Benchmarking y optimización de algoritmos criptográficos

Miguel Montes
Facultad de Ingeniería
Instituto Universitario Aeronáutico
mmontes@iua.edu.ar

Daniel Penazzi
Famaf
Universidad Nacional de Córdoba
penazzi@famaf.unc.edu.ar

Resumen

El presente trabajo presenta los resultados de la optimización de algoritmos criptográficos en procesadores que disponen de las nuevas instrucciones AES, desarrolladas por Intel. En particular se trabajó sobre AES-CPFB y Silver, dos algoritmos de cifrado autenticado presentados en la competencia CAESAR.

1. Introducción

AES (Advanced Encryption Standard) es un algoritmo de cifrado de bloques estandarizado por NIST (National Institute of Standards and Technology) en 2001 [1] . Desarrollado por dos criptógrafos belgas, Vincent Rijmen y Joan Daemen, utiliza bloques de 128 bits, y admite claves de 128, 192 y 256 bits [2, 3].

Se trata del algoritmo de cifrado simétrico más utilizado, y es probablemente el algoritmo de cifrado mejor analizado de la historia, por lo que es frecuente su uso como componente de otros algoritmos y protocolos. Tal es el caso de Silver [4] y AES-CPFB [5], que constituyen el objeto del presente trabajo. Ambos son algoritmos de cifrado autenticado, presentados en el marco de la competencia CAESAR: Competition for Authenticated Encryption: Security,Applicability, and Robustness, y usan AES como componente esencial. AES-CPFB es un modo de operación, que utiliza AES en modo "caja negra", mientras que Silver utiliza una versión modificada de AES.

Si bien la seguridad es la característica primordial de un algoritmo criptográfico, también es extremadamente importante su velocidad. Los grandes anchos de banda en uso en la actualidad requieren de cifradores que sean capaces de procesar varios gigabits de datos por segundo. Eso ha motivado, por un lado, el desarrollo de algoritmos de cifrado de flujo cada vez más rápidos, y por el otro, la aceleración por hardware de algoritmos estándar como AES o SHA.

En este trabajo nos centraremos en la utilización de una de estas tecnologías de aceleración por hardware: las nuevas instrucciones AES desarrolladas por Intel, y disponibles en las últimas generaciones de procesadores x86.

2. Benchmarking

La medición de velocidad de algoritmos criptográficos presenta tres problemas básicos: qué medir, en qué unidades hacerlo, y cómo hacerlo. Se busca encontrar una métrica que permita hacer comparaciones significativas entre algoritmos. Al comparar algoritmos se suele recurrir a evaluar su comportamiento asintótico, pero ese recurso no es útil en este caso: todos los algoritmos de cifrado de bloques tienen un costo lineal con respecto al tamaño de la entrada expresado en bloques.

Sin embargo, suelen tener un elevado costo de inicialización, habitualmente asociado con la expansión de la clave, el cual se vuelve relevante para pequeños tamaños de entrada. Si bien este costo de inicialización se amortiza fácilmente en el cifrado de un archivo de varios megabytes, es significativo en el cifrado de paquetes pequeños a ser enviados por la red. Este problema es especialmente severo en el caso de los algoritmos de cifrado autenticado, en los cuales se suele utilizar un *nonce* asociado con cada mensaje, lo cual implica costos de inicialización adicionales para cada *nonce*. Por ello se suele medir el comportamiento de estos algoritmos para distintos tamaños de entrada, y a veces se usa un promedio ponderado de ciertos tamaños típicos de paquete. En nuestro caso hemos utilizado un promedio ponderado de paquetes de 44, 552, 576 y 1500 bytes, tal como se muestra en el cuadro 1. Esta proporción se corresponde con el *Internet Performance Index* (IPI) propuesto por David McGrew y John Viega [**?**], y utilizada por Rogaway en la implementación de referencia de OCB [6]. El IPI es el número esperado de bits procesados por ciclo de reloj cuando la distribución de paquetes

se corresponde con la del cuadro 1, y se considera un indicador útil del rendimiento de un módulo criptográfico que protege tráfico IP.

