Dos nuevos algoritmos de cifrado autenticado: Silver y CPFB

Miguel Montes[1] and Daniel Penazzi[2]

[1] Instituto Universitario Aeronáutico, Córdoba, Argentina,
mmontes@iua.edu.ar,
[2] Universidad Nacional de Córdoba, Facultad de Matemática, Astronomía y Física, Córdoba, Argentina,
penazzi@famaf.unc.edu.ar

Resumen El presente trabajo resume el proceso de diseño de dos algoritmos de cifrado autenticado, Silver y CPFB, que participan en la competencia CAESAR para la selección de algoritmos de cifrado autenticado. Silver es un algoritmo de cifrado basado en AES-128, mientras que CPFB es un modo de operación. Ambos basan su seguridad en la presunción de que AES es indistinguible de un *random oracle*.

Keywords: criptografía, cifrado autenticado, modos de operación

1. Introducción

Los modos de operación tradicionales (ECB, CBC, OFB, CFB y CTR)[1] se enfocan únicamente en el problema de la confidencialidad, y no pretenden resolver el problema de la autenticación. La forma normal de resolver este último problema es combinar el algoritmo de cifrado con un código de autenticación de mensaje (MAC), como por ejemplo HMAC o CBC-MAC. Sin embargo, la adecuada combinación de ambas primitivas es compleja, y existen muchos ejemplos de implementaciones en las que esta combinación se ha realizado en forma incorrecta.

Es por ello que en la última década se ha comenzado a reconocer la importancia de los modos de operación de cifrado autenticado (AE), y en particular, la de aquellos que permiten también la autenticación de datos adicionales (AEAD). Entre ellos podemos mencionar OCB, CCM y GCM.

OCB [2](Offset Codebook Mode) es un modo diseñado por Phillip Rogaway. Es muy rápido, en el sentido que impone muy baja sobrecarga sobre el algoritmo de cifrado subyacente, pero presenta el incoveniente de que utiliza técnicas patentadas.

CCM [3](Counter with CBC-MAC) fue desarrollado por Russ Housley, Doug Whiting y Niels Ferguson. Es considerablemente más lento que OCB, debido a que requiere dos aplicaciones del algoritmo de cifrado por bloque, pero muchos consideran esta disminución en el rendimiento un adecuado precio a pagar por la ausencia de patentes.

GCM [4](Galois/Counter Mode) diseñado por David McGrew y John Viega usa CTR como modo de cifrado para obtener confidencialidad, y autenticación basada en la multiplicación en $GF(2^{128})$. Es un modo con baja sobrecarga, y no afectado por patentes. Sin embargo, tiene el inconveniente de que una tag de n bits no proporciona n bits de seguridad, por lo que no se recomienda su uso con tags de longitud pequeña.

1.1. CAESAR

CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) es una competencia para seleccionar una cartera de algoritmos de cifrado autenticado que

1. presenten ventajas sobre AES-GCM, y
2. sean adecuados para ser ampliamente adoptados.

La competencia está coordinada por Daniel Bernstein, y responde a la tradición criptográfica de seleccionar algoritmos por este mecanismo. Ejemplos de esta tradición son las competencias organizadas por NIST para AES y SHA-3; y eSTREAM, organizada por la Unión Europea para la selección de algoritmos de cifrado de flujo.

La fecha de cierre para la presentación de algoritmos fue el 15 de marzo de 2014, y al momento de escribir este trabajo se encuentra en proceso la recepción de implementaciones de referencia.

De acuerdo con los términos del llamado, un algoritmo de cifrado autenticado es una función con cinco entradas (inputs) y una salida (output), todas ellas consistentes en secuencias de bytes (byte-strings). Las cinco entradas son:

- Texto claro (plaintext) de longitud variable.
- Datos asociados (associated data) de longitud variable.
- Número de mensaje secreto, de longitud fija.
- Número de mensaje público, de longitud fija.
- Clave, de longitud fija.

La salida es un texto cifrado (ciphertext) de longitud variable.

Las primeras cuatro entradas tienen distintos propósitos:

	Integridad	Confidencialidad	Nonce
Texto claro	Sí	Sí	No
Datos asociados	Sí	No	No
Número de mensaje secreto	Sí	Sí	Sí
Número de mensaje público	Sí	No	Sí

No se requiere que los algoritmos presentados soporten el uso de números de mensaje, ya sean secretos o públicos. Si se utilizan números de mensaje, el diseñador puede exigir que dicho número sea un *nonce*, es decir, que no se repita durante el tiempo de uso de una clave dada. Sin embargo, no puede exigirse de

dicho número ninguna otra característica adicional, tal como que se generen en forma secuencial, o que tengan cierta estructura.

Es admisible que el algoritmo pierda toda seguridad si el usuario repite un *nonce*

2. Criterios usados para el diseño

Al definir nuestra participación en la competencia, decidimos presentar dos algoritmos, con objetivos ligeramente diferentes, pero con ciertas ideas básicas en común.

