VAES3 scheme for FFX

An addendum to "The FFX Mode of Operation for Format-Preserving Encryption"

A parameter collection for encipher strings of arbitrary radix with subkey operation to lengthen life of the enciphering key

Joachim Vance[1]

Draft 1.0
May 20, 2011

1 Introduction

This document specifies the VAES3 Format Preserving Encryption (FPE) scheme. VAES conforms to the proposed FFX standard [2]. This document outlines VAES as a set of parameters for FFX.

VAES stands for variable AES. VAES3 is the third generation format-preserving encryption algorithm that was developed in a report [4] simultaneously with the comprehensive paper on FPE [1] and subsequently updated slightly to be in concert with the FFX standard proposal. The standard proposal of FFX includes, in an appendix, example instantiations called A2 and A10. A follow on addendum [3] includes an instantiation called FFX[radix] . The stated intent of FFX is that it is a framework under which many implementations are compliant. The