Además, hemos medido el rendimiento en paquetes de 65536 bytes, como muestra del comportamiento en mensajes largos.

Cuadro 1. Distribución típica de tamaños de paquete

Bytes	Peso
44	0.05
552	0.15
576	0.20
1500	0.60

Una vez definido qué vamos a medir, es necesario determinar en qué unidades mediremos. Parecería razonable medir en bytes por unidad de tiempo (por ejemplo megabytes por segundo), pero no es una solución adecuada, ya que es una métrica muy dependiente de la velocidad del procesador utilizado, y no permite comparar algoritmos entre sí. Es necesario encontrar una forma de medir que sea relativamente independiente de la velocidad del procesador, y que permita extrapolar el rendimiento de un algoritmo en distintos procesadores (al menos dentro de una misma arquitectura). Una solución es utilizar bits por ciclo, tal como en el IPI mencionado más arriba. Otra solución, muy difundida en el ambiente criptográfico y que es la que hemos adoptado, consiste en medir ciclos por byte (cpb). Es decir, cuantos ciclos de procesador se consumen para procesar un byte de entrada.

Nuestro promedio ponderado no es, pues, el IPI, ya que éste mide bits por ciclo y nosotros medimos ciclos por byte. Pero como utilizamos la distribución de tamaños de paquetes del IPI, y Silver y AES-CPFB operan sobre bytes completos, el IPI puede calcularse fácilmente a partir de nuestro promedio ponderado.

El siguiente problema es cómo medir esta velocidad. No existe un mecanismo uniforme de medición de los ciclos de procesador utilizados. Algunos mecanismos de benchmarking, como por ejemplo el de la biblioteca CryptoCPP, miden bytes por segundo y convierten a ciclos por byte conociendo la velocidad del procesador. En otros casos, se utilizan instrucciones del procesador que permiten obtener la cantidad de ciclos en forma directa. En nuestro caso, hemos utilizado este último sistema.

Como el trabajo de optimización se realizó en equipos con arquitectura x86, utilizamos la instrucción `rdtsc` (Read Timestamp Counter)[7]. Esta instrucción puede ser utilizada desde un programa en C utilizando *intrinsics*, en el caso de los compiladores de Intel y de Microsoft, y usando inline assembler en el caso de GCC.

El uso de `rdtsc` presenta varias sutilezas.

- Distintas familias de procesadores incrementan el contador de forma diferente. En procesadores como el Pentium M, P6, o los primeros Pentium 4, el contador se incrementa con cada ciclo del reloj del procesador. En procesadores posteriores, el contador se incrementa a una tasa constante, que depende de la máxima frecuencia nominal del procesador configurada en el momento del arranque. Esto significa que el contador sólo equivale a ciclos cuando el procesador funciona a esa frecuencia.
- Los procesadores actuales no trabajan a una frecuencia fija. Los mecanismos de ahorro de energía hacen que el procesador pueda trabajar en una gama de velocidades que pueden ser controladas por el sistema operativo. Por ejemplo, un procesador Core i7-4771 tiene una frecuencia mínima de 800 MHz y una frecuencia máxima de 3.5 GHz. El sistema operativo puede optar por correr siempre a 800 MHz (modo ahorro de energía), siempre a 3.5 GHz (modo rendimiento), o trabajar a demanda (800 MHz cuando no hay carga, y aumentar a 3.5 GHz cuando la hay). Esto altera las mediciones, por lo que es necesario forzar a que el equipo trabaje a su máxima frecuencia nominal (en este caso, 3.5 GHz).
- Los últimos procesadores Intel cuentan con una tecnología denominada Turbo Boost, que incrementa de forma automática la velocidad de procesamiento de los núcleos por encima de la frecuencia operativa nominal si no se han alcanzado los límites especificados de energía, corriente y temperatura. Esto dificulta aún más la medición, ya que depende de factores externos como la temperatura. En el caso del procesador Core i7-4771 mencionado anteriormente, la frecuencia máxima con Turbo Boost es 3.9 GHz, lo cual representa un incremento de velocidad de un 11.43 %. En procesadores de bajo consumo la diferencia es aún mayor. Por ejemplo, en un procesador Core i5-4200U, la máxima frecuencia operativa nominal es 1.6 GHz, mientras que la frecuencia con Turbo Boost es 2.6 GHz, es decir, un incremento del 62.5 %. Esto implica que para medir adecuadamente es necesario deshabilitar esta tecnología. En una máquina con Linux esto puede hacerse en tiempo de ejecución, modificando el archivo `/sys/devices/system/cpu/cpufreq/boost`. En

una máquina con Windows esto no es posible, por lo que es necesario inhabilitarla previo al arranque, accediendo al setup de la máquina.