2.1. Uso de AES

Si bien evaluamos el diseño de un algoritmo de cifrado de flujo ad-hoc, finalmente decidimos basar ambos algoritmos en AES. Esta decisión se basa en los siguientes hechos:

1. AES es probablemente el algoritmo de cifrado más analizado de la historia, lo que permite depositar considerable confianza en su diseño.
2. Es un algoritmo implementado en una amplia variedad de plataformas, tanto en software como en hardware, lo cual facilita el despliegue de cualquier modo de operación, algoritmo o protocolo que lo use, ya sea en su totalidad, como caja negra, o parcialmente, mediante el uso de sus componentes.
3. En arquitecturas x86, las nuevas instrucciones AES-NI aceleran considerablemente el uso de AES, y solucionan el problema de los "side-channel attacks" asociados con el uso de tablas. Estas instrucciones está disponibles en los procesadores Intel desde la microarquitectura Westmere en adelante, y en los procesadores AMD, desde Bulldozer.

Silver y CPFB usan AES de formas diferentes, pero comparten una idea fundamental: usar "sólo" AES. Es decir, no utilizar otras operaciones (tales como multiplicaciones es $GF(2^{128})$) aparte de las componentes de AES y operaciones aritméticas y lógicas básicas.

En el caso de Silver, buscamos un algoritmo con mínima sobrecarga con el objetivo de lograr máxima velocidad, competitiva con OCB. Para lograr este objetivo sacrificamos el uso de AES como caja negra, y construimos un "tweaked cipher" mediante la modificación de las claves de ronda. El algoritmo resultante ya no es AES, sino un AES "tweaked", pero que retiene sus propiedades y permite usar optimizaciones tales como las instrucciones AES-NI.

En CPFB la decisión fue diseñar un modo de operación, en el cual AES pueda usarse como caja negra. Este enfoque permite una implementación más sencilla, y, en el caso en que AES resultara vulnerable, facilita su reemplazo por otro. CPFB puede utilizarse sin modificaciones con cualquier algoritmo de cifrado con bloques de 128 bits, y puede adaptarse fácilmente a mayores tamaños de bloque.

2.2. Uso del *nonce*

Ambos algoritmos soportan el uso de números de mensaje públicos, pero no el de números de mensaje secretos. Se requiere que el número de mensaje sea un *nonce*, y no se garantiza seguridad si este se repite.

La forma de usar el *nonce* es distinta del tradicional recurso de utilizarlo como vector de inicialización (IV). Tanto en Silver como en CPFB el *nonce* se utiliza para la derivación de clave. La clave principal, o maestra, no se utiliza nunca para cifrado o autenticación, sino solo para la generación de claves auxiliares a partir del *nonce*.

2.3. Orientación a bytes

Silver y CPFB procesan secuencias de bytes. Muchos algoritmos criptográficos están definidos en términos de secuencias de bits, pero:

1. en la práctica rara vez es necesario procesar mensajes cuya longitud no sea un número entero de bytes, y
2. complica enormemente certificar que una implementación cumple con la especificación.

Por esas razones, ambos algoritmos están definidos en términos de bytes, y se asume que la longitud de cualquier mensaje es un múltiplo de 8 bits.

3. Notación

$\oplus$ denota el o-exclusivo bit a bit (bitwise XOR).

$MSB_m(X)$ es la secuencia de bits consistente en los m bits más significativos de la secuencia de bits X.

$||$ denota concatenación.

$\text{AES}_K(X)$ denota la aplicación de la función de cifrado de AES con clave K al bloque X.

$|X|$ denota la longitud en bits de la secuencia de bits X.

$\{0\}^n$ denota una secuencia de ceros de n bits de longitud.

$[x]_s$ es la representación binaria del entero no negativo x como una secuencia de bits de longitud s, donde $x < 2^s$. El orden de los bits en esa representación (endianess) depende del algoritmo. Silver usa una representación little-endian, y CPFB una representación big-endian.

4. Silver

4.1. Cifrado y autenticación

Entradas. Las entradas del algoritmo de cifrado son un texto plano P, datos adicionales que deben autenticarse pero no cifrarse (por ejemplo, headers) A, un número de mensaje público *npub* y una clave *key*. *npub* debe ser un *nonce*, i.e., no repetirse en una misma clave y en Silver fijamos que el *nonce* debe ser un bloque de 128 bits.

Salidas El resultado es un par (C, T): un texto cifrado no autenticado C y un autenticador o *tag* T de 16 bytes (o menos, si se la quiere truncar. Sin embargo no deben usarse tags de distinto tamaño con la misma clave). La longitud de C es la misma de la del texto plano P, es decir, salvo por la *tag*, no hay expansión de texto.

