Cifrado de datos con preservación del formato

V. Gayoso Martínez[1], L. Hernández Encinas[1], A. Martín Muñoz[1]
J. M. de Fuentes[2], L. González Manzano[2]

[1]Instituto de Tecnologías Físicas y de la Información (ITEFI)
Consejo Superior de Investigaciones Científicas (CSIC), Madrid
{victor.gayoso, luis, agustin}@iec.csic.es

[2]Computer Security Lab (COSEC), Universidad Carlos III de Madrid, Leganés, Madrid
{jfuentes, lgmanzan}@inf.uc3m.es

Resumen—**En esta contribución se introducen las técnicas de cifrado con preservación del formato, las cuales tienen una importancia fundamental en el cifrado de bases de datos en las que, por motivos de compatibilidad, no se desea realizar modificaciones en el formato de los tipos de datos. De manera adicional, se presentan los detalles de implementación de uno de los algoritmos más conocidos sobre este tema, aportando un ejemplo completo de cifrado y descifrado que permitirá comprobar sus propias implementaciones al lector interesado.**

I. Introducción

El uso de las Tecnologías de la Información y de la Comunicación (TIC) se ha generalizado en el día a día de empresas y usuarios de internet. Esta situación, aunque por un lado facilita el intercambio de grandes volúmenes de información, conlleva al mismo tiempo importantes riesgos y amenazas [1].

Uno de los pilares básicos de cualquier estrategia de ciberseguridad lo constituyen las técnicas de cifrado, que tienen como objetivo principal asegurar la confidencialidad de la información, de manera que un atacante no pueda acceder a los datos originales.

Dentro de esas técnicas, se encuentran las de clave simétrica, donde los dos extremos de la comunicación utilizan la misma clave (mencionada habitualmente como clave secreta), y las de clave asimétrica, que utilizan durante su ejecución un par de claves (a las que se suele denominar clave pública y privada) [2]. En ambos casos, el resultado de cifrar cualquier tipo de información es una secuencia binaria aparentemente aleatoria y cuya longitud dependerá del algoritmo utilizado.

Los algoritmos de clave simétrica tienen como ventaja sobre los de clave asimétrica su mejor rendimiento y menor longitud de clave, debido principalmente a que las operaciones matemáticas empleadas en ese caso son más sencillas que las equivalentes utilizadas en los algoritmos de clave asimétrica. En cuanto a los principales algoritmos y funciones de clave simétrica, los más conocidos son DES y su sucesor AES [3].

La especificación de AES describe cómo cifrar bloques de 128 bits empleando claves de 128, 192 o 256 bits, obteniendo como resultado un bloque cifrado de 128 bits [4]. Aunque empleando AES con cualquiera de los modos de operación recomendados [5] es posible cifrar cualquier dato, su utilización no permite garantizar el cumplimiento de un requisito que, aunque parezca secundario, en determinados entornos es fundamental: la preservación del formato de los datos.

En el mundo empresarial y de las administraciones públicas existen circunstancias en las que, además de ser necesario cifrar los datos, estos deben mantener el mismo formato que los datos originales. Ello se debe principalmente a la necesidad de asegurar la compatibilidad con las bases de datos y aplicaciones ya existentes, y que fueron diseñadas para gestionar ciertos tipos de datos con formatos predefinidos.

Obviamente, sería posible modificar dichas bases de datos y aplicaciones para que manejaran los datos cifrados mediante, por ejemplo, el algoritmo AES, pero el coste económico y los plazos de modificación para realizar tales cambios aconsejan la búsqueda de soluciones alternativas. Y precisamente la mejor alternativa en esas situación lo constituye lo que en inglés se conoce como Format Preserving Encryption (FPE), o cifrado con preservación del formato.

Utilizando las técnicas de FPE es posible, por ejemplo, cifrar números de tarjetas de crédito de manera que el dato cifrado también sea un número de tarjeta de crédito, o cifrar identificadores de DNIe de forma que el resultado sea un DNIe distinto. Esto permite almacenar los datos cifrados en bases de datos sin modificar su formato de manera que, aunque su contenido sea hecho público, los datos verdaderos no sean conocidos.