- La instrucción `rdtsc` no se serializa u ordena con otras instrucciones. No necesariamente espera a que hayan finalizado otras instrucciones para leer el contador.
- No existe garantía de que todos los procesadores de un sistema multiprocesador o multicore tengan sus contadores sincronizados, por lo que es conveniente ligar cada proceso a un procesador específico. En Linux esto puede realizarse en tiempo de ejecución mediante el comando `taskset`.
- Existe sobrecarga asociada con el proceso de medición, por lo que el algoritmo de medición debe encargarse de calcular y sustraer esta sobrecarga.

El procedimiento de *benchmark* implica por lo tanto:

1. Deshabilitar Turbo Boost
2. Configurar el sistema operativo para trabajar en modo "rendimiento", es decir, que el procesador trabaje siempre a su máxima frecuencia operativa nominal.
3. Ejecutar el programa de medición ligándolo a un procesador usando los mecanismos de afinidad de procesador disponibles en cada sistema operativo.

El pseudocódigo de la figura 1 muestra la estructura del programa de medición.

```
BENCHMARK(M,N,P)
 1  for i = 1 to M
 2      c1 = RDTSC()
 3      for j = 0 to N
 4          P()
 5      c1 = RDTSC() − c1
 6      c2 = RDTSC()
 7      P()
 8      c2 = RDTSC() − c2
 9      A[i] = (c1 − c2)/N
10  return MEDIAN(A)
```

Figura 1. El procedimiento BENCHMARK

M y N son dos constantes utilizadas para parametrizar la medición. P es el procedimiento cuya velocidad queremos medir. Como puede verse en el lazo interno, ejecutamos $N+1$ veces el procedimiento P y medimos el tiempo utilizado. Luego medimos una vez más, y sustraemos ambas medidas. El resultado es el tiempo insumido en ejecutar N veces el procedimiento P, y con la sobrecarga asociada con la medición ya sustraída (asumimos que la sobrecarga de ejecutar una vez es la misma que al ejecutar $N+1$ veces). Repetimos el proceso M veces, almacenando los resultados parciales en un vector A de tamaño M (nótese que en este pseudocódigo, el primer elemento del vector es el 1). Finalmente, devolvemos la mediana de los valores medidos.

3. Las instrucciones AES-NI

Las nuevas instrucciones AES (cuadro 2) fueron introducidas por Intel en el año 2008 y constituyen una extensión del conjunto de instrucciones x86 que tienen el propósito de acelerar las operaciones de cifrado y descifrado con AES.

Cuadro 2. Instrucciones AES-NI

`AESENC`	Ejecuta una ronda de cifrado AES
`AESDEC`	Ejecuta una ronda de descifrado AES
`AESENCLAST`	Ejecuta la última ronda de cifrado AES
`AESDECLAST`	Ejecuta la última ronda de descifrado AES
`AESKEYGENASSIST`	Asiste en la generación de claves de ronda
`AESIMC`	Asiste en la operación Inverse Mix Columns

Estas instrucciones son accesibles mediante lenguaje assembler, o pueden ser utilizadas en C/C++ mediante extensiones provistas por el compilador (*intrinsics*). Los principales compiladores disponibles en la plataforma x86 (GCC, Intel y Microsoft) cuentan con estas extensiones. Por ejemplo, la instrucción `AESENC` puede ser utilizada desde C mediante la siguiente función *intrinsic*:

```
__m128i _mm_aesenc_si128(__m128i a,
                         __m128i RoundKey)
```

cuyo prototipo está declarado en el archivo `wmmintrin.h`. Los argumentos son registros de 128 bits, de tipo `__m128i`. La función ejecuta una ronda de cifrado AES en los datos en `a` usando la clave de ronda en `RoundKey`. Los argumentos deben estar alineados a 128 bits (es decir, en una dirección de memoria múltiplo de 16), por lo que para transferir los datos desde y hacia posiciones no alineadas deben usarse operaciones tales como `_mm_loadu_si128` y `_mm_storeu_si128`. La figura 2 muestra el código en C de una implementación simple de la función de cifrado de un bloque.

