4;
suma1=C+(UINT8)*(a->N+i)+(UINT8)*(b->N+i);
*(c->N+i++)=(UINT4)suma1;
C=suma2>>Byte4;
suma2=C+(UINT8)*(a->N+i)+(UINT8)*(b->N+i);
*(c->N+i++)=(UINT4)suma2;}
```

(introducimos las variables `suma1` y `suma2`, del mismo tipo que antes era `suma`) se obtiene una reducción de ciclos de reloj del 2.45 %, mientras que todos los demás parámetros que definen el comportamiento del algoritmo se mantienen prácticamente igual.

Por último, recogemos dos ejemplos de mejora por optimización de los accesos a memoria. El primero viene de la mano de la optimización de una función que realiza el cociente entre dos enteros largos. En [12] puede verse una secuencia de sucesivas mejoras de un algoritmo similar. La idea que se recoge en ese libro, y que hemos utilizado para modificar el código de nuestra función, consiste en realizar todas las operaciones que conducen a la división en un solo vector: mezclar la información de diferentes arrays en uno solo, perdiendo ciertamente claridad en la presentación del código, pero logrando reducir los accesos a memoria por un factor de 2.63. También quedan reducidos a la mitad, en nuestro caso, los fallos de acceso a la memoria caché L2. Gracias a esas mejoras las instrucciones a ejecutar se reducen en un orden de 2.56, y la función se ejecuta 2.40 veces más rápido.

El segundo ejemplo que presentamos, consiste en la modificación de las definiciones de una serie de matrices, de forma que podamos reducir el número de fallos en los accesos a las memorias cachés. La función implicada en esta modificación realiza el proceso de eliminación gaussiana (cfr [1]) en dos matrices que abarcan entre ambas algo más de los dos Mbytes de información. El ejemplo muestra la gran importancia que tiene escoger adecuadamente el orden en que se recorren las matrices. Al alterar ese orden, y al redefinir las matrices, hemos llegado a una implementación que, aunque supone mayor número de referencias a la memoria de datos, reduce en un factor de 30 el número de fallos a la memoria Caché L2 y a 2.5 el de los fallos a L1. Y gracias a ello, aunque el número de instrucciones que se han ejecutado en le proceso se ha mantenido casi constante, el tiempo de ejecución se ha reducido en un factor de casi 20.

V. Evaluación y análisis de resultados

Todo nuestro trabajo de evaluación ha sido realizado sobre un Pentium III a 1 GHz; hemos trabajado sobre el

sistema operativo Linux Red Hat, versión 6.2, con la versión de núcleo 2.2. El compilador utilizado ha sido el gcc, y todas las compilaciones las hemos hecho en su opción –O3. En nuestra máquina tenemos dos niveles de memoria caché: L1 y L2. El tamaño de la memoria L1 es de 16 Kbytes. El de L2 de 256 Kbytes. El tamaño del bloque es de 32 bytes.

La herramienta que hemos utilizado para acceder a los valores de los contadores o registros internos del microprocesador ha sido Rabbit (cfr. [15]). Esta herramienta, de libre distribución, permite acceder y mostrar los contadores de eventos de los procesadores Intel y AMD ocurridos en la ejecución de programas implementados en C bajo el sistema operativo Linux. Los eventos o registros que hemos estudiado, de entre todos los que ofrece la herramienta Rabbit, han sido 9 (distribuidos en tres bloques), y que quedan recogidos en la Tabla I.

TABLA I

LISTADO DE REGISTROS TOMADOS CON LA HERRAMIENTA RABBIT.

0x24	Fallos en L2
0x43	Referencias a memoria de datos
0x45	Fallos en L1
0x79	Ciclos de reloj
0xC0	Instrucciones ejecutadas
0xC4	Instruc. de salto ejecutadas
0xC5	Instruc. de salto equivocadas predichas
0xC9	Saltos tomados ejecutados
0xCA	Saltos tomados mal predichos ejecutados

Trabajando tal y como hemos descrito en n. 3, la primera tarea ha de ser la ejecución reiterada del programa de factorización, para obtener, gracias a la lectura del contador 0x79, el tiempo de ejecución de cada una de las 40 funciones que intervienen en todo el proceso. En la Tabla II recogemos la relación de las 20 funciones más gravosas en tiempo de ejecución, ordenadas según esos valores de tiempo obtenidos. El tiempo de las otras 20 funciones podemos considerarlo despreciable, y supone menos del 1 % del total. Por la Ley de Amhdal, bien podemos prescindir de ellas para la tarea de las optimizaciones.

En esos tiempos hay que tener en cuenta que no todas las funciones son invocadas el mismo número de veces. También nos interesa saber las llamadas que recibe cada una de esas 20 funciones. En la Tabla III mostramos las mismas 20 funciones de la Tabla II, ordenadas ahora por el número de veces que cada una es invocada en la aplicación. Esa información también es de gran utilidad porque una pequeña optimización en una función invocada varios millones de veces es siempre más rentable que esa misma optimización en una función apenas invocada.