El presente trabajo tiene como primer objetivo realizar una introducción a las técnicas de cifrado con preservación del formato, la cual está recogida en la Sección II. La Sección III describe los procesos de estandarización en este campo, mientras que la Sección IV proporciona los detalles de implementación de uno de los algoritmos propuestos para su estandarización. Por su parte, la Sección V incluye un ejemplo completo de cifrado y descifrado utilizando una implementación Java del algoritmo anteriormente mencionado. Finalmente, la Sección VI contiene nuestras conclusiones junto con la identificación de algunos potenciales problemas cuya solución, al igual que la implementación del algoritmo en dispositivos con recursos limitados, constituye un reto para los investigadores en el campo de la ciberseguridad.

II. Cifrado con preservación del formato

Los orígenes de las técnicas FPE pueden situarse en 1981, cuando se publicó la norma FIPS 74 [6] que contenía un procedimiento basado en DES que permitía que el alfabeto de los mensajes en claro y los mensajes cifrados fuera el mismo. Posteriormente, en 1997, Smith y Brightwell [7] ahondaron en la misma idea, refiriéndose a este concepto como *datatype-preserving encryption* debido a su utilización para preservar los tipos en las bases de datos.

A pesar de la novedad que representaron esos documentos, en los dos casos mencionados el tratamiento no era lo suficientemente profundo, por lo que suele considerarse que la primera contribución consistente en este campo fue la de Black y Rogaway [8], quienes se centraron en cómo cifrar un entero del rango $\{0, N-1\}$ de manera que el mensaje cifrado representara otro número del mismo rango.

Para resolver este problema, Black y Rogaway propusieron tres soluciones. El primer método, descrito por los autores como *prefix cipher*, consistía en realizar una permutación de los números enteros del rango entre 0 y $N-1$. Para ello, sugerían utilizar una función de cifrado simétrico con una clave secreta aleatoria, de manera que se procesaran todos los números del rango y se ordenaran los resultados (es decir, los mensajes cifrados) por orden alfabético.

Por ejemplo, dados los números del 0 al 9, empleando por sencillez en la explicación la función resumen SHA- 1 en lugar de una función de cifrado simétrico con una clave aleatoria, la ordenación alfabética de los resúmenes generaría la secuencia 9, 4, 1, 3, 7, 5, 0, 6, 2 y 8, tal como puede comprobarse con los resúmenes mostrados a continuación, donde el prefijo `0x` indica que la cadena situada a continuación está representada en formato hexadecimal.

```
0:0xB6589FC6AB0DC82CF12099D1C2D40AB994E8410C
1:0x356A192B7913B04C54574D18C28D46E6395428AB
2:0xDA4B9237BACCCDF19C0760CAB7AEC4A8359010B0
3:0x77DE68DAECD823BABBB58EDB1C8E14D7106E83BB
4:0x1B6453892473A467D07372D45EB05ABC2031647A
5:0xAC3478D69A3C81FA62E60F5C3696165A4E5E6AC4
6:0xC1DFD96EEA8CC2B62785275BCA38AC261256E278
7:0x902BA3CDA1883801594B6E1B452790CC53948FDA
8:0xFE5DBBCEA5CE7E2988B8C69BCFDFDE8904AABC1F
9:0x0ADE7C2CF97F75D009975F4D720D1FA6C19F4897
```

Con ello, al realizar una operación de cifrado el número 0 se transformaría en el 6, el 1 en el 2, y así sucesivamente.

El segundo método, denominado por los autores *cycle-walking cipher*, consistía en utilizar una función de cifrado simétrico cuyo dominio fuera mayor que el conjunto de elementos a cifrar, realizando tantas operaciones de cifrado como fuera necesario hasta obtener un mensaje cifrado que perteneciera al conjunto de números válidos.