Ahora bien, si bien el código de la figura 2 es correcto, no aprovecha plenamente las capacidades de las nuevas instrucciones. Como puede verse en el cuadro 3 la instrucción `AESENC` tiene una latencia de 6 y un throughput de 2 en procesadores Westmere, mientras que

```
void AES_128_encrypt(
  const unsigned char* in,
  unsigned char* out,
  const unsigned char* subkeys){

  int i;
  __m128i subkey = _mm_loadu_si128(
                    (__m128i*) subkeys);
  __m128i output = _mm_xor_si128(
        _mm_loadu_si128((__m128i*)in),subkey);

  for (i = 1; i < ROUNDS; i++){
    subkey = _mm_loadu_si128(
         (__m128i*)(subkeys+(i*BLOCK_SIZE)));
    output = _mm_aesenc_si128(output,subkey);
  }

  subkey = _mm_loadu_si128(
              (__m128i*)(subkeys+
              (ROUNDS*BLOCK_SIZE)));
  output = _mm_aesenclast_si128(output, subkey);
  _mm_storeu_si128((__m128i*) out, output);
}
```

Figura 2. Cifrado de un bloque

esos valores son 7 y 1, respectivamente, en procesadores posteriores. Esto significa que en un procesador Haswell, como por ejemplo el Core i7-4771 antes mencionado, la ejecución de una ronda de AES toma 7 ciclos de reloj, pero es posible iniciar la ejecución de una segunda instrucción AES un ciclo después de comenzada la primera. Si el programa es capaz de introducir instrucciones AES independientes, es decir, en las cuales la entrada de una no dependa de la salida de la anterior, se podrá obtener el throughput prometido, es decir, se ejecutará una instrucción AES por ciclo.

Cuadro 3. Performance de `AESENC`

Arquitectura	Latencia	Throughput
Haswell	7	1
Ivy Bridge	7	1
Sandy Bridge	7	1
Westmere	6	2

Obviamente el código mostrado no cumple con este requerimiento, ya que las rondas deben ejecutarse en forma secuencial. No es posible comenzar la segunda ronda hasta que no haya finalizado la primera. Aún suponiendo que la planificación de instrucciones es perfecta, la ejecución de las 10 rondas de AES-128 tendrá una cota inferior de $10 \times 7 = 70$ ciclos.

La solución es cifrar varios bloques en paralelo. Supongamos que se decide cifrar 10 bloques en paralelo. Comienza la ejecución de la primera ronda del primer bloque. Un ciclo más tarde comienza la primera ronda del segundo bloque. Un ciclo más tarde comienza la primera ronda del bloque siguiente, y así sucesivamente. De esta forma, la ejecución de las 10 rondas independientes tomará 16 ciclos de reloj, lo cual representa una mejora significativa sobre los 70 ciclos del caso anterior.

Esta optimización requiere que el modo de operación sea paralelizable. De los modos convencionales [8] ECB y CTR son completamente paralelizables, CBC y CFB son paralelizables en el descifrado, pero no en el cifrado, y OFB no es paralelizable. En el caso de los algoritmos que nos ocupan, ambos fueron diseñados teniendo en cuenta esta propiedad. Silver es completamente paralelizable, mientras que AES-CPFB es paralelizable en el cifrado, pero no en el descifrado.

