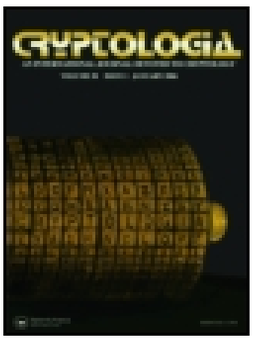

Cryptologia

ISSN: 0161-1194 (Print) 1558-1586 (Online) Journal homepage: http://www.tandfonline.com/loi/ucry20

Format-preserving encryption: Overview and NIST specification

William Stallings

To cite this article: William Stallings (2016): Format-preserving encryption: Overview and NIST specification, Cryptologia, DOI: 10.1080/01611194.2016.1169457

To link to this article: http://dx.doi.org/10.1080/01611194.2016.1169457

Published online: 13 Jun 2016.

CRYPTOLOGIA
http://dx.doi.org/10.1080/01611194.2016.1169457

Taylor & Francis
Taylor & Francis Group

Format-preserving encryption: Overview and NIST specification

William Stallings

ABSTRACT
This article reviews the concepts of and motivation for format-preserving encryption (FPE), and then describes three FPE algorithms approved by the National Institute of Standards and Technology (NIST).

KEYWORDS
block cipher; format-preserving encryption; symmetric cipher

Introduction

Format-preserving encryption (FPE) refers to any encryption technique that takes a plaintext in a given format and produces a ciphertext in the same format. For example, credit cards consist of 16 decimal digits. An FPE that can accept this type of input would produce a ciphertext output of 16 decimal digits. Note that the ciphertext need not be, and in fact is unlikely to be, a valid credit card number. It will, however, have the same format and can be stored in the same way as credit card number plaintext.

A simple encryption algorithm is not format preserving, with the exception that it preserves the format of binary strings. For example, Table 1 shows three types of plaintext for which it might be desired to perform FPE. The third row shows examples of what might be generated by an FPE algorithm. The fourth row shows (in hexadecimal) what is produced by AES with a given key.

Motivation

FPE facilitates the retrofitting of encryption technology to legacy applications, where a conventional encryption mode might not be feasible because it would disrupt data fields/pathways. FPE has emerged as a useful cryptographic tool, whose applications include financial-information security, data sanitization, and transparent encryption of fields in legacy databases.

The principal benefit of FPE is that it enables protection of particular data elements in a legacy database that did not provide encryption of those data elements, while still enabling workflows that were in place before FPE was in use. With FPE, as opposed to ordinary AES encryption or TEA encryption,

CONTACT William Stallings ws@shore.net Independent Consultant, Brewster, MA, USA.
Color versions of one or more of the figures in the article can be found online at www.tandfonline.com/ucry.

Table 1. Comparison of FPE and AES.

	Credit card	Tax ID	Bank account number
Plaintext	8123 4512 3456 6780	219-09-9999	800N2982 K-22
FPE	8123 4521 7292 6780	078-05-1120	709G9242H-35
AES (hex)	af411326466add24 c86abd8aa525db7a	7b9af4f3f218ab25 07c7376869313afa	9720ec7f793096ff d37141242e1c51bd

no database schema changes and minimal application changes are required. Only applications that must see the plaintext of a data element must be modified, and generally these modifications will be minimal.

Difficulties in designing an FPE

A general-purpose, standardized FPE should meet a number of requirements:

1. The ciphertext is of the same length and format as the plaintext.
2. It should be adaptable to work with a variety of character and number types. Examples include decimal digits, lowercase alphabetic characters, and the full character set of a standard keyboard.
3. It should work with variable plaintext lengths.
4. Security strength should be comparable to that achieved with AES.
5. Security should be strong even for very small plaintext lengths.

Meeting the first requirement is not at all straightforward. As illustrated in Table 1, a straightforward encryption with AES yields a 128-bit binary block that does not resemble the required format. Also, a standard symmetric block cipher is not easily adaptable to produce an FPE.

Consider a simple example. Assume that we want an algorithm that can encrypt decimal digit strings of maximum length 32 digits. The input to the algorithm can be stored in 16 bytes (128 bits) by encoding each digit as four bits and using the corresponding binary value for each digit (e.g., 6 is encoded as 0101). Next, we use AES to encrypt the 128-bit block, in the following fashion:

1. The plaintext input X is represented by the string of 4-bit decimal digits $X[1] \ldots X[16]$. If the plaintext is less than 16 digits long, it is padded out to the left (most significant) with zeros.
2. Treating X as a 128-bit binary string and using key K, form ciphertext $Y = \mathrm{AES}_K(X)$.
3. Treat Y as a string of length 16 of 4-bit elements.
4. Some entries in Y may have values greater than 9 (e.g., 1100). To generate ciphertext Z in the required format, calculate

$$Z[i] = Y[i] \; mod \; 10, 1 \leq i \leq 16$$

This generates a ciphertext of 16 decimal digits, which conforms to the desired format. However, this algorithm does not meet the basic requirement of any encryption algorithm of reversibility. It is impossible to decrypt Z to recover the original plaintext X because the operation is one-way; that is, it is a many-to-one function. For example, 12 mod 10 = 2 mod 10 = 2. Thus, we must design a reversible function that is both a secure encryption algorithm and format preserving.