Descripción y motivación. El diseño de Silver tuvo dos objetivos básicos: el primero, por supuesto, es la seguridad, para lo cual decidimos basar la construcción en AES. Segundo, deseábamos un cifrador que fuera más rápido que AES-GCM y capaz de competir con OCB. Para lograr esas velocidades, se requiere que haya una capacidad de paralelización. ECB es paralelizable pero tiene el problema obvio de que bloques iguales se cifran igual, por lo que pensamos en las ideas de [8] sobre usar un "tweaked block cipher" para crear un sistema de cifrado/autenticación. OCB [2] usa esa estructura básica, con un tweak que consiste en xorear al texto plano y al texto cifrado una máscara que cambia de bloque en bloque, usando AES como caja negra. No cualquier máscara es segura, y además intentar algo similar a esto puede infringir la patente de OCB. Pensamos en que en vez de tratar de usar AES como caja negra, podíamos meternos dentro de la estructura de AES y usar un tweak interno en vez de externo. Un cambio de las claves de ronda es una idea natural. Una posibilidad que pensamos fue permutar las claves de ronda y luego de algunos bloques obtener una nueva clave de ronda, eliminando otra, y permutar otra vez. El problema con esto es que también queríamos una propiedad de acceso aleatorio, i.e., que se pueda cifrar los bloques en cualquier orden, por ejemplo para no tener que recalcular todo cuando se cambia un sector de un disco o una entrada en una base de datos. Por lo que preferimos cambiar las claves usando simplemente un contador fácilmente calculable, pero secreto, que depende de la clave y el *nonce*. La siguiente pregunta fue cuales claves cambiar. Cambiar todas las claves puede producir una disminución en la eficiencia, y de todos modos no es claro que sea seguro, dado los recientes ataques de claves relacionadas en por ejemplo [6]. Por otro lado, la probabilidad diferencial de cualquier camino diferencial en 4 rondas de AES es menor o igual a 2^{-150} por lo que si solo se cambian claves cada 4 rondas el ataque de [6] no puede aplicarse. Así nos decidimos a cambiar las claves de las rondas 1,5 y 9 de AES.

Una segunda cuestión era el tratamiento del *nonce*. Una posibilidad sería usarlo para determinar el estado inicial del contador. Por motivos de seguridad sería razonable no dejar que el rival pueda controlar el contador, por lo que habría que cifrar el *nonce* antes de usarlo. Y ya habiendo hecho ese paso, naturalmente ocurre la idea de además expandir este *nonce* interno, obteniendo nuevas claves de ronda, con lo cual todo el cifrador cambiaría de un *nonce* al otro, y no solo las claves de ronda de algunas rondas. Aquí hay una pérdida de eficiencia para mensajes cortos, puesto que la expansión de clave de AES, si bien mucho más rápida que otras (por ejemplo que la de Twofish) es sin embargo más lenta que un cifrado completo de AES (al menos con las instrucciones AES-NI). Pero este proceso bloquea tantos posibles ataques que nos pareció razonable implementar-

lo. Hacer que las claves de sesión dependan del *nonce* sin embargo es peligroso si no se hace bien, pues podría existir un ataque de colisión de claves, si, por ejemplo, usáramos directamente el cifrado del *nonce* como clave a expandir. Dos algoritmos presentados en la competición CAESAR, Calico y Avalanche usaban el *nonce* en la clave, pero no tomaron precauciones para prevenir esta situación y fueron rápidamente atacados. Afortunadamente nosotros previmos este ataque: las claves de ronda son una mezcla de las claves de ronda correspondientes a la clave maestra con las claves de ronda correspondientes al cifrado del *nonce*. Esto tiene un beneficio añadido en que ahora las claves de ronda dependen en forma muy altamente no lineal de la clave y el *nonce*, con lo cual también se anula parte del ataque de [6] contra AES normal, que aprovecha que la expansión de clave de AES es, si bien no lineal, no lo suficientemente no lineal. En resumen, tenemos lo siguiente, en donde $+$ es la suma de $(\mathbb{Z}/2^{64}\mathbb{Z}) \times (\mathbb{Z}/2^{64}\mathbb{Z})$, $i * M$ es $M + M + \cdots + M$ (i veces) y $AESroundkeys_i(K)$ es la i-ésima clave de ronda generada por AES a partir de una clave K.

GenerateKeys$(npub, key)$
1 $\kappa = \text{AES}_{key}(npub)$
2 $roundkey_0 = AESroundkey_0(key) \oplus AESroundkey_1(\kappa)$
3 **for** $i \leftarrow 2$ **to** 8 $roundkey_i = AESroundkey_i(key) \oplus AESroundkey_i(\kappa)$
4 $roundkey_{10} = AESroundkey_{10}(key) \oplus AESroundkey_{10}(\kappa)$
5 **for** $i \leftarrow 1, 9$ $roundkey_i = AESroundkey_i(key)$
6 $IC \leftarrow AESroundkey_9(\kappa)\text{OR}([1]_{64} \,||\, [1]_{64})$
7 **return** $(roundkeys, \kappa, IC)$

TAES$(X, roundkeys, \kappa, counter)$
1 **for** $i \leftarrow 0$ **to** 10
2 **if** $(i \neq 1, 5, 9)$
3 **then** $temprkeys_i = roundkeys_i$
4 **else** $temprkeys_i = roundkeys_i \oplus (\kappa + counter)$
5 $Y =$encrypt X using AES with $temprkeys$
6 **return** Y

Para cifrar simplemente procesamos todo en modo ECB, cambiando el contador de bloque en bloque. Para autenticar, guardamos el xor de todos los textos planos, de los textos cifrados (estos últimos con un post cifrado ligero extra) y de los cifrados ocultos de los datos asociados (con un contador distinto). Si el bloque de datos final es parcial, lo ciframos usando modo CTR, para evitar alargar el texto, pero también aplicamos relleno (*padding*) al bloque y lo ciframos para usar este último cifrado en la *tag*. Esto involucra una pérdida de velocidad significativa para bloques parciales. Una alternativa es "ciphertext stealing". El problema es que si el mensaje entero es de menos de 128 bits, no habría de donde hacer el *steal*, salvo que se hiciera directamente de la *tag*. Pensamos en esta opción, pero obligaba a usar siempre tags de 128 bits, sin poder truncarlas. En el caso de bloque final de datos asociados parcial, ahí no hay problema, simplemente aplicamos relleno y ciframos.