4. Benchmarking de la implementación sin instrucciones AES-NI

Los dos algoritmos evaluados poseen una implementación de referencia desarrollada en lenguaje C de acuerdo a los requisitos de la competencia CAESAR y cuya API se muestra en la figura 3. En la implementación de referencia el énfasis está puesto en la claridad y portabilidad del código, y no en su eficiencia, por lo que no es útil para medir la velocidad de ambos algoritmos. Por lo tanto, y para poder comparar adecuadamente los beneficios obtenidos por la utilización de las instrucciones AES-NI usamos otras versiones más optimizadas.

Como AES-CPFB usa AES como caja negra, su velocidad depende de la versión de AES utilizada. Existen múltiples implementaciones disponibles, entre las que se destacan dos:

- La implementación optimizada de Brian Gladman.
- La implementación optimizada en ANSI C desarrollada por Vincent Rijmen, Antoon Bosselaers y Paulo Barreto.

AES-CPFB posee implementaciones basadas en ambas. Las velocidades son muy similares, y el rendimiento relativo depende de la arquitectura. En algunas computadoras es más rápida la versión de Gladman, y en otras ocurre lo contrario. A los efectos de este trabajo, usamos la versión de Rijmen, Bosselaers y Barreto.

En el caso de Silver, es necesario modificar AES, por lo que usamos una versión desarrollada ad-hoc, basada en la implementación de Rijmen, Bosselaers y Barreto.

En los cuadros 4 y 5 puede verse el resultado de correr el programa de benchmarking en las versiones mencionadas de ambos algoritmos. Todos los resultados fueron obtenidos con el procedimiento de benchmarking descripto en la sección 2, y corresponden, salvo

```
#include "crypto_aead.h"

int crypto_aead_encrypt(
  unsigned char *c,unsigned long long *clen,
  const unsigned char *m,unsigned long long mlen,
  const unsigned char *ad,unsigned long long adlen,
  const unsigned char *nsec,
  const unsigned char *npub,
  const unsigned char *k
)
{
 /*
 * the code for the cipher implementation goes here,
 * generating a ciphertext c[0],c[1],...,c[*clen-1]
 * from a plaintext m[0],m[1],...,m[mlen-1]
 * and associated data ad[0],ad[1],...,ad[adlen-1]
 * and secret message number nsec[0],nsec[1],...
 * and public message number npub[0],npub[1],...
 * and secret key k[0],k[1],...
 */
  return 0;
}

int crypto_aead_decrypt(
  unsigned char *m,unsigned long long *mlen,
  unsigned char *nsec,
  const unsigned char *c,unsigned long long clen,
  const unsigned char *ad,unsigned long long adlen,
  const unsigned char *npub,
  const unsigned char *k
)
{
 /*
 * the code for the cipher implementation goes here,
 * generating a plaintext m[0],m[1],...,m[*mlen-1]
 * and secret message number nsec[0],nsec[1],...
 * from a ciphertext c[0],c[1],...,c[clen-1]
 * and associated data ad[0],ad[1],...,ad[adlen-1]
 * and public message number npub[0],npub[1],...
 * and secret key k[0],k[1],...
 */
  return 0;
}
```

Figura 3. API de CAESAR

que se indique lo contrario, a una computadora de escritorio con procesador Intel® Core i7-4771 con Ubuntu 14.04. El código fue compilado con gcc 4.8.2.

Cuadro 4. AES-CPFB sin instrucciones AES-NI

Op.	44	552	576	1500	65536	PP
Enc	51.30	20.14	20.03	18.55	17.64	20.72
Dec	51.44	20.19	20.11	18.54	17.62	20.75

En ciclos por byte

Los cuadros muestran el resultado de cifrar y descifrar paquetes de 44, 552, 576, 1500 y 65536 bytes. La última columna presenta el promedio ponderado de los cuatro primeros tamaños, según los pesos establecidos en el cuadro 1. Los valores indican ciclos por byte, por lo que menores números indican mejor rendimiento.

Puede apreciarse que:

- El costo de AES-CPFB es aproximadamente un 50 % mayor que el de Silver.

Cuadro 5. Silver sin instrucciones AES-NI

Op.	44	552	576	1500	65536	PP
Enc	34.04	13.23	12.56	11.93	11.33	13.35
Dec	42.58	16.47	15.88	14.91	14.10	16.72

En ciclos por byte

- En AES-CPFB no existe diferencia significativa entre las operaciones de cifrado y descifrado.
- En Silver el cifrado es ligeramente más eficiente que el descifrado.
- En paquetes pequeños, la eficiencia merma considerablemente, debido a los costos fijos de expansión de clave.