A second difficulty in designing an FPE is that some input strings are quite short. For example, consider the 16-digit credit card number (CCN). The first six digits provide the issuer identification number (IIN), which identifies the institution that issued the card. The final digit is a check digit to catch typographical errors or other mistakes. The remaining nine digits are the user's account number. However, a number of applications require that the last four digits be in the clear (the check digit plus three account digits) for applications such as credit card receipts, which leaves only six digits for encryption. Now, suppose that an adversary is able to obtain a number of plaintext/ciphertext pairs. Each such pair corresponds to not just one CCN, but multiple CCNs that have the same middle six digits. In a large database of CCNs, there may be multiple card numbers with the same middle six digits. An adversary may be able to assemble a large dictionary mapping known six-digit plaintexts to their corresponding ciphertexts. This could be used to decrypt unknown ciphertexts from the database. As pointed out in Bellare, Rogaway, and Spies (2010a), in a database of 100 million entries, on average about 100 CCNs will share any given middle six digits. Thus, if the adversary has learned k CCNs and gains access to such a database, the adversary can decrypt approximately 100 k CCNs.

The solution to this second difficulty is to use a tweak, which functions in much the same manner as a salt used with passwords. The concept of the tweakable block cipher was first introduced in Liskov, Rivest, and Wagner (2002) An ordinary block cipher has two inputs: the encryption key, which must be kept secret, and the plaintext. The tweakable block cipher adds an additional input parameter, the tweak, which alters the algorithm so that for a given plaintext and given key, different tweaks produce different ciphertexts. The tweak itself need not be secret and must be shared with the encryption and decryption sites.

For example, the tweak for CCNs could be the first two and last four digits of the CCN. Prior to encryption, the tweak is added, digit-by-digit mod 10, to the middle six-digit plaintext, and the result is then encrypted. Two different CCNs with identical middle six digits will yield different tweaked inputs and therefore different ciphertexts. Consider the following:

CCN	Tweak	Plaintext	Plaintext + Tweak
4012 8812 3456 1884	401884	123456	524230
5105 1012 3456 6782	516782	123456	639138

Two CCNs with the same middle six digits have different tweaks and therefore different values to the middle six digits after the tweak is added.

Feistel structure for FPE

As the preceding discussion shows, the challenge with FPE is to design an algorithm for scrambling the plaintext that is secure, preserves format, and is reversible. A number of approaches have been proposed in recent years (Bellare et al. 2009; Rogaway 2010) for FPE algorithms. The majority of these proposals use a Feistel structure. Although IBM introduced this structure with their Lucifer cipher (Smith 1971) almost half a century ago, it remains a powerful basis for implementing ciphers (Stallings 2015). It was used in the Data Encryption Standard and is still in use in the NIST-approved Triple Data Encryption Algorithm (TDEA). In addition, the Camellia block cipher is a Feistel structure; it is one of the possible symmetric ciphers in TLS and a number of other Internet security protocols.

This section provides a general description of how the Feistel structure can be used to implement an FPE. In the following section, we look at three specific Feistel-based algorithms that are in the process of receiving NIST approval.

Encryption and decryption

Figure 1 shows the Feistel structure used in all the NIST algorithms, with encryption shown on the left-hand side and decryption on the right-hand side. The input to the encryption algorithm is a plaintext character string of $n = u + v$ characters. If n is even, then $u = v$; otherwise, u and v differ by 1. The two parts of the string pass through an even number of rounds of processing to produce a ciphertext block of n characters with the same format as the plaintext. Each round i has inputs A_i and B_i, derived from the preceding round (or plaintext for round 0).

All rounds have the same structure. On even-numbered rounds, a substitution is performed on the left part (length u) of the data, A_i. This is done by applying the round function F_K to the right part (length v) of the data, B_i, and then performing a modular addition of the output of F_K with A_i. The addition function is described subsequently. On odd-numbered rounds, the substitution is done on the right part of the data. F_K is a one-way function that converts the input into a binary string, performs a scrambling transformation on the string, and then converts the result back into a character string of suitable format and length. The function has as parameters the secret key K, the plaintext length n, a tweak T, and the round number i.

The process of decryption is essentially the same as the encryption process. The differences are (1) the addition function is replaced by a subtraction function that is its inverse, and (2) the order of the round indices is reversed.