ENCRYPT$(P, roundkeys, \kappa, IC)$

1 $s = \lceil |P|/128 \rceil$
2 $\ell = |P|/8 \mod 16$
3 partir P en $P_1||P_2||....||P_t$ c/u de 128 bits excepto quizás t.
4 $XT \leftarrow \{0\}^{128}$
5 $counter \leftarrow \{0\}^{128}$
6 **for** $i \leftarrow 1$ **to** $s-1$
7 **do** $counter \leftarrow counter + IC$
8 $C_i \leftarrow \text{TAES}(P_i, roundkeys, \kappa, counter)$
9 $XT \leftarrow XT \oplus P_i \oplus (C_i + \kappa + counter)$
10 **if** $\ell \neq 0$
11 **then** $bP = \left[\frac{|P|}{8}\right]_{64}$
12 $counter \leftarrow counter + IC$
13 $tmp = \text{TAES}((bP||bP), roundkeys, \kappa, counter)$
14 Partir tmp en bytes $tmp_1||tmp_2||...||tmp_{16}$
15 $C_s = P_s \oplus (tmp_1||...||tmp_\ell)$
16 $B = P_s||tmp_{\ell+1}||...||tmp_{15}||\,[\ell]_8$
17 $counter \leftarrow counter + IC$
18 $XT \leftarrow XT \oplus \text{TAES}(B, roundkeys, \kappa, counter)$
19 **else** $counter \leftarrow counter + IC$
20 $C_s \leftarrow \text{TAES}(P_s, roundkeys, \kappa, counter)$
21 $XT \leftarrow XT \oplus P_s \oplus (C_s + \kappa + counter)$
22 **return** (C, XT)

PROCESSAD$(A, roundkeys, \kappa, IC)$

1 $t = \lceil |A|/128 \rceil$
2 $r = |A|/8 \mod 16$
3 **if** $(r \neq 0)$
4 **then** $A^* = A||\,[1]_8\,||\{0\}^{15-r}$
5 **else** $A^* = A$
6 partir A^* en $A_1||A_2||....||A_t$ c/u de 128 bits.
7 $AT \leftarrow \{0\}^{128}$
8 $AIC \leftarrow IC\&(\{1\}^{64}||\{0\}^{64})$
9 $counter \leftarrow \{0\}^{128}$
10 **for** $i \leftarrow 1$ **to** $t-1$
11 **do** $counter \leftarrow counter + AIC$
12 $AT \leftarrow AT \oplus \text{TAES}(A_i, roundkeys, \kappa, counter)$
13 **if** $r \neq 0$
14 **then** $counter \leftarrow \{0\}^{128}$
15 **else** $counter \leftarrow counter + AIC$
16 $AT \leftarrow AT \oplus \text{TAES}(A_t, roundkeys, \kappa, counter)$
17 **return** AT

La *tag* final es el cifrado de $AT \oplus XT$ usando contador $b_A||b_P$ donde b_A y b_P son las longitudes en bytes de A y P en little-endian, y además se per-

muta el orden de las claves con la permutación $(2,3,4,6,7,8,10,0)(9,1,5)$. (esa permutación mantiene el cambio de contador en las rondas 1,5,9). Es decir:

EncryptAndAuthenticate$(A, P, npub, key)$
1 $(roundkeys, \kappa, IC) \leftarrow$ GenerateKeys$(npub, key)$
2 $AT \leftarrow$ ProcessAD$(A, roundkeys, \kappa, IC)$
3 $(C, XT) \leftarrow$ Encrypt$(P, roundkeys, \kappa, IC)$
4 $g \leftarrow \left(\left[\frac{|A|}{8}\right]_{64} || \left[\frac{|P|}{8}\right]_{64}\right)$
5 $permroundkeys = roundkeys$ permutadas con $(2,3,4,6,7,8,10,0)(9,1,5)$
6 $T \leftarrow$ Encrypt$(AT \oplus XT, permroundkeys, \kappa, g)$
7 **return** (C, T)

Todos estos detalles tienen como objetivo bloquear ciertos ataques. Por ejemplo, al usar tanto el texto plano como el cifrado para la *tag* del cifrado, pero solo el texto cifrado (que el adversario, en este caso, nunca ve) para los datos a autenticar, nos aseguramos de que ambas partes se tratan de forma distinta, pero además (para prevenir el caso donde el xor de los textos planos es cero) el IC usado es distinto en ambos. (esto además es necesario para ciertos detalles técnicos de la prueba de seguridad). CALICO no tomó precauciones para distinguir datos de texto plano con datos asociados, y fue atacado por esto. La *tag* se calcula con un orden distinto de las claves distinto al resto para asegurarse que ese cifrado no se usa antes. Se usan repetidas veces las longitudes en bytes y se usa un padding que también tiene en cuenta las longitudes para forzar al adversario a tener que usar dos textos de la misma longitud si quiere realizar una falsificación. (algunos cifradores presentados a la competencia no se aseguraron de evitar este tipo de ataque y sufrieron por ello. Por ejemplo en SCREAM (versión 1) la *tag* de $M||0||P$ es la misma que la de $M||P$, así que los autores se vieron forzados a cambiar SCREAMv1 por SCREAMv2).