5. Benchmarking de la implementación con instrucciones AES-NI

Una implementación trivial de AES con las instrucciones AES-NI utiliza un código similar al de la figura 2. Sin embargo, este código no aprovecha las oportunidades de paralelización que brindan las instrucciones AES-NI. Tal como se describe en la sección 3 es posible cifrar varios bloques en paralelo, con un código similar al que se muestra en la figura 4, el cual cifra 8 bloques en paralelo.

```
void AES_128_encrypt(const unsigned char* in,
                     unsigned char* output,
                     const unsigned char* subkeys){
  int i,j;
  __m128i subkey = _mm_loadu_si128(
        (__m128i*) subkeys);
  __m128i out[8];
  for (j = 0; j < 8; j++)
    out[j] = _mm_xor_si128(_mm_loadu_si128(
        (__m128i*)(in + (j*BLOCK_SIZE))),subkey);
  for (i = 1; i < ROUNDS; i++){
    subkey = _mm_loadu_si128(
        (__m128i*)(subkeys+(i*BLOCK_SIZE)));
    for (j = 0; j < 8; j++)
      out[j] = _mm_aesenc_si128(out[j],subkey);
  }
  subkey = _mm_loadu_si128(
        (__m128i*)(subkeys+(ROUNDS*BLOCK_SIZE)));
  for (j = 0; j < 8; j++)
    out[j] = _mm_aesenclast_si128(out[j],subkey);
  for (j = 0; j < 8; j++)
    _mm_storeu_si128(
        (__m128i*) (output+(j*BLOCK_SIZE)), out[j]);
}
```

Figura 4. Cifrado de 8 bloques en paralelo

Si bien las técnicas de optimización que pueden aplicarse en los dos algoritmos son similares, nos concentraremos en los resultados obtenidos con AES-CPFB. En este algoritmo el cifrado es paralelizable, pe-

ro el descifrado no, por lo que nos permite comparar los resultados de distintas optimizaciones en ambos casos. Silver, por el contrario, es completamente paralelizable, por lo que solo mostraremos los resultados finales de la optimización.

En el cuadro 6 pueden verse los resultados obtenidos con una versión relativamente temprana de la implementación de AES-CPFB con instrucciones AES-NI.

Cuadro 6. AES-CPFB con AES-NI

Op.	44	552	576	1500	65536	PP
Enc	19.99	2.79	2.75	1.78	1.30	3.03
Dec	24.84	8.06	7.98	7.16	6.68	8.34

En ciclos por byte

Puede apreciarse que las mejoras de rendimiento son notables. En mensajes largos, el costo del cifrado se ha reducido de 17.6 cpb a 1.3 cpb, es decir, la velocidad se ha multiplicado por un factor superior a 13. En mensajes cortos la mejora es menor, pero también es importante. En mensajes de 44 bytes se ha reducido de más de 50 cpb a menos de 20 cpb.

También es posible apreciar como el cifrado es apreciablemente más rápido que el descifrado. Esto se debe a que por las características del modo de operación, el descifrado no es paralelizable, ya que para descifrar un bloque es necesario haber descifrado el bloque anterior. Como en la implementación sin instrucciones AES-NI los costos de cifrado y descifrado son equivalentes, es claro que las diferencias que se perciben en este caso se deben a la capacidad de paralelización de las nuevas instrucciones.

Esta versión incluye algunas optimizaciones adicionales, tales como una reducción binomial del grado de paralelización para mensajes cortos. Se definieron funciones que cifran en paralelo 8, 4 y 2 bloques. De esta forma, para cifrar un mensaje de 7 bloques, por ejemplo, primero se cifran 4 bloques en paralelo, luego 2 bloques, y finalmente 1 bloque. Esto permite reducir en forma apreciable el tiempo de procesamiento de mensajes cortos, con una pequeña reducción de la eficiencia en mensajes largos.