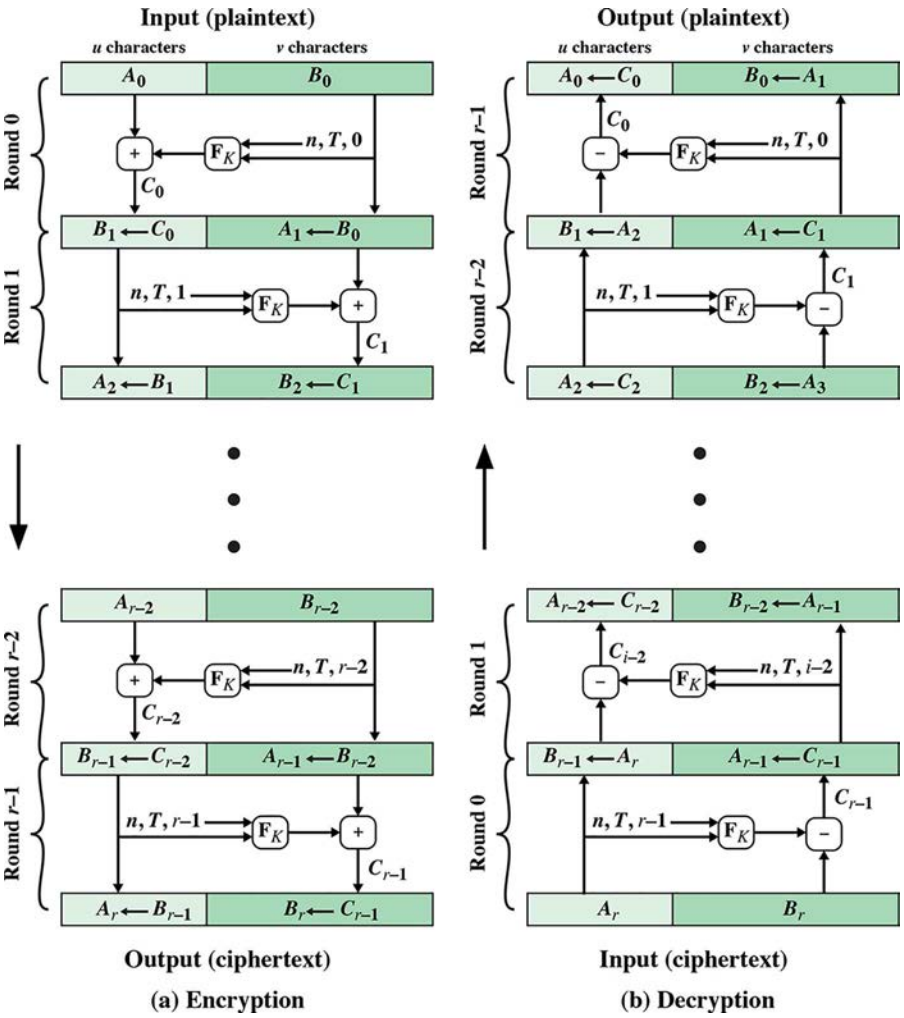

Figure 1. Feistel structure for FPE.

To demonstrate that the decryption produces the correct result, Figure 1b shows the encryption process going down the left-hand side and the decryption process going up the right-hand side. The diagram indicates that at every round, the intermediate value of the decryption process is equal to the corresponding value of the encryption process. We can walk through the figure to validate this, starting at the bottom. The ciphertext is produced at the end of round $r-1$ as a string of the form $A_r \parallel B_r$, with A_r and B_r having string lengths u and v, respectively. Encryption round $r-1$ can be described with the following equations:

$$A_r = B_r - 1$$

$$B_r = A_r - 1 + \mathrm{F}_K[B_r - 1]$$

Prerequisites:
Approved, 128-bit block cipher, CIPH;
Key, K, for the block cipher;

Input:
Nonempty bit string, X, such that LEN(X) = is a multiple of 128.
Output:
128-bit block, Y

Steps:
1. Let m = LEN(X)/128.
2. Partition X into m 128-bit blocks $X_1, \ldots, X_m$, so that $X = X_1 \| \ldots \| X_m$
3. Let $Y_0 = [0]^{16}$
4. For j from 1 to m:
5 let $Y_j = \text{CIPH}_K(Y_{j-1} \oplus X_j)$.
6. Return Y_m.

Figure 2. Algorithm PRF(X).

Because we define the subtraction function to be the inverse of the addition function, these equations can be rewritten:

$$B_r - 1 = A_r$$

$$A_r - 1 = B_r - \mathrm{F}_K[B_r - 1]$$

It can be seen that the last two equations describe the action of round 0 of the decryption function, so that the output of round 0 of decryption equals the input of round $r-1$ of encryption. This correspondence holds all the way through the r iterations, as is easily shown.

Note that the derivation does not require that F be a reversible function. To see this, take a limiting case in which F produces a constant output (e.g., all ones) regardless of the values of its input. The equations still hold.

Character strings

The NIST algorithms, and the other FPE algorithms that have been proposed, are used with plaintext consisting of a string of elements, called characters. Specifically, a finite set of two or more symbols is called an *alphabet*, and the elements of an alphabet are called characters. A *character string* is a finite sequence of characters from an alphabet. Individual characters may repeat in the string. The number of different characters in an alphabet is called the *base*, also referred to as the *radix* of the alphabet. For example, the lowercase English alphabet a, b, c, … has a radix of 26. For purposes of encryption and decryption, the plaintext alphabet must be converted to numerals, where a *numeral* is a nonnegative integer that is less than the base. For example, for the lowercase alphabet, the assignment could be characters a, b, c, …, z map into 0, 1, 2, …, 25.

A limitation of this approach is that all elements in a plaintext format must have the same radix. For example, an identification number that consists of an

alphabetic character followed by nine numeric digits cannot be handled in format-preserving fashion by the FPEs that have been implemented so far.

The NIST document defines notation for specifying these conversions (Table 2a). To begin, it is assumed that the character string is represented by a numeral string. To convert a numeral string X into a number, the function $\mathrm{NUM}_{radix}(\mathrm{X})$ is used. Viewing X as the string $\mathrm{X}[1]\ldots\mathrm{X}[m]$ with the most significant numeral first, the function is defined as

$$\mathrm{NUM}_{radix}(X) = \sum_{i=1}^{m} X[i]\ radix^{m-i} = \sum_{i=0}^{m-1} X[m-i]\ radix^{i}$$

For example, consider the string zaby in radix 26, which converts to the numeral string 25 0 1 24. This converts to the number $x = (25\times 26^3) + (1\times 26^1) + 24 = 439{,}450$. To go in the opposite direction and convert from a number $x < radix^m$ to a numeral string X of length m, the function is $\mathrm{STR}^m_{radix}(x)$ used:

$$\mathrm{STR}^m_{radix}(x) = X[1]\ldots X[m], \quad \text{where}$$
$$X[i] = \left\lfloor \frac{x}{radix^{m-1}} \right\rfloor \bmod radix, \quad i = 1\ldots m$$

With the mapping of characters to numerals and the use of the NUM function, a plaintext character string can be mapped to a number and stored as an unsigned integer. We would like to treat this unsigned integer as a bit

Table 2. Notation and parameters used in FPE algorithms.