Por ejemplo, dados de nuevo los números del 0 al 9, si tuviéramos una función de cifrado del tipo $(ax+b) \bmod n$ y seleccionáramos los valores $a = 3$, $b = 7$ y $n = 17$, dada la siguiente secuencia (representada en la Fig. 1), el resultado de cifrar el número 2 sería el valor 9.

$$c_1 = E_K(2) = 6 + 7 = 13 \equiv 13 \bmod 17$$

$$c_2 = E_K(13) = 39 + 7 = 46 \equiv 12 \bmod 17$$

$$c_3 = E_K(12) = 36 + 7 = 43 \equiv 9 \bmod 17$$

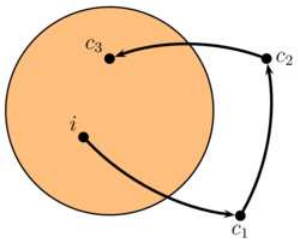

Fig. 1: Ejemplo de cifrados sucesivos (fuente: www.di-mgt.com.au).

El tercer método, denominado *generalized-Feistel cipher* por los autores, permitía una mayor flexibilidad en el proceso de cifrado mediante el empleo de un esquema Feistel. En este método, el número a cifrar (presumiblemente de muchas cifras) se descomponía en dos números de aproximadamente el mismo tamaño, procediéndose a continuación a realizar varias rondas hasta obtener un valor final. En cada ronda (ver Fig. 2) se utilizaría una clave K_i derivada de un elemento que podría ser hecho público, y que habitualmente se denomina *tweek*. En caso de que el elemento cifrado estuviera fuera del rango permitido, se podría utilizar este método junto con el anteriormente expuesto.

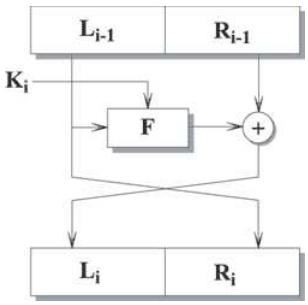

Fig. 2: Esquema de una ronda Feistel (fuente: Wikimedia Commons).

Aunque las tres propuestas estaban bien argumentadas, y su seguridad fue objeto de estudio en [8], no estaban exentas de limitaciones.

Debido a la necesidad de generar (y almacenar de forma segura) la ordenación de los elementos, el primer método sólo estaba indicado para rangos de valores pequeños. Por su parte, el segundo método presenta como inconveniente el hecho de que, si la diferencia entre el dominio de la función de cifrado y el cardinal del conjunto a cifrar es grande, cada proceso de cifrado necesitará un valor potencialmente elevado de operaciones de cifrado simétrico, por lo que su uso estaba recomendado para valores cuya longitud en bits no difiriera mucho del tamaño de bloque de

un algoritmo de cifrado simétrico. Por otra parte, la seguridad el método basado en la estructura Feistel dependía del número de rondas, y no era un método indicado para cadenas de datos de poca longitud (como por ejemplo las cadenas que representan el número de una tarjeta de crédito).

La Tabla 1 muestra un resumen de las longitudes en bits y en dígitos recomendables para cada una de las tres propuestas. Los datos de dicha tabla fueron calculados por Rerence Spies [9], quien además fue el investigador que utilizó por primera vez la expresión *format-preserving encryption* [10].

Método	Tamaño en bits	Número de dígitos
Método 1	1–20	1–6
Método 2	50–63	16–19
Método 3	40–240	12–80

Tabla 1: Comparativa de longitudes recomendadas.

III. Estandarización

Debido al interés y aplicabilidad de las técnicas de cifrado con preservación del formato, el National Institute of Standards and Technology (NIST) inició el trabajo de estandarización de estas técnicas aprobando como esquema general el conocido como FFX, propuesto por Bellare, Rogaway y Spies en 2010 [11].

Posteriormente, el 9 de junio de 2011 procedió a solicitar propuestas concretas de algoritmos de FPE que se ajustaran a dicho modelo. De entre las recibidas, el NIST incluyó tres en el primer borrador de su documento SP 800-38G [12], publicado en 2013. En concreto, las tres propuestas incluidas en dicho borrador fueron:

- FFX[Radix] (denominado FF1 internamente en el documento), desarrollado por Bellare, Rogaway y Spies [13].
- VAES3 (FF2 en el documento), propuesto por Joachim Vance [14].
- BPS (FF3 en el documento), presentado por Eric Brier, Thomas Peyrin y Jacques Stern [15].