Sin embargo aún es posible realizar optimizaciones adicionales:

- El algoritmo requiere la generación de al menos dos claves auxiliares mediante el cifrado con la clave maestra de dos *nonces* derivados del número de mensaje público. Es posible paralelizar estas operaciones con la expansión de la clave maestra, en un procedimiento similar al descripto por Shay Gueron en [10] (AES encryption with On-The-Fly Key Expansion). En este caso, a diferencia del planteado por Gueron, se trata de una expansión de clave ejecutada en paralelo con el cifrado de dos bloques.

- Las dos claves auxiliares pueden expandirse en paralelo.[1]

- Las operaciones `load` y `store` de 128 bits, necesarias para transferir datos entre memoria y registros, son más eficientes que las transferencias de bytes, inclusive cuando se trata de bloques incompletos. Es decir, es más rápido transferir 96 bits con una operación de 128 bits, por más que se desperdicien 32 bits, que hacer la transferencia en unidades más pequeñas. Sin embargo, es necesario tener en cuenta que las operaciones mencionadas no salgan de los buffers proporcionados, tanto para lectura como para escritura. Inclusive dentro del mismo buffer, hay que tener en cuenta que los buffers de entrada y salida podrían apuntar a la misma área de memoria, y debe evitarse sobrescribir las partes del mensaje aún no cifradas.

La aplicación de las optimizaciones mencionadas produce los resultados que se muestran en el cuadro 7.

Cuadro 7. AES-CPFB con AES-NI (final)

Op.	44	552	576	1500	65536	PP
Enc	13.09	2.01	1.92	1.44	1.11	2.21
Dec	18.07	7.04	7.01	6.47	6.16	7.24

En ciclos por byte

El cuadro 8 muestra los resultados correspondientes a Silver.

Cuadro 8. Silver con AES-NI (final)

Op.	44	552	576	1500	65536	PP
Enc	11.85	1.64	1.41	1.10	0.76	1.78
Dec	15.08	2.07	1.80	1.37	0.95	2.25

En ciclos por byte

6. Otras plataformas

A continuación se presentan los resultados de correr el mismo código en distintas plataformas.

[1] En rigor no siempre es necesaria la generación y expansión de las dos claves. Sin embargo, esto sólo ocurre en el infrecuente caso de mensaje nulo, y por otro lado la paralelización de las dos operaciones hace que su costo no sea sustancialmente superior al de generar y expandir una sola clave.

Intel® Core i7-3770k

El procesador Intel® Core i7-3770k (Ivy Bridge) es una generación anterior al Core i7-4771 (Haswell) de las pruebas precedentes. La frecuencia nominal, 3.5 GHz, es la misma, por lo que las diferencias de rendimiento son atribuibles a la diferencia en la arquitectura.

Cuadro 9. AES-CPFB sin AES-NI, Ivy Bridge

Op.	44	552	576	1500	65536	PP
Enc	57.37	22.68	22.57	20.90	19.86	23.33
Dec	57.45	22.65	22.52	20.84	19.82	23.28

En ciclos por byte

Cuadro 10. AES-CPFB con AES-NI, Ivy Bridge

Op.	44	552	576	1500	65536	PP
Enc	12.78	2.19	2.01	1.66	1.32	2.37
Dec	18.12	7.78	7.74	7.25	6.96	7.97

En ciclos por byte

Cuadro 11. Silver sin AES-NI, Ivy Bridge

Op.	44	552	576	1500	65536	PP
Enc	37.06	14.48	13.81	13.16	12.51	14.68
Dec	46.39	17.97	17.31	16.29	15.39	18.25

En ciclos por byte

Cuadro 12. Silver con AES-NI, Ivy Bridge

Op.	44	552	576	1500	65536	PP
Enc	13.25	1.89	1.63	1.31	0.91	2.06
Dec	15.65	2.47	2.21	1.77	1.33	2.66

En ciclos por byte

Como puede verse en los cuadros 9 a 12, el rendimiento en Ivy Bridge es consistentemente inferior al rendimiento en Haswell.