(a) Notation	
$[x]^s$	Converts an integer into a byte string; it is the string of s bytes that encodes the number x, with $0 \le x < 2^{8s}$. The equivalent notation is $\mathrm{STR}^{8s}_2(x)$.
LEN(X)	Length of the character string X.
$\mathrm{NUM}_{radix}(X)$	Converts strings to numbers. The number that the numeral string X represents in base radix, with the most significant character first. In other words, it is the non-negative integer less than $radix^{\mathrm{LEN}(X)}$ whose most-significant-character-first representation in base radix is X.
$\mathrm{PRF}_K(X)$	A pseudorandom function that produces a 128-bit output with X as the input, using encryption key K.
$\mathrm{STR}^m_{radix}(x)$	Given a nonnegative integer x less than $radix^m$, this function produces a representation of x as a string of m characters in base radix, with the most significant character first
$[i\ldots j]$	The set of integers between two integers i and j, including i and j.
$X[i\ldots j]$	The substring of characters of a string X from $X[i]$ to $X[j]$, including $X[i]$ and $X[j]$.
REV(X)	Given a bit string, X, the string that consists of the bits of X in reverse order.
(b) Parameters	
radix	The base, or number of characters, in a given plaintext alphabet.
tweak	Input parameter to the encryption and decryption functions whose confidentiality is not protected by the mode.
tweakradix	The base for tweak strings.
minlen	Minimum message length, in characters.
maxlen	Maximum message length, in characters.
maxTlen	Maximum tweak length.

string that can be input to a bit-scrambling algorithm in F_K. However, different platforms store unsigned integers differently, some in little-endian and some in big-endian fashion. One more step is needed. By the definition of the STR function, $\mathrm{STR}_2^{8s}(X)$ will generate a bit string of length 8*s*, equivalently a byte string of length *s*, which is a binary integer with the most significant bit first, regardless of how *x* is stored as an unsigned integer. For convenience, the following notation is used: $[x]^s = \mathrm{STR}_2^{8s}(x)$. Thus, $[\mathrm{NUM}_{radix}(X)]^s$ will convert the character string *X* into an unsigned integer and then convert that to a byte string of length *s* bytes with the most significant bit first.

The function F_k

We can now define in general terms the function F_K. The core of F_K is some type of randomizing function whose input and output are bit strings. For convenience, the strings should be multiples of 8 bits, forming byte strings. Define *m* to be *u* for even rounds and *v* for odd rounds; this specifies the desired output character string length. Define *b* to be the number of bytes needed to store the number representing a character string of *m* bytes. Then, the round, including F_K, consists of the following general steps:

1. $Q \leftarrow [\mathrm{NUM}_{radix}(X)]^b$	Converts numeral string *X* into byte string *Q* of length b bytes
2. $Y \leftarrow \mathrm{RAN}[Q]$	A pseudorandom function PRNF scrambles the bits of *Q* to produce byte string *Y*
3. $y \leftarrow \mathrm{NUM}_2(Y)$	Converts *Y* into unsigned integer
4. $c \leftarrow (\mathrm{NUM}_{radix}(A) + y) \bmod radix^m$	Converts numeral string *A* into an integer and adds to *y*, modulo $radix^m$
5. $C \leftarrow \mathrm{STR}_{radix}^m(c)$	Converts *c* into a numeral string *C* of length *m*
6. $A \leftarrow B$; $B \leftarrow C$	Completes the round by placing the unchanged value of *B* from the preceding round into *A*, and placing *C* into *B*

Steps 1 through 3 constitute the round function F_K. Step 3 is presented with *Y*, which is an unstructured bit string. Because different platforms may store unsigned integers using different word lengths and endian conventions, it is necessary to perform $\mathrm{NUM}_2(Y)$ to get an unsigned integer *y*. The stored bit sequence for *y* may or may not be identical to the bit sequence for *Y*.

The pseudorandom function in step 2 need not be reversible. Its purpose is to provide a randomized, scrambled bit string. Virtually all FPE schemes that use the Feistel structure use the encryption algorithm AES as the basis for the scrambling function to achieve strong security.

Relationship between radix, message length, and bit length

Consider a numeral string *X* of length *len* and base *radix*. If we convert this to a number $x = \mathrm{NUM}_{radix}(X)$, then the maximum value of *x* is $radix^{len} - 1$. The

number of bits needed to encode x is

$$bitlen = \lceil \mathrm{LOG}_2(radix^{len}) \rceil = \lceil len \mathrm{LOG}_2(radix) \rceil$$

Observe that an increase in either *radix* or *len* increases *bitlen*. Often, we want to limit the value of *bitlen* to some fixed upper limit, for example 128 bits, which is the size of the input to AES encryption. We also want the FPE to handle a variety of radix values. The typical FPE, and all the ones discussed subsequently, allow a given range of radix values and then define a maximum character string length in order to provide the algorithm with a fixed value of *bitlen*. Let the range of radix values be from 2 to *maxradix* and the maximum allowable character string value be *maxlen*. Then, the following relationship holds:

$$maxlen \leq \lfloor bitlen / \mathrm{LOG}_2(radix) \rfloor, \text{ or equivalently}$$

$$maxlen \leq \lfloor bitlen \times \mathrm{LOG}_{radix}(2) \rfloor$$

For example, for a radix of 10, $maxlen \leq \lfloor 0.3 \times bitlen \rfloor$; for a radix of 26, $maxlen \leq \lfloor 0.21 \times bitlen \rfloor$. The larger the radix is, the smaller the maximum character length is for a given bit length.