Tras la publicación del borrador, el NIST anunció la apertura de un período de comentarios, el cual finalizó el 3 de septiembre de 2013. De los comentarios recibidos, el más importante fue un análisis de VAES3 [16] que afirma haber descubierto un ataque de texto plano escogido que demostraría que la seguridad de esa propuesta es menor de los 128 bits requeridos por el NIST. En el momento de escribir esta contribución, todavía no se ha publicado la versión definitiva del documento, desconociéndose cuándo se realizará dicha publicación.

En paralelo a las actividades del NIST, el American National Standards Institute (ANSI) inició un trabajo equivalente con la idea de generar dos estándares relacionados con el FPE: X9.119 [17], publicado en 2013, y que define los requisitos de seguridad mínimos para proteger datos sensibles de tarjetas en las transacciones comerciales, y X9.124, centrado en las técnicas FPE, y que todavía está en desarrollo, sin que exista información disponible para el público interesado.

IV. Implementación

Puesto que el estándar X9.119 no permite su libre distribución, y no hay información pública de la norma X9.124, el borrador del documento SP 800-38G es el único estándar relacionado con las técnicas FPE libremente disponible para la comunidad académica.

Con el objetivo de realizar una implementación software de alguna de las tres propuestas incluidas en ese documento, VAES3 fue descartado debido al análisis recogido en [16]. De las dos restantes, la propuesta elegida para su implementación fue FFX[Radix] debido a que, a diferencia de BPS, dispone de vectores de prueba con los que contrastar que la implementación realizada es correcta [18].

Tal como se describe en [12], dada una cadena de caracteres X de longitud n, representada en la base *radix*, una cadena de bytes T, de longitud t (el componente *tweak*) y una clave de cifrado K a utilizar con el algoritmo AES, el proceso de cifrado consiste en los siguientes pasos:

1. Calcular los valores $u = \lfloor n/2 \rfloor$ y $v = n - u$.
2. Obtener los elementos $A = X[1 \ldots u]$ y $B = X[u + 1 \ldots n]$ (es decir, A representa los primeros u caracteres de la cadena X, y B el resto).
3. Calcular los valores $b = \lceil \lceil v \log_2(radix) \rceil / 8 \rceil$ y $d = 4\lceil b/4 \rceil + 4$.
4. Generar el elemento $P = [1]^1 || [2]^1 || [1]^1 || [radix]^3 || [10]^1 || [u \bmod 256]^1 || [n]^4 || [t]^4$, donde la expresión $[x]^y$ se utiliza para representar una cadena de y bytes todos ellos con el valor numérico x, y el símbolo $||$ indica la operación de concatenación.
5. Comenzando con $i = 0$, repetir los siguientes pasos mientras $i < 10$, incrementando en cada iteración el valor de i en una unidad:
   a) Generar el elemento $Q = T || [0]^{-t-b-1 \bmod 16} || [i]^1 || [\mathrm{num}_{radix}(B)]^b$, donde $\mathrm{num}_{radix}(x)$ es una función que sirve para calcular el valor numérico asociado a la representación x en la base *radix*.
   b) Obtener el elemento $R = \mathrm{PRF}(P||Q)$, donde PRF(x) es una función que genera una salida de 128 bits a partir de una entrada cuya longitud sea múltiplo de 128, para lo cual utiliza varias iteraciones de la función de cifrado AES (para más detalles, consultar [12]).
   c) Identificar el elemento S como los primeros d bytes de la cadena $R||AES_K(R \oplus [1]^{16})||AES_K(R \oplus [2]^{16})|| \ldots ||AES_K(R \oplus [\lceil d/16 \rceil - 1]^{16})$, la cual está formada por $\lceil d/16 \rceil$ bloques, y donde el simbolo $\oplus$ representa la operación XOR.
   d) Calcular el valor $y = \mathrm{num}_2(S)$, es decir, el valor numérico de la representación binaria de S.
   e) Realizar la asignación $m = u$ si el valor de i es par, en caso contrario hacer $m = v$.
   f) Calcular el valor $c = (\mathrm{num}_{radix}(A) + y) \bmod radix^m$.
   g) Generar el elemento $C = \mathrm{STR}^m_{radix}(c)$, donde la función $\mathrm{STR}^m_{\mathrm{rad}}(x)$ permite obtener la representación del valor numérico x como una cadena de m caracteres empleando la base *radix*.
   h) Realizar las asignaciones $A = B$ y $B = C$ en el orden indicado.
6. Devolver $A||B$.