El enmascaramiento del texto cifrado con el contador en la construcción de XT sirve dos propósitos: diferencia aún más AT de XT y da cierto grado de protección si por error se repite el *nonce*.

4.2. Descifrado y validación

Entradas. Las entradas del algoritmo de descifrado son un texto cifrado C, un autenticador T, datos adicionales A, un número de mensaje público *npub* y una clave *key*.

Salidas. Si el mensaje es válido, la salida es P, y error en caso contrario.

Descripción.

Decrypt$(C, roundkeys, \kappa, IC)$
1 Lo mismo que Encrypt cambiando P por C en lineas 1,2,3,8,15,20 y 22
2 y $TAES$ por $TAES^{-1}$ en lineas 8 y 20.

DecryptAndVerify$(A, C, T, npub, key)$
1 $(roundkeys, \kappa, IC) \leftarrow$ GenerateKeys$(npub, key)$
2 $AT \leftarrow$ ProcessAD$(A, roundkeys, \kappa, IC)$
3 $(P, XT) \leftarrow$ Decrypt$(C, roundkeys, \kappa, IC)$
4 $g \leftarrow \left(\left[\frac{|A|}{8}\right]_{64} || \left[\frac{|P|}{8}\right]_{64}\right)$
5 $permroundkeys = roundkeys$ permutadas con $(2,3,4,6,7,8,10,0)(9,1,5)$
6 $T' \leftarrow$ Encrypt$(AT \oplus XT, permroundkeys, \kappa, g)$
7 **if** $T \neq T'$
8 **then return** $\perp$
9 **return** P

4.3. Velocidad

Con instrucciones AES-NI en Haswell, Silver cifra a 0,73 cpb (ciclos por byte) y descifra a 0,81 cpb para mensajes largos, 1 cpb/1,2 cpb para 1536 bytes y 10,8/9,6 cpb para 44 bytes.

Sin instrucciones AES-NI se obtiene alrededor de 11,45/12,9 cpb para mensajes largos, 11,85/13,59 para 1536 bytes y 30,4/28,2 cpb para 44 bytes.

5. CPFB

CPFB (Counter/Plaintext Feedback) es un algoritmo que combina características de los modos CTR y PFB. Este último es un modo poco usado debido a que no presenta ventajas sobre CFB, y es susceptible a ataques de texto claro elegido. Sin embargo, su combinación con el modo CTR brinda características de privacidad equivalentes a este último, y el uso de realimentación del texto claro en la secuencia cifrante permite utilizar esta última para obtener un autenticador.

A continuación se describe el algoritmo en forma abreviada. Para la especificación completa, ver [10].

5.1. Cifrado y autenticación

Entradas. Las entradas del algoritmo de cifrado son un mensaje M, datos adicionales AD, un número de mensaje público *npub* y una clave *key*.

El mensaje M. El mensaje M es una secuencia de bytes al cual se le brinda confidencialidad e integridad. Para procesarlo se divide en bloques $M_1, M_2, \ldots, M_n$ donde $|M_i| = 96$. Si la longitud del mensaje no es múltiplo de 96 bits, el último bloque M^*_{n+1} se rellena con la cantidad de ceros necesaria para que así sea.

Los datos adicionales AD. Los datos adicionales son una secuencia de bytes, a los cuales se les proporciona integridad pero no confidencialidad. Para procesar AD se lo divide en bloques $AD_1, AD_2, \ldots, AD_m$ donde $|M_i| = 96$. Si la longitud total no es múltiplo de 96 bits, el último bloque AD^*_{m+1} se rellena con la cantidad de ceros necesaria para que así sea.

El número de mensaje público *npub*. El número de mensaje público puede tener entre 8 y 15 bytes, y debe ser un *nonce*, es decir, no puede reusarse durante el tiempo de vida de la clave. Este número se usa para generar un *nonce* de 128 bits.

La clave *key*. La clave puede tener 128 o 256 bits. Nunca se usa en forma directa, sino sólo para generar, a partir del *nonce*, una serie de claves auxiliares $\kappa_0, \kappa_1, \ldots, \kappa_k$, donde k depende de la longitud de *npub*.

Salidas. Las salidas son un texto cifrado C y un autenticador (*tag*) T.

El texto cifrado C. C es una secuencia de bytes de la misma longitud que M.

El autenticador T. El autenticador o *tag* es una secuencia de bytes del tamaño de un bloque del algoritmo de cifrado (16 bytes). Pueden obtenerse autenticadores de menor longitud mediante truncamiento. No deben usarse autenticadores de distinto tamaño con una misma clave.