NIST methods for FPE

In March 2016, NIST issued SP 800-38G: *Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption*. This Recommendation specifies three methods for FPE, called FF1, FF2, and FF3. The three methods all use the Feistel structure shown in Figure 1. They employ somewhat different round functions F_K, which are all built on the use of AES. Important differences are the following:

- FF1 supports the greatest range of lengths for the plaintext character string and the tweak. To achieve this, the round function uses a cipher-block-chaining (CBC) style of encryption, whereas FF1 and FF2 employ simple electronic code book (ECB) encryption.
- FF2 uses a different subkey generated from the encryption key for each round, whereas FF1 and FF2 use the encryption key directly in each round. The use of subkeys may help protect the original key from side-channel analysis, which is an attack based on information gained from the physical implementation of a cryptosystem, rather than brute force or cryptanalysis. Examples of such attacks are attempts to deduce key bits based on power consumption or execution time.
- FF3 offers the lowest round count, eight, compared to ten for FF1 and FF2, and is the least flexible in the tweaks that it supports.

Algorithm FF1

Algorithm FF1 was submitted to NIST as a proposed FPE mode (Bellare et al. 2010a, 2010b) with the name FFX[Radix]. FF1 uses a pseudorandom function $\mathrm{PRF}_K(X)$ that produces a 128-bit output with inputs X that is a multiple of 128 bits and encryption key K (Figure 2). In essence, $\mathrm{PRF}_K(X)$ uses CBC encryption (Stallings 2010b) with X as the plaintext input, encryption key K, and an initial vector (IV) of all zeros. The output is the last block of ciphertext produced. This is also equivalent to the message authentication code known as CBC-MAC, or CMAC (Stallings 2010a).

The FF1 encryption algorithm is defined in Figure 3. The shaded lines correspond to the function F_K. The algorithm has ten rounds and the following parameters (Table 2b):

- $radix \in [2 \ldots 2^{16}]$
- $\mathrm{radix}^{minlen} \geq 100$
- $minlen \geq 2$
- $maxlen < 2^{32}$. For the maximum radix value of 2^{16}, the maximum bit length to store the integer value of X is 16×2^{32} bits; for the minimum radix value of 2, the maximum bit length to store the integer value of X is 2^{32} bits.
- $maxTlen < 2^{32}$

The inputs to the encryption algorithm are a character string X of length n and a tweak T of length t. The tweak is optional in that it may be the empty string. The output is the encrypted character string Y of length n. What follows is a step-by-step description of the algorithm.

Prerequisites:
Approved, 128-bit block cipher, CIPH;
Key, *K*, for the block cipher;
Base, *radix*, for the character alphabet;
Range of supported message lengths, [minlen .. maxlen];
Maximum byte length for tweaks, maxTlen.

Inputs:
Character string, *X*, in base radix of length n such that $n \in$ [minlen .. maxlen];
Tweak *T*, a byte string of byte length t, such that $t \in$ [0 .. maxTlen].
Output:
Character string, *Y*, such that LEN(*Y*) = n.

Steps:
1. Let $u = \lfloor n/2 \rfloor$; $v = n - u$.
2. Let $A = X[1 \ldots u]$; $B = X[u+1 \ldots n]$.
3. Let $b = \lceil \lceil v \, \mathrm{LOG}_2(radix) \rceil / 8 \rceil$; $d = 4 \lceil b/4 \rceil + 4$
4. Let $P = [1]^1 \| [2]^1 \| [1]^1 \| [radix]^3 \| [10]^1 \| [u \bmod 256]^1 \| [n]^4 \| [t]^4$.
5. For i from 0 to 9:
   i. Let $Q = T \| [0]^{(-t-b-1) \bmod 16} \| [i]^1 \| [\mathrm{NUM}_{radix}(B)]^b$.
   ii. Let $R = \mathrm{PRF}_K(P \| Q)$.
   iii. Let *S* be the first *d* bytes of the following string of $\lceil d/16 \rceil$ 128-bit blocks:
   $R \| \mathrm{CIPH}_K(R \oplus [1]^{16}) \| \mathrm{CIPH}_K(R \oplus [2]^{16}) \| \ldots \| \mathrm{CIPH}_K(R \oplus [\lceil d/16 \rceil - 1]^{16})$.
   iv. Let $y = \mathrm{NUM}_2(S)$.
   v. If *i* is even, let $m = u$; else, let $m = v$.
   vi Let $c = (\mathrm{NUM}_{radix}(A) + y) \bmod radix^m$.
   vii. Let $C = \mathrm{STR}^m_{radix}(c)$.
   viii. Let $A = B$.
   ix. Let $B = C$.
6. Return $Y = A \| B$.

Figure 3. Algorithm FF1 (FF*X*[Radix]).

(1, 2) The input X is split into two substrings A and B. If n is even, A and B are of equal length. Otherwise, B is one character longer than A.

(3) The expression $\lceil v \ \mathrm{LOG}_2(\mathrm{radix}) \rceil$ equals the number of bits needed to encode B, which is v characters long. Encoding B as a byte string, b is the number of bytes in the encoding. The definition of d ensures that the output of the Feistel round function is at least 4 bytes longer than this encoding of B, which minimizes any bias in the modular reduction in Step 5.vi, as explained subsequently.