Intel® Core i5-4200U

El procesador Intel® Core i5-4200U es de la misma generación que el Core i7-4771 (Haswell). Es un procesador de bajo consumo, utilizado habitualmente en ultrabooks, y tiene una frecuencia nominal de 1.6 GHz.

Los benchmarks en este procesador pueden verse en los cuadros 13 a 16.

Cuadro 13. AES-CPFB sin AES-NI, i5-4200U

Op.	44	552	576	1500	65536	PP
Enc	74.05	29.33	29.17	26.89	25.44	30.07
Dec	74.66	29.22	29.06	26.75	25.42	29.98

En ciclos por byte

Cuadro 14. AES-CPFB con AES-NI, i5-4200U

Op.	44	552	576	1500	65536	PP
Enc	18.82	2.90	2.76	2.09	1.60	3.18
Dec	25.98	10.15	10.07	9.31	8.86	10.42

En ciclos por byte

Cuadro 15. Silver sin AES-NI, i5-4200U

Op.	44	552	576	1500	65536	PP
Enc	48.39	18.61	17.73	16.89	16.10	18.89
Dec	61.86	23.60	22.78	21.44	20.20	24.05

En ciclos por byte

Cuadro 16. Silver con AES-NI, i5-4200U

Op.	44	552	576	1500	65536	PP
Enc	16.98	2.36	2.01	1.58	1.09	2.56
Dec	21.67	2.99	2.59	1.97	1.37	3.23

En ciclos por byte

7. Conclusiones

La experiencia obtenida en este trabajo de medición y optimización nos permite concluir que:

- Medir adecuadamente el rendimiento de algoritmos criptográficos es una tarea no trivial, ya que existen numerosas oportunidades de realizar mediciones erróneas. No sólo es necesario encontrar la métrica adecuada, sino que las características de los procesadores modernos, tales como ejecución fuera de orden, multiplicidad de núcleos y frecuencia dinámica afectan el proceso de medición.
- Los algoritmos criptográficos que puedan utilizar las nuevas instrucciones AES-NI se verán altamente beneficiados, aún en el caso en que no sean paralelizables. Estos beneficios no se reducen a incrementos de velocidad. La seguridad de los algoritmos se ve mejorada a través de la protección contra *side channels attacks* (como por ejemplo los *cache timing attacks* a los que ha sido sometido AES).
- Si además de poder usar estas instrucciones el algoritmo puede paralelizarse, las mejoras de rendimiento son todavía mayores. Los números presentados en este trabajo hablan por sí solos.

- Distintas optimizaciones funcionan de manera distinta en diferentes plataformas. Las estrategias de optimización de los distintos compiladores hacen que la optimización que mejora el rendimiento en uno de ellos puede empeorarlo con otro.

8. Bibliografía

Referencias

[1] "Specification for the Advanced Encryption Standard (AES)", Federal Information Processing Standards Publication 197, 2001

[2] Joan Daemen y Vincent Rijmen. "AES Proposal: Rijndael". National Institute of Standards and Technology, 1999.

[3] Joan Daemen and Vincent Rijmen, "The Design of Rijndael, AES - The Advanced Encryption Standard", Springer-Verlag 2002 (238 pp.)

[4] Daniel Penazzi y Miguel Montes, "Silver v1", CAESAR website, 2014

[5] Miguel Montes y Daniel Penazzi, "AES-CPFB v1", CAESAR website, 2014

[6] David McGrew y John Viega, "The Security and Performance of the Galois/Counter Mode (GCM) of Operation", In INDOCRYPT, volume 3348 of LNCS, 2004

[7] P. Rogaway, M. Bellare, J. Black, y T. Krovitz, "OCB: a block-cipher mode of operation for efficient authenticated encryption", ACM CCS, 2001

[8] "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide"

[9] Morris Dworking, "NIST Special Publication 800-38A Recommendation for Block Cipher Modes of Operation", 2001 Edition

[10] David McGrew y John Viega, "The Galois/Counter Mode of Operation (GCM)", NIST website,

[11] Shay Gueron, "Intel® Advanced Encryption Standard (AES) New Instructions Set", Intel Corporation, Mayo 2010.