En cuanto al proceso de descifrado, que básicamente consiste en repetir los mismos pasos pero en orden inverso, dada una cadena de caracteres X de longitud n representada en la base *radix*, una cadena de bytes T de longitud t y una clave de cifrado K, está compuesto por los siguientes pasos:

1. Calcular los valores $u = \lfloor n/2 \rfloor$ y $v = n - u$.
2. Obtener los elementos $A = X[1 \ldots u]$ y $B = X[u + 1 \ldots n]$.
3. Calcular los valores $b = \lceil \lceil v \log_2(radix) \rceil / 8 \rceil$ y $d = 4\lceil b/4 \rceil + 4$.
4. Generar el elemento $P = [1]^1 || [2]^1 || [1]^1 || [radix]^3 || [10]^1 || [u \bmod 256]^1 || [n]^4 || [t]^4$.
5. Comenzando con $i = 9$, repetir los siguientes pasos mientras $i > 0$, decrementando en cada iteración el valor de i en una unidad:
   a) Generar el elemento $Q = T || [0]^{-t-b-1 \bmod 16} || [i]^1 || [\mathrm{num}_{radix}(A)]^b$.
   b) Obtener el elemento $R = \mathrm{PRF}(P||Q)$.
   c) Identificar el elemento S como los primeros d bytes de la cadena $R||AES_K(R \oplus [1]^{16})||AES_K(R \oplus [2]^{16})|| \ldots ||AES_K(R \oplus [\lceil d/16 \rceil - 1]^{16})$.
   d) Calcular el valor $y = \mathrm{num}_2(S)$.
   e) Realizar la asignación $m = u$ si el valor de i es par, en caso contrario hacer $m = v$.
   f) Calcular el valor $c = (\mathrm{num}_{radix}(B) + y) \bmod radix^m$.
   g) Generar el elemento $C = \mathrm{STR}^m_{radix}(c)$.
   h) Realizar las asignaciones $B = A$ y $A = C$, en ese orden.
6. Devolver $A||B$.

Es importante señalar que, tal como está descrita la operativa de descifrado en [12], esta contiene varios errores de transcripción (más concretamente, en los pasos 5-(a), 5-(f) y 5-(h)).

La implementación del algoritmo FFX[Radix] se ha realizado utilizando el lenguaje de programación Java Standard Edition 8. Para ello, se ha empleado la clase `BigInteger` [19], que permite operar con número enteros de cualquier longitud, superando de esta manera las limitaciones de los tipos primitivos `int` y `long` [20].

V. Ejemplo

Con el fin de demostrar la aplicabilidad de esta técnica al campo del comercio electrónico, y puesto que ninguno de los cinco vectores de test incluidos en [18] se refiere al cifrado de una cadena de 16 dígitos, que es el caso de los números de las tarjetas de crédito, en los siguientes apartados se muestra un ejemplo de cifrado y descifrado obtenido mediante nuestra implementación Java.

A. Cifrado

A continuación se incluyen los datos del ejemplo de cifrado para la cadena `1234567890123456` (con representación decimal), para lo cual se ha empleado la cadena *tweak* `9876543210` y la clave `0x2B7E151628AED2A6ABF7158809CF4F3C` a utilizar por el algoritmo AES.

Los datos iniciales, calculados antes de ejecutar las 10 rondas, son $A = 12345678$, $B = 90123456$, $n = 16$, $u = 8$, $v = 8$, $b = 4$, $d = 8$, $P =$ `0x01020100000A0A0800000010 0000000A` y $T =$ `0x39383736353433323130`. A continuación, las Tablas 2 a la 11 muestran los datos intermedios obtenidos en cada ronda, donde el elemento m toma siempre el valor 8.