(4) P is a 128-bit (16-byte) block that is a function of *radix, u, n*, and t. It serves as the first block of plaintext input to the CBC encryption mode used in 5.ii, and is intended to increase security.

(5) The loop through the ten rounds of encryption.

(5.i) The tweak, T, the substring, B, and the round number, i, are encoded as a binary string, Q, which is one or more 128-bit blocks in length. To understand this step, first note that the value $\mathrm{NUM}_{radix}(B)$ produces a numeral string that represents B in base *radix*. How this numeral string is formatted and stored is outside the scope of the standard. Then, the value $[\mathrm{NUM}_{radix}(B)]^b$ produces the representation of the numerical value of B as a binary number in a string of b bytes. We also have the length of T is t bytes, and the round number is stored in a single byte. This yields a length of $(t + b + 1)$ bytes. This is padded out with $z = (-\mathrm{t} - \mathrm{b} - 1) \bmod 16$ bytes. From the rules of modular arithmetic, we know that $(z + t + b + 1) \bmod 16 = 0$. Thus, the length of Q is one or more 128-bit blocks.

(5.ii) The concatenation of P and Q is input to the pseudorandom function PRF to produce a 128-bit output R. This function is the pseudorandom core of the Feistel round function. It scrambles the bits of B_i (Figure 1).

(5.iii) This step either truncates or expands R to a byte string S of length d bytes. That is, if $d \leq 16$ bytes, then R is the first d bytes of R. Otherwise, the 16-byte R is concatenated with successive encryptions of R XORed with successive constants to produce the shortest string of 16-byte blocks whose length is greater than or equal to d bytes.

(5.iv) This step begins the process of converting the results of the scrambling of B_i into a form suitable for combining with A_i. In this step, the d-byte string S is converted into a numeral string in base *2* that represents S. That is, S is represented as a binary string y.

(5.v) This step determines the length m of the character string output that is required to match the length of the B portion of the round output. For even-numbered rounds, the length is u characters, and for odd-numbered rounds, it is v characters, as shown in Figure 1.

(5.vi) The numerical values of A and y are added modulo $radix^m$. This truncates the value of the sum to a value c that can be stored in m characters.

(5.vii) This step converts the c into the proper representation C as a string of m characters.

(5.viii, 5.ix) These steps complete the round by placing the unchanged value of B from the preceding round into A, and placing C into B.

(6) After the final round, the result is returned as the concatenation of A and B.

It may be worthwhile to clarify the various uses of the NUM function in FF1. NUM converts strings with a given radix into integers. In step 5.i, B is a character string in base radix, so $\text{NUM}_{radix}(B)$ converts this into an integer, which is stored as a byte string, suitable for encryption in step 5.ii. For step 5.iv, S is a byte string output of an encryption function, which can be viewed a bit string, so $\text{NUM}_2(S)$ converts this into an integer.

Finally, a brief explanation of the variable d is in order, which is best explained by example. Let $radix = 26$ and $v = 30$ characters. Then, $b = 18$ bytes, and $d = 24$ bytes. Step 5.ii produces an output R of 16 bytes. We desire a scrambled output of b bytes to match the input, and so R must be padded out. Rather than padding with a constant value such as all zeros, step 5.iii pads out with random bits. The result in step 5.iv is a number greater than $radix^m$ of fully randomized bits. The use of randomized padding avoids a potential security risk of using a fixed padding.

Algorithm FF2

Algorithm FF2 was submitted to NIST as a proposed FPE mode with the name VAES3 (Vance 2011) The encryption algorithm is defined in Figure 4. The shaded lines correspond to the function F_K. The algorithm has the following parameters:

- $radix \in [2 \ldots 2^8]$
- $tweakradix \in [2 \ldots 2^8]$
- $\text{radix}^{minlen} \geq 100$
- $minlen \geq 2$
- $maxlen \leq 2 \lfloor 120/\text{LOG}_2(radix) \rfloor$ if $radix$ is a power of 2. For the maximum radix value of 2^8, $maxlen \leq 30$; for the minimum radix value of 2, $maxlen \leq 240$. In both cases, the maximum bit length to store the integer value of X is 240 bits, or 30 bytes.
- $maxlen \leq 2 \lfloor 98/\text{LOG}_2(radix) \rfloor$ if $radix$ is a not a power of 2. For the maximum radix value of 255, $maxlen \leq 24$; for the minimum radix value of 3, $maxlen \leq 124$.
- $maxTlen \leq \lfloor 104/\text{LOG}_2(tweakradix) \rfloor$. For the maximum $tweakradix$ value of 2^8, $maxTlen \leq 13$.

For FF2, the plaintext character alphabet and that of the tweak may be different.

Prerequisites:
Approved, 128-bit block cipher, CIPH;
Key, *K*, for the block cipher;
Base, *tweakradix*, for the tweak character alphabet;
Range of supported message lengths, [*minlen* .. *maxlen*]
Maximum supported tweak length, *maxTlen*.

Inputs:
Numeral string, *X*, in base *radix*, of length *n* such that $n \in$ _[*minlen* .. *maxlen*];
Tweak numeral string, *T*, in base *tweakradix*, of length *t* such that $t \in$ [0 .. *maxTlen*].
Output:
Numeral string, *Y*, such that LEN(*Y*) = *n*.