Descripción. Inicialmente se generan las claves κ_0, κ_1. De acuerdo al tamaño del mensaje, puede ser necesario generar claves adicionales[3].

EncryptAndAuthenticate($AD, M, npub, key$)
1 $(\kappa_0, \kappa_m) \leftarrow$ GenerateKeys($npub, key$)
2 $mlen \leftarrow |M|/8$
3 $adlen \leftarrow |AD|/8$
4 $X_{AD} \leftarrow$ ProcessAD(AD, κ_0)
5 $(C, X_M) \leftarrow$ Encrypt(M, κ_m, κ_0)
6 $L \leftarrow \mathrm{AES}_{\kappa_0}([mlen]_{64} \,||\, [adlen]_{32} \,||\, \{0\}^{32})$
7 $T \leftarrow \mathrm{AES}_{\kappa_0}(X_{AD} \oplus X_M \oplus L)$
8 **return** (C, T)

Para procesar los datos adicionales, cada bloque de 96 bits se concatena con un contador de 32 bits y se cifra con κ_0. El procedimiento retorna el or-exclusivo de los bloques cifrados, los cuales constituyen un autenticador parcial de AD. Para procesar el mensaje cada bloque de 96 bits se concatena con un contador de 32 bits, se hace un or-exclusivo con κ_0 y se cifra con κ_1. El resultado se utiliza como secuencia cifrante para obtener el texto cifrado, y se combina con los resultados de los demás bloques para obtener un autenticador parcial de M. Como bloque inicial se utiliza un bloque de ceros, que participa del cifrado pero no de la autenticación. Finalmente se combinan ambos autenticadores parciales con el cifrado de un bloque que codifica las longitudes del mensaje (64 bits) y de los datos adicionales (32 bits).

[3] Por claridad se presenta una versión simplificada, en la cual el tamaño máximo del mensaje es $2^{32} - 1$ bloques de 96 bits. Para ver una descripción completa del procedimiento de generación de claves, ver [10]

```
ProcessAD(AD, κ)
 1  n = ⌊|AD|/96⌋
 2  r = |AD|  mód 96
 3  X ← {0}^128
 4  counter ← 0
 5  for i ← 1 to n
 6        do counter ← counter +1
 7           X ← X ⊕ AES_κ(AD_i || [counter]_32)
 8  if r ≠ 0
 9     then counter ← counter +1
10           X ← X ⊕ AES_κm(AD*_n+1 || {0}^96−r || [counter]_32)
11  return X
```

```
Encrypt(M, κ_m, κ_0)
 1  n = ⌊|M|/96⌋
 2  r = |M|  mód 96
 3  X ← {0}^128
 4  counter ← 0
 5  stream ← AES_κm(X ⊕ κ_0)
 6  for i ← 1 to n
 7        do C_i ← M_i ⊕ MSB_96(stream)
 8           counter ← counter +1
 9           stream ← AES_κm((M_i || [counter]_32) ⊕ κ_0)
10           X ← X ⊕ stream
11  if r ≠ 0
12     then C*_n+1 ← M*_n+1 ⊕ MSB_r(stream)
13           counter ← counter +1
14           stream ← AES_κm((M*_n+1 || {0}^96−r || [counter]_32) ⊕ κ_0)
15           X ← X ⊕ stream
16  return (C, X)
```

5.2. Descifrado y validación

Entradas. Las entradas del algoritmo de descifrado son un texto cifrado C, un autenticador T, datos adicionales AD, un número de mensaje público *npub* y una clave *key*. Estos cuatro últimos responden a las definiciones de la sección 5.1.

El texto cifrado C. El texto cifrado C es una secuencia de bytes. Para procesarlo se divide en bloques $C_1, C_2, \ldots, C_n$ donde $|C_i| = 96$. Si la longitud C no es múltiplo de 96 bits, puede existir un último bloque C^*_{n+1}. No es necesaria la utilización de relleno (*padding*).

Salidas. Si el el mensaje es válido, la salida es M, y error en caso contrario

Cada bloque se concatena con un contador de 32 bits, y se cifra con AES usando una clave κ_j. La secuencia cifrante así generada se usa para cifrar el

texto claro, y para producir el autenticador. Dado que el algoritmo se usa como cifrado de flujo, el descifrado es simétrico, y sólo se requiere la función de cifrado de AES (no su inversa).

Descripción. La generación de claves y el procesamiento de los datos adicionales son idénticos al presentado en la sección 5.1.

DecryptAndVerify$(AD, C, T, npub, key)$

1 $(\kappa_0, \kappa_m) \leftarrow$ GenerateKeys$(npub, key)$
2 $mlen \leftarrow |M|/8$
3 $adlen \leftarrow |AD|/8$
4 $X_{AD} \leftarrow$ ProcessAD(AD, κ_0)
5 $(M, X_M) \leftarrow$ Decrypt(C, κ_m, κ_0)
6 $L \leftarrow \mathrm{AES}_{\kappa_0}([mlen]_{64} \,||\, [adlen]_{32} \,||\{0\}^{32})$
7 $T' \leftarrow \mathrm{AES}_{\kappa_0}(X_{AD} \oplus X_M \oplus L)$
8 **if** VerifyTag(T, T')
9 **then return** M
10 **return** $\perp$

Para procesar el texto cifrado, cada bloque del texto se descifra realizando un or-exclusivo del mismo con el cifrado del bloque de texto claro anterior, concatenado con un contador de 32 bits, y al cual se le realiza un or-exclusivo con κ_0 previo al cifrado. La secuencia cifrante se utiliza para producir un autenticador parcial. El calculo del autenticador final se realiza de la misma forma que en el procedimiento de cifrado. Si el autenticador generado es igual T, se devuelve el mensaje descifrado. En caso contrario, se devuelve error, denotado en el pseudocódigo por $\perp$.