Q	`0x39383736353433323130000055F2CC0`
R	`0xA63A9DBAAAB229E8CC093726AF2CFAB8`
S	`0xA63A9DBAAAB229E8`
y	11978059583998536168
c	10881846
A	90123456
B	10881846

Tabla 2: Ronda 0 del proceso de cifrado.

Q	`0x39383736353433323130000100A60B36`
R	`0xB145E2726676AC58DF76B53984E55C29`
S	`0xB145E2726676AC58`
y	12773864899079482456
c	69605912
A	10881846
B	69605912

Tabla 3: Ronda 1 del proceso de cifrado.

Q	`0x39383736353433323130000204261A18`
R	`0x44A46670E32AB291F7BBBF5CAB2C2864`
S	`0x44A46670E32AB291`
y	4946190925793243793
c	4125639
A	69605912
B	04125639

Tabla 4: Ronda 2 del proceso de cifrado.

Q	`0x39383736353433323130000003003EF3C7`
R	`0xF2987448C3DE1143C7391C098CA2408E`
S	`0xF2987448C3DE1143`
y	17480849809511158083
c	80763995
A	04125639
B	80763995

Tabla 5: Ronda 3 del proceso de cifrado.

Q	`0x39383736353433323130000404D05C5B`
R	`0xE68B9CF4E1D79BE885B6C4C2C1321FDE`
S	`0xE68B9CF4E1D79BE8`
y	16612544226061163496
c	65289135
A	80763995
B	65289135

Tabla 6: Ronda 4 del proceso de cifrado.

Q	0x39383736353433323130000503E43BAF
R	0xF6ED3ADC4E6892E097175E3AABEF3B38
S	0xF6ED3ADC4E6892E0
y	17792942420693390048
c	74154043
A	65289135
B	74154043

Tabla 7: Ronda 5 del proceso de cifrado.

Q	0x39383736353433323130000 6046B803B
R	0xA94C81462E19F661CDDB6D22EC50940B
S	0xA94C81462E19F661
y	12199267629060978273
c	26267408
A	74154043
B	26267408

Tabla 8: Ronda 6 del proceso de cifrado.

Q	0x39383736353433323130000070190CF10
R	0xFA71EAF5C58B4D3FC795BB4323188D43
S	0xFA71EAF5C58B4D3F
y	18046463523152416063
c	26570106
A	26267408
B	26570106

Tabla 9: Ronda 7 del proceso de cifrado.

Q	0x39383736353433323130000801956D7A
R	0xD3ADCC550C3C295F5A072FD2604ACD95
S	0xD3ADCC550C3C295F
y	15253072178623293791
c	49561199
A	26570106
B	49561199

Tabla 10: Ronda 8 del proceso de cifrado.

Q	0x39383736353433323130000902F43E6F
R	0x9AD091D9651C215FF52BC77234D59DE5
S	0x9AD091D9651C215F
y	11155576639886991711
c	13561817
A	49561199
B	13561817

Tabla 11: Ronda 9 del proceso de cifrado.

Una vez concluida la última ronda, se puede concluir que el resultado final es el valor obtenido al concatenar los elementos A y B o, lo que es lo mismo, el número 4956119913561817.

B. Descifrado

A continuación se incluyen los datos del ejemplo de descifrado correspondientes a la cadena `4956119913561817` (con representación decimal), utilizando la misma cadena *tweak* `9876543210` y la misma clave `0x2B7E151628AED2A6ABF7158809CF4F3C` para la función AES que en el ejemplo anterior. La inclusión de los datos asociados al proceso de descifrado será útil para los potenciales implementadores de este algoritmo, puesto que los vectores de prueba de [18] no contienen los valores intermedios de ningún ejemplo de descifrado.