Steps:
1. Let $u = \lfloor n/2 \rfloor$; $v = n - u$.
2. Let $A = X[1 .. u]$; $B = X[u + 1 .. n]$.
3. If $t>0$, $P = [radix]^1 \| [t]^1 \| [n]^1 \| [\text{NUM}_{tweakradix}(T)]^{13}$;
   else $P = [radix]^1 \| [0]^1 \| [n]^1 \| [0]^{13}$.
4. Let $J = \text{CIPH}_K(P)$
5. For i from 0 to 9:
   i. Let $Q \leftarrow [i]^1 \| [\text{NUM}_{radix}(B)]^{15}$
   ii. Let $Y \leftarrow \text{CIPH}_J(Q)$.
   iii. Let $y \leftarrow \text{NUM}_2(Y)$.
   iv. If *i* is even, let $m = u$; else, let $m = v$.
   v. Let $c = (\text{NUM}_{radix}(A) + y) \bmod radix^m$.
   vi. Let $C = \text{STR}^m_{radix}(c)$.
   vii. Let $A = B$.
   viii. Let $B = C$.
6. Return $Y = A \| B$.

Figure 4. Algorithm FF2 (VAES3).

The first two steps of FF2 are the same as FF1, setting values for *v, u, A*, and *B*. FF2 proceeds with the following steps:

(3) *P* is a 128-bit (16-byte) block. If there is a tweak, then *P* is a function of *radix, t, n*, and the 13-byte numerical value of the tweak. If there is no tweak, then *P* is a function of *radix* and *n*. *P* is used to form an encryption key in step 4.

(4) *J* is the encryption of *P* using the input key *K*.

(5) The loop through the ten rounds of encryption.

(5.i) *B* is converted into a 15-byte number, prepended by the round number to form a 16-byte block *Q*.

(5.ii) *Q* is encrypted using the encryption key *J* to yield *Y*.

The remaining steps are the same as for FF1. The essential difference is the way in which all the parameters are incorporated into the encryption that takes place in the block F_K. In both cases, the encryption is not simply an encryption of *B* using key *K*. For FF1, *B* is combined with the tweak, the round number, *t, n, u*, and *radix* to form a string of multiple 16-byte blocks. Then CBC encryption is used with *K* to produce a 16-byte output. For FF2, all the parameters besides *B* are combined to form a 16-byte block, which is then encrypted with *K* to form the key value *J*. *J* is then used as the key for the one-block encryption of *B*.

The structure of FF2 explains the maximum length restrictions. In step 3, *P* incorporates the radix, tweak length, the numeral string length, and the tweak into the calculation. As input to AES, P is restricted to 16 bytes. With a

maximum radix value of 2^8, the radix value can be stored in one byte (byte value 0 corresponds to 256). The string length n and tweak length t each easily fit into one byte. This leaves a restriction that the value of the tweak should be stored in at most 13 bytes, or 104 bits. The number of bits to store the tweak is $\text{LOG}_2(tweakradix^{Tlen})$. This leads to the restriction $maxTlen \leq \lfloor 104/\text{LOG}_2(tweakradix) \rfloor$. Similarly, step 5i incorporates B and the round number into a 16-byte input to AES, leaving 15 bytes to encode B, or 120 bits, so that the length must be less than or equal to $\lfloor 120/\text{LOG}_2(radix) \rfloor$. The parameter *maxlen* refers to the entire block, consisting of partitions A and B, thus $maxlen \leq 2 \lfloor 120/\text{LOG}_2(radix) \rfloor$.

Algorithm FF3

Algorithm FF3 was submitted to NIST as a proposed FPE mode with the name BPS-BC (Brier, Peyrin, and Stern 2010). The encryption algorithm is defined in Figure 5. The shaded lines correspond to the function F_K. The algorithm has the following parameters:

- $radix \in [2 \ldots 2^{16}]$
- $\text{radix}^{minlen} \geq 100$
- $minlen \geq 2$
- $maxlen \leq 2 \lfloor \text{LOG}_{radix}(2^{96}) \rfloor$. For the maximum radix value of 2^{16}, $maxlen \leq 12$; for the minimum radix value of 2, $maxlen \leq 192$. In both cases, the maximum bit length to store the integer value of X is 192 bits, or 24 bytes.
- Tweak length = 64 bits

Prerequisites:
Approved, 128-bit block cipher, CIPH;
Key, *K*, for the block cipher;
Base, *radix*, for the character alphabet such that *radix* ∈ [2 .. 2^{16}];
Range of supported message lengths, [*minlen* .. *maxlen*],
such that *minlen* ≥ 2 and *maxlen* ≤ $2\lfloor \log_{radix}(2^{96}) \rfloor$.

Inputs:
Numeral string, *X*, in base *radix* of length *n* such that *n* ∈ [*minlen* .. *maxlen*];
Tweak bit string, *T*, such that LEN(*T*) = 64.
Output:
Numeral string, *Y*, such that LEN(*Y*) = *n*.

Steps:
1. Let $u = \lceil n/2 \rceil$; $v = n - u$.
2. Let $A = X[1 .. u]$; $B = X[u + 1 .. n]$.
3. Let $T_L = T[0..31]$ and $T_R = T[32..63]$
4. For *i* from 0 to 7:
   i. If *i* is even, let $m = u$ and $W = T_R$, else let $m = v$ and $W = T_L$.
   ii. $P = W \oplus [i]^4 \,||\, [\text{NUM}_{radix}(\text{REV}(A))]^{12}$.
   iii. Let $S = \text{REVB}(\text{CIPH}_{\text{REVB}(K)}\text{REVB}(P))$.
   iv. Let $y = \text{NUM}(S)$.
   v. Let $c = (\text{NUM}_{radix}(\text{REV}(A)) + y) \bmod radix^m$.
   vi. Let $C = \text{REV}(\text{STR}^m_{radix}(c))$.
   vii. Let $A = B$.
   viii. Let $B = C$.
5. Return $A \,||\, B$.