En la Tabla IV recogemos de nuevo la lista de las 20 funciones, esta vez ordenadas según el valor obtenido gracias al contador 0xC0 (instrucciones ejecutadas). Con esos valores, y los presentados en la Tabla II, podemos conocer el valor del índice IPC de cada función, y detectar el grado de paralelismo de cada función.

TABLA II

FUNCIONES MÁS COSTOSAS EN TIEMPO DE EJECUCIÓN (EN CICLOS DE RELOJ).

1	Suave_S	2.991.356.843
2	Modulo	2.302.815.183
3	RelacionesBhascara	1.325.755.488
4	EliminacionGaussiana	983.505.324
5	OrdenRelaciones	867.243.499
6	longitud	742.386.827
7	Cociente	617.352.622
8	MODULO	571.220.615
9	DESPL_izda	484.609.406
10	CrearNumero	336.206.555
11	PROD_bit	326.226.736
12	COCIENTE	304.322.614
13	PonerACero	271.170.391
14	SUMA	148.622.492
15	RESTA	132.761.172
16	CopiarNumero	95.507.792
17	BuscarCuadrados	77.532.679
18	Euclides	31.991.818
19	PonerACeroRel	24.662.592
20	orden	19.006.563

TABLA III

FUNCIONES MÁS COSTOSAS, ORDENADAS SEGÚN EL NÚMERO DE LLAMADAS QUE RECIBEN.

Modulo	14.641.005
DESPL_izda	12.921.596
PonerACero	3.000.411
CopiarNumero	1.659.896
longitud	1.345.085
orden	1.083.761
CrearNumero	849.711
SUMA	600.412
RESTA	598.654
PROD_bit	249.181
Cociente	145.303
MODULO	88.544
COCIENTE	83.123
PonerACeroRel	82.757
RelacionesBhascara	82.757
Suave_S	41.620
Euclides	950
BuscarCuadrados	9
OrdenRelaciones	4
EliminacionGaussiana	1

El protocolo de optimización lo hemos aplicado por riguroso orden de peso: tal y como nos indica la Tabla II. En el apartado 4 de este artículo hemos mostrado unos pocos ejemplos del proceso. En total se han originado más de 200 versiones diferentes de las distintas funciones optimizadas. Los valores comparados (antes y después de la optimización) de los valores de los índices estudiados con la herramienta Rabbit para la aplicación de factorización completa, están recogidos en la Tabla V.

Los valores que se recogen en la esa última tabla muestran una reducción importante en todos los valores de los contadores del microprocesador. De su análisis, lo primero que podemos señalar es que se ha logrado aumentar la velocidad de ejecución del proceso en un factor de 4.5. El número de instrucciones ejecutadas también ha bajado muy sensiblemente: casi en un orden de 6.

TABLA IV

FUNCIONES MÁS COSTOSAS, ORDENADAS SEGÚN EL NÚMERO DE INSTRUCCIONES QUE EJECUTAN.

Suave_S	2.961.746.229
Modulo	2.057.140.312
RelacionesBhascara	1.713.630.879
OrdenRelaciones	1.212.667.593
longitud	944.378.153
MODULO	853.348.861
Cociente	820.917.600
DESPL_izda	780.978.112
COCIENTE	418.521.009
PROD_bit	350.243.126
CrearNumero	185.742.033
SUMA	140.773.445
RESTA	120.465.161
BuscarCuadrados	100.080.633
PonerACero	94.152.291
CopiarNumero	89.621.492
EliminacionGaussiana	55.235.941
Euclides	37.179.341
PonerACeroRel	14.897.570
orden	13.161.663

TABLA V

REGISTROS ANALIZADOS CON RABBIT, EN LA APLICACIÓN DE FACTORIZACIÓN, ANTES Y DESPUÉS DE LA OPTIMIZACIÓN.

	Versión V0	Versión V1	V0/V1
0x24	5.254.965	25.781	203,83
0x43	4.441.428.694	808.793.008	5,49
0x45	80.424.542	93.366	861,39
0x79	7.672.819.100	1.702.847.174	4,51
0xC0	7.038.923.209	1.181.864.423	5,96
0xC4	1.029.056.327	186.338.505	5,52
0xC5	31.804.944	15.702.519	2,03
0xC9	594.373.205	131.769.458	4,51
0xCA	15.007.514	8.108.609	1,85

El descenso de las referencias a la memoria de datos también es importante: del orden de 5.49. Pero los valores más llamativamente significativos respecto a los accesos a memoria los encontramos en el enorme descenso obtenido en los fallos a L1 (del orden de 861) y a L2 (del orden de casi 204). Este descenso ha sido alcanzado gracias a las redefiniciones de las matrices de relaciones que se generan en los procesos de factorización, y a las modificaciones en el modo de recorrer los valores de esas matrices, procurando aprovechar la localidad temporal y, en nuestro caso especialmente, la localidad espacial.