Decrypt(C, κ_m, κ_0)

1 $n = \lfloor |M|/96 \rfloor$
2 $r = |C| \mod 96$
3 $X \leftarrow \{0\}^{128}$
4 $counter \leftarrow 0$
5 $stream \leftarrow \mathrm{AES}_{\kappa_m}(X \oplus \kappa_0)$
6 **for** $i \leftarrow 1$ **to** n
7 **do** $M_i \leftarrow C_i \oplus \mathrm{MSB}_{96}(stream)$
8 $counter \leftarrow counter + 1$
9 $stream \leftarrow \mathrm{AES}_{\kappa_m}((M_i || [counter]_{32}) \oplus \kappa_0)$
10 $X \leftarrow X \oplus stream$
11 **if** $r \neq 0$
12 **then** $M^*_{n+1} \leftarrow C^*_{n+1} \oplus \mathrm{MSB}_r(stream)$
13 $counter \leftarrow counter + 1$
14 $stream \leftarrow \mathrm{AES}_{\kappa_m}((M^*_{n+1} || \{0\}^{96-r} || [counter]_{32}) \oplus \kappa_0)$
15 $X \leftarrow X \oplus stream$
16 **return** (M, X)

5.3. Motivación de algunas características

Hay algunos detalles que tienen como objetivo bloquear ciertos ataques. Por ejemplo, usar κ_0 para los datos asociados y κ_m para el mensaje tiene, ademas de un objetivo técnico que facilita la prueba de autenticación, el objetivo de aseguramos de diferenciar ambas partes. Como mencionamos antes, algunos otros algoritmos, como Calico, no tomaron precauciones para distinguir datos de texto plano con datos asociados, con consecuencias nefastas. En el cálculo de la tag se usan las longitudes en bytes de los datos asociados y el mensaje para forzar al adversario a tener que usar dos textos de la misma longitud si quiere realizar una falsificación. Como dijimos antes, algunos cifradores presentados a la competencia no se aseguraron de evitar este tipo de ataque y sufrieron por ello. Se usa κ_0 en el cálculo de la tag y no κ_m pues las salidas de los encriptamientos con κ_m son vistos por el adversario, pero los anteriores usos de κ_0 no. El xor del bloque con κ_0 en la linea 9. de ENCRYPT tiene el siguiente propósito: sin ese xor, el cifrado depende exclusivamente de κ_m, y permite montar un ataque colisión de clave. Al usar ese xor, el cifrado depende tanto de κ_m como de κ_0 y el ataque se anula. No es necesario tomar una precaución similar con

el procesamiento de los datos asociados, pues las salidas de estos cifrados nunca son vistas por el adversario.

5.4. Velocidad

La velocidad usando instrucciones AES-NI es de 1,1 cpb para cifrar mensajes largos y 4,4 cpb para descifrarlos. Para mensajes de 1536 bytes las velocidades son 1,5 cpb y 4,8 cpb respectivamente, y para mensajes de 44 bytes son 14 cpb y 13 cpb. Sin usar instrucciones AES-NI, las velocidades son 28.5 cpb/28,13 cpb para largos, 30 cpb/29,6 cpb para 1536 bytes y 77 cpb/56 cpb para 44 bytes

6. Seguridad

Los detalles técnicos son demasiado largos para incluirlos en este trabajo, pero la seguridad de ambos algoritmos se basa en la suposición de que AES es indistinguible de random.

En el caso de Silver, algunas claves de ronda cambian de bloque en bloque y la totalidad de las claves cambian de mensaje en mensaje, por lo que las salidas de cada llamada a AES serán independientemente aleatorias una de otra. Esto implica que un adversario será incapaz de distinguir Silver de un cifrador que simplemente toma (N, P) y devuelve una serie aleatoria de bytes.

En el caso de CPFB la prueba es todavía más fácil, pues al ser CPFB una variación del modo CTR, su seguridad se reduce a la prueba de seguridad del modo counter. La ventaja del "P" en "CPFB" no es tanto para privacidad sino que permite autenticar sin necesidad de usar un algoritmo extra de autenticación (como GHASH en AES-GCM).

En cuanto a incapacidad de falsificación, básicamente si el adversario produce algo con un *nonce* que no ha sido usado nunca, puesto que tanto en Silver como

en CPFB las claves dependen del *nonce*, todas las claves serán aleatoriamente distintas de cualquiera usada antes y todas la salidas y en particular la *tag*, serán aleatorias.