Los datos iniciales, calculados antes de ejecutar las 10 rondas, son $A = 49561199$, $B = 13561817$, $n = 16$, $u = 8$, $v = 8$, $b = 4$, $d = 8$, $P =$ `0x01020100000A0A0800000 0100000000A` y $T =$ `0x39383736353433323130`. Puesto que en cada ronda de descifrado los elementos Q, R, S, y y m coinciden con los de la correspondiente ronda de cifrado, la Tabla 12 muestra únicamente los elementos c, A y B obtenidos en cada ronda (donde el lector debe recordar que, en el proceso de descifrado, la primera ronda ejecutada es la número 9, y la última la número 0).

Ronda	c	A	B
9	26570106	26570106	49561199
8	26267408	26267408	26570106
7	74154043	74154043	26267408
6	65289135	65289135	74154043
5	80763995	80763995	65289135
4	4125639	04125639	80763995
3	69605912	69605912	04125639
2	10881846	10881846	69605912
1	90123456	90123456	10881846
0	12345678	12345678	90123456

Tabla 12: Datos obtenidos en las rondas de descifrado.

Una vez concluida la última ronda de descifrado, se puede concluir que el resultado final es el número 1234567890123456, obtenido al concatenar los elementos A y B.

A título ilustrativo, el tiempo medio de ejecución de 100 procesos de cifrado ha sido 326 milisegundos, mientras que el tiempo medio de ejecución de 100 descifrados ha sido 323 milisegundos. Tal como era de esperar, debido al diseño del algoritmo los tiempos de cifrado y descifrado son prácticamente iguales. Para la realización de las pruebas, se ha utilizado un PC con sistema operativo Windows 7 Professional y procesador Intel Core i7 a 3.40 GHz.

VI. Conclusiones

En esta contribución, se ha presentado y descrito uno de los algoritmos de FPE candidatos para su estandarización por el NIST y ANSI. Este algoritmo, denominado FFX[Radix], es lo bastante flexible para poder ser usado con distintas bases (decimal, hexadecimal, alfanumérica, etc.) y con independencia de la presencia del valor *tweak*.

Tras su implementación mediante el lenguaje Java, puede observarse que su aplicación al cifrado del número de las tarjetas de crédito parece limitar su funcionamiento, puesto que por ejemplo los diversos valores y generados no difieren mucho entre sí en cada iteración del bucle, y el elemento S siempre se obtiene únicamente a partir del elemento R, lo que aconseja un estudio en mayor profundidad a fin de determinar si dichas características facilitan la aparición de vulnerabilidades y los consiguientes ataques.

Otra línea de investigación resultante del trabajo realizado consiste en el estudio de la posibilidad de implementar el mismo esquema en dispositivos con características computacionales limitadas, como por ejemplo las tarjetas inteligentes. Este tipo de dispositivos se caracteriza por su portabilidad y elevado nivel de seguridad, características requeridas en algunos escenarios reales de implantación comercial [21].

Como es sabido [22], el lenguaje Java Card tienen por definición algunas limitaciones comparado con el lenguaje Java utilizado en ordenadores personales y servidores, como por ejemplo la imposibilidad de utilizar elementos de tipo `long`, `String` o `BigInteger`.

Aunque es cierto que en su versión 2.2.2 Java Card presentó la clase `BigNumber` [23], la cual permite realizar sumas y restas con números enteros de longitud arbitraria, en la práctica existen varias limitaciones en su uso. Para empezar, el paquete al que pertenece, `javacardx.framework.math`, es opcional, por lo que no se encuentra implementado en muchas de las tarjetas disponibles comercialmente. Además, la especificación Java Card indica que, en caso de ser implementada la clase `BigNumber`, el número mínimo de bytes en los que almacenar el número entero con el que se deseen realizar los cálculos es 8 [24], por que lo el correcto funcionamiento de esta clase con números que requieran un mayor espacio de almacenamiento no está garantizado. Por último, la clase `BigNumber` no incluye ningún método para realizar reducciones modulares [24] (como las necesarias en el paso 5-f de los algoritmos de cifrado y descifrado presentados en la Sección IV), por lo que dichas operaciones tendrían que implementarse mediante sucesivas operaciones de sustracción, lo que desde el punto de vista de la eficiencia computacional constituye una opción demasiado costosa.