La tarea de optimización no se termina sino que se deja. Podríamos continuar, especialmente procurando la reducción de instrucciones de salto, y especialmente aquellas que resultan mal predichas y ejecutadas, donde la mejora solo ha alcanzado a ser dividir por dos.

VI. CONCLUSIONES

Hemos presentado un protocolo de actuación para optimizar implementaciones de algoritmos. El plan de trabajo puede resultar válido para cualquier tarea de optimización. Nosotros lo hemos definido y empleado para aplicaciones de criptoanálisis, donde los algoritmos que se manejan habitúan a ser costosos en tiempos de computación y, por tanto, donde es conveniente siempre procurar métodos y procedimientos de trabajo que nos permitan aumentar la eficiencia de los códigos.

Hemos mostrado diferentes resultados de optimización, según hayamos centrado la atención en la reducción de instrucciones o específicamente en la reducción de las instrucciones de salto; en la reducción de accesos a memoria; en el cambio de algoritmo por otro más eficiente; o en evitar las dependencias de datos. El resultado final ha sido la reducción de los ciclos de reloj en un orden de magnitud de 4.5.

Hemos comparado nuestra implementación final con otras existentes en el mercado y de las que hemos podido disponer en la misma máquina donde hemos realizado todas las mediciones. Nos hemos centrado específicamente en aquellas que factorizan los enteros largos con el algoritmo CFRAC, y no implementan otros como QS, NFS o el algoritmo de factorización por curvas elípticas (ECM). Nuestra implementación, una vez optimizada, resulta ser entre 1.5 y 3 veces más veloz que las estudiadas: MAPLE, Matemática y Derive.

REFERENCIAS

[1] D. Bressoud. And S. Wagon. Computational Number Theory. Key College Publishing 2000.

[2] S. Cavallar, B. Dodson, A. K. Lenstra, K. Lioen, P. L. Montgomery, B. Murphy, H. Te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Miffert, C and C. Putnan, P. Zimmermann. Factorization of a 512-Bit RSA Modulus. LNCS 1807. Eurocrypt'2000.

[3] C. Clapp. Optimizing a Fast Stream Copher for VLIW, SIMD, and Superescalar Processors. LNCS 1267. Fast Software Encryption'97.

[4] H. Cohen. A Course in Computational Algebraic Number Theory. Springer Verlag 1993.

[5] http://www.utm.edu/research/primes/index.html

[6] N. Koblitz. A Course in Number Theory and Cryptography. Springer Verlag 1994.

[7] N. Koblitz. Algebraic Aspects of Cryptography. Springer Verlag 1998.

[8] A. K. Lenstra and H. W. Lenstra jr. Editors. The development of the number field sieve. Lecture Notes in Mathematics, 1554. Springer Verlag 1993.

[9] A. K. Lenstra y E. R. Verhenl. Selecting Cryptographic Key Sizes in Comercial Applications. PriceWaterhouseCoopers, 1999

[10] A. K. Lenstra. Selecting Cryptographic Key Sizes. Journal of Cryptology 14: 255–293 (2001).

[11] M. A. Morrison and J. Brillhart. A Method of Factoring and the Factorization of F7. Math. of Comp, Vol 29. Nº 129, pags 183 - 205. Jan 1975.

[12] D. A. Patterson y J. L. Hennessy. Estructura y diseño de computadores. Reverté, 2000.

[13] C. Pomerance. Analysis and comparison of some integer factoring algorithms. Computational Methods in Number Theory. (H. Lenstra and R. Tijdeman eds.), pag 89–141. 1982.

[14] C. Pomerance. A Tale of Two Sieves. Noticies of the AMS. December 1996.

[15] http://www.scl.ameslab.gov/Projects/Rabbit/

[16] H. Riesel. Prime Nubers and Computer Methods for Factorization. Birkhäuser, 1987.

[17] R. L. Rivest, A. Shamir and L. Adleman. A Method for Obtaining Digital Signatures and Public – Key Cryptosystems. Comunication of the ACM. V 21, n. 2, pags 120 – 130. February 1978.

[18] K. H. Rosen. Elementary Number Theory and its applications. Addison – Wesley 1993.

[19] H. Schildt. C: Guía de Autoenseñanza. Osborne McGraw – Hill 1994.

[20] B. Schneier and D. Whiting. Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor. LNCS 1267. Fast Software Encryption'97.

[21] http://www.intel.com/software/products/vtune/