Si por el contrario el adversario produce un texto (A, C, T) con un *nonce* que se uso antes entonces si la longitud de A o la de C difiere de la vez anterior, como tanto en Silver como CPFB se incluyen las longitudes en el calculo de la *tag*, nos aseguramos que la *tag* producida ahora será aleatoria. Si las longitudes son iguales, y hay un cambio en A, entonces en ambos casos el cifrado del bloque en que difieren sera aleatoriamente nuevo, y la *tag* parcial correspondiente al procesamiento de datos asociados nuevos diferirá aleatoriamente de la vieja, y la *tag* final solo será igual con probabilidad 2^{-128}.

Si el cambio se produce en C, debemos analizar Silver y CPFB por separado.

En Silver el razonamiento es parecido pero ahora tiene que ver conque al descifrar el bloque de texto cifrado distinto, el texto plano que se obtiene sera aleatoriamente distinto del usado antes, y el XT nuevo sera aleatoriamente distinto del viejo puesto que los textos planos forman parte del calculo de XT. Una ventaja de Silver respecto de otros esquemas es que también resiste falsificación del texto plano, con un razonamiento parecido al anterior, pues ahora el texto cifrado seria el que difiere aleatoriamente y XT también involucra los textos cifrados.

En el caso de CPFB, la *tag* involucra no los textos planos en si mismos, sino la secuencia cifrante. Sin embargo, puesto que la secuencia cifrante es el cifrado de un bloque de 128 bits que contiene un bloque de 96 bits del texto plano, cualquier variación en los textos planos producida por un cambio en el texto cifrado provocará que el xor de los bloques de la stream sea en el nuevo caso aleatoriamente distinto del viejo. Esta es la ventaja de la "P" en "CPFB". Edemas, este razonamiento muestra que CPFB al igual que Silver resiste intentos de falsificación del texto plano.

Otras ventajas que estos dos algoritmos tienen respecto de AES-GCM son que no necesitan usar implementaciones de multiplicación en cuerpos finitos, pueden procesar mensajes más largos y la seguridad de la *tag* equivale a su longitud.

Respecto de OCB tienen la ventaja que pueden resistir ataques de falsificación de texto plano (en OCB es trivial hacer tal ataque), además de que no están patentados.

En el caso de Silver, su velocidad es competitiva respecto de OCB, y además, una ventaja respecto de AES-GCM u OCB es que tiene cierto grado de resistencia, tanto para privacidad como para autenticación, si el *nonce* se repite accidentalmente, incluso más de una vez (pero no estamos pensando en una repetición, digamos de 2^{64} veces el *nonce*, solo una cantidad moderada de repeticiones accidentales). AES-GCM pierde toda seguridad si se repite el *nonce* incluso una vez.

En el caso de CPFB el análisis de exactamente cual es la perdida de seguridad si se repite el *nonce* es más complicado, así que por el momento recomendamos fuertemente no repetirlo. De todos modos, una propiedad que tanto CPFB como

Silver tienen es que repetición del *nonce* solo aumenta el peligro para mensajes cifrados con ese *nonce*, los mensajes cifrados con otro *nonce* son completamente invulnerables(esto no pasa en AES-GCM: repetición de un *nonce* afecta la seguridad de todos los otros).

7. Conclusiones

La competencia CAESAR se encuentra aún en la primera etapa de un proceso que se extiende hasta 2017. Durante ese tiempo todos los algoritmos serán sujetos a riguroso análisis. Creemos que tanto Silver como CPFB aportan contribuciones valiosas en su campo, y esperamos que las sucesivas etapas confirmen esa creencia.

Referencias

1. Morris Dworking, NIST Special Publication 800-38A Recommendation for Block Cipher Modes of Operation, 2001 Edition
2. P. Rogaway, M. Bellare, J. Black, y T. Krovitz, "OCB: a block-cipher mode of operation for efficient authenticated encryption", ACM CCS, 2001
3. Morris Dworking, NIST Special Publication 800-38C Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, 2007.
4. David McGrew y John Viega, "The Galois/Counter Mode of Operation (GCM)", http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf
5. Jonathan Katz, Moti Yung, *"Unforgeable Encryption and Chosen Ciphertext Secure Modes of Operation"*, Fast Software Encryption Lecture Notes in Computer Science Volume 1978, 2001, pp 284-299
6. Alex Biryukov and Dmitry Khovratovich, *"Related-key Cryptanalysis of the Full AES-192 and AES-256"*, Advances in Cryptology- ASIACRYPT 2009 Lecture Notes in Computer Science Volume 5912, 2009, pp 1-18.
7. Stefan Lucks, "Two-Pass Authenticated Encryption Faster Than Generic Composition", Fast Software Encryption Lecture Notes in Computer Science Volume 3557, 2005, pp 284-298
8. Moses Liskov, Ronald L. Rivest y David Wagner, *"Tweakable Block Ciphers"*, Advance in Cryptology, CRYPTO'02, Lecture Notes in Computer Science Volume vol 2442, 2002, pp 31-46
9. Daniel Penazzi y Miguel Montes, *"Silver v1"* CAESAR website, http://competitions.cr.yp.to/round1/silverv1.pdf
10. Miguel Montes y Daniel Penazzi, *"AES-CPFB v1"* CAESAR website, http://competitions.cr.yp.to/round1/aescpfbv1.pdf