Por todo ello, implementar el algoritmo FFX[Radix] en tarjetas Java Card constituye un reto cuya consecución permitiría expandir las posibilidades de utilización de las técnicas FPE. Con el fin de conseguir una implementación que fuera válida en cualquier tarjeta Java Card, sería necesario desarrollar una clase equivalente a `BigInteger` a partir de los tipos numéricos disponibles en Java Card, que permitiera operar con números de longitud variable (pero con representaciones mayores de 8 bytes) y que implementara de manera eficiente operaciones de aritmética modular.

Agradecimientos

Este trabajo ha sido parcialmente subvencionado por la Comunidad de Madrid (España) bajo el proyecto S2013/ICE-3095-CM (CIBERDINE) y por el Ministerio de Economía y Competitividad (España) bajo el proyecto TIN2014-55325-C2-1-R (ProCriCiS).

Referencias

[1] Departamento de Seguridad Nacional, "Estrategia de ciberseguridad nacional", Presidencia del Gobierno, 2013.
[2] A. Fúster Sabater et al., Técnicas criptográficas de protección de datos (3ª ed.). Madrid: RA-MA, 2004.
[3] A. Fúster Sabater et al., Criptografía, protección de datos y aplicaciones. Madrid: RA-MA, 2012.
[4] National Institute of Standards and Technology, "Advanced Encryption Standard", NIST FIPS 197, 2001.
[5] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST SP 800-38A, 2001.
[6] National Institute of Standards and Technology, "Guides for Implementing and Using the NBS Data Encryption Standard", NIST FIPS 74, 1981.
[7] M. Brightwell y H. Smith, "Using Datatype-Preserving Encryption to Enchance Data Warehouse Security", en 20th NISSC Proceedings, pp. 141–149, 1997.
[8] J. Black y P. Rogaway, "Ciphers with Arbitrary Finite Domains", Lecture Notes in Computer Science, vol. 2271, pp. 114–130, 2002.
[9] T. Spies, "Format Preserving Encryption", inédito, 2011.
[10] M. Bellare et al., "Format-Preserving Encryption", Lecture Notes in Computer Science, vol. 5867, pp. 295–312, 2009.
[11] M. Bellare et al., "The FFX Mode of Operation for Format-Preserving Encryption", propuesta enviada al NIST, 2010.
[12] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption", NIST SP 800-38G, 2013.
[13] M. Bellare et al., "Addendum to The FFX Mode of Operation for Format-Preserving Encryption", propuesta enviada al NIST, 2010.
[14] J. Vance et al., "VAES3 Scheme for FFX", propuesta enviada al NIST, 2011.
[15] E. Brier et al., "BPS: a Format-Preserving Encryption Proposal", propuesta enviada al NIST, 2010.
[16] National Institute of Standards and Technology, "Analysis of VAES3 (FF2)", comentarios al borrador SP 800-38G, 2014.
[17] American National Standards Institute, "Retail Financial Services - Requirements for Protection of Sensitive Payment Card Data - Part 1: Using Encryption Methods", ANSI X9.119, 2013.
[18] Voltage Security, "AES FFX Test Vector Data version 1.0", documento enviado al NIST, 2011.
[19] Oracle Corporation, BigInteger (Java Platform SE 8), 2014. Disponible en: `http://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html`.
[20] H. Schildt, Java: The Complete Reference (9ª ed.). Nueva York: Mcgraw-Hill Osborne Media, 2014.
[21] W. Rankl y W. Effing , Smart Card Handbook (4ª ed.). West Sussex: John Wiley & Sons, 2010.
[22] V. Gayoso Martínez et al., "Java Card implementation of ECIES using prime and binary finite fields", en 4th International Conference on Computational Intelligence in Security for Information Systems (CISIS 2011), pp. 160–167, 2011.
[23] Oracle Corporation, Release Notes - Java Card Specifications - Version 2.2.2, 2006. Disponible en: `http://www.oracle.com/technetwork/java/javacard/documentation/releasenotes-jcspecspw-2-2-2-142671.html`.
[24] Oracle Corporation, Application Programming Interface - Java Card Platform - Version 2.2.2, 2006. Disponible en: `http://www.oracle.com/technetwork/java/javacard/specs-138637.html`.