Figure 5. Algorithm FF3 (BPS-BC).

FF3 proceeds with the following steps:

(1, 2) The input X is split into two substrings A and B. If n is even, A and B are of equal length. Otherwise, A is one character longer than B, in contrast to FF1 and FF2, where B is one character longer than A.

(3) The tweak is partitioned into a 32-bit left tweak T_L and a 32-bit right tweak T_R.

(4) The loop through the eight rounds of encryption.

(4.i) As in FF1 and FF2, this step determines the length m of the character string output that is required to match the length of the B portion of the round output. The step also determines whether T_L or T_R will be used as W in Step 4ii.

(4.ii) The bits of B are reversed, then $\text{NUM}_{radix}(B)$ produces a 12-byte numeral string in base *radix*; the results are again reversed. A 32-bit encoding of the round number i is stored in a 4-byte unit, which is reversed and then XORed with W. P is formed by concatenating these two results to form a 16-byte block.

(4.iii) P is encrypted using the encryption key K to yield Y.

(4.iv) This is similar to step 5.iv in FF1, except that Y is reversed before converting it into a numeral string in base 2.

(4.v) The numerical values of the reverse of A and y are added modulo $radix^m$. This truncates the value of the sum to a value c that can be stored in m characters.

(4.vi) This step converts c to a numeral string C.

The remaining steps are the same as for FF1.

Algorithm parameter choices

Several parameters used in the three FPE algorithms are chosen to satisfy security requirements:

- The length of the key affects the algorithm's resistance to brute-force attack. The choice of AES as the cipher dictates that the key be 16, 24, or 32 bytes in length. Most implementations are likely to use the 16-byte length as providing adequate security.
- The requirement for algorithm mode that $radix^{minlen} \geq 100$ precludes a generic meet-in-the-middle attack on the Feistel structure. The nature of such an attack is described in Patarin (2008), and Appendix H in Bellare, Rogaway, and Spies (2010) derives the specific requirement for Feistel-based FPEs.
- There is a requirement in FF2 that $maxlen \leq 2\ \lfloor \text{LOG}_{radix}(2^{96}) \rfloor$ if *radix* is not a power of 2. As explained in [VANC11], when the radix is not a power of 2, modular arithmetic causes the value (y mod $radix^m$) to not have uniform distribution in the output space, which can result in a cryptographic weakness.

Otherwise, the choices of the mode parameters (e.g., *radix, minlen*, and *maxlen*) are determined by the needs of the application, not by security considerations.

About the author

William Stallings holds a PhD from MIT in Computer Science. He is an independent consultant and author of numerous textbooks on security, computer networking, and computer architecture. He has 12 times received the award for the Best Computer Science and Engineering Textbook of the Year from the Textbook and Academic Authors Association. His most recent book is *Cryptography and Network Security, Principles and Practice, Seventh Edition* (Pearson, 2016). He is also co-author, with Lawrie Brown, of *Computer Security, Principles and Practice, Third Edition* (Pearson, 2015). He created and maintains the Computer Science Student Resource Site at Computer ScienceStudent.com. This site provides documents and links on a variety of subjects of general interest to computer science students (and professionals). Dr. Stallings is on the editorial board of *Cryptologia*.

References

Bellare, M., T. Ristenpart, P. Rogaway, and T. Stegers. 2009. Format preserving encryption. *Proceedings of SAC 2009* (*Selected Areas in Cryptography*), November 2009. Available at *Cryptology ePrint Archive* http://eprint.iacr.org/2009/

Bellare, M., P. Rogaway, and T. Spies. 2010a. Addendum to the FFX mode of operation for format-preserving encryption: A parameter collection for enciphering strings of arbitrary radix and length. NIST. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ffx/ffx-spec2.pdf (accessed September 2010).

Bellare, M., P. Rogaway, and T. Spies. 2010b. The FFX mode of operation for format-preserving encryption, draft 1.1. NIST. http://csrc.nist.gov /groups/ST/toolkit/BCM/documents/proposedmodes/ffx/ffx-spec.pdf (accessed February 2010).

Brier, E., T. Peyrin, and J. Stern. 2010. BPS: A format-preserving encryption proposal. NIST. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/bps/bps-spec.pdf (accessed April 2010).

Liskov, M., R. Rivest, and D. Wagner. 2002. Tweakable block ciphers. *Advances in Cryptology —CRYPTO 2002*.

Patarin, J. 2008. Generic attacks on Feistel schemes. ePrint report 2008/036. http://eprint.iacr.org/2008/ (accessed January 24, 2008).

Rogaway, P. 2010. A synopsis of format-preserving encryption. Unpublished Manuscript. http://web.cs.ucdavis.edu/~rogaway/papers (accessed March 2010).

Smith, J. 1971. The design of Lucifer: A cryptographic device for data communications. IBM Research Report RC 3326. (April 15, 1971).

Stallings, W. 2010a. NIST block cipher modes of operation for confidentiality. *Cryptologia* 34 (2):163–75. doi:10.1080/01611190903185401.

Stallings, W. 2010b. NIST block cipher modes of operation for authentication and combined confidentiality and authentication. *Cryptologia* 34 (3):225–35. doi:10.1080/01611191003598295.

Stallings, W. 2015. *Cryptography and network security*. Upper Saddle River, NJ: Pearson.

Vance, J. 2011. *VAES3* scheme for FFX. NIST. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ffx/ffx-ad-VAES3.pdf (accessed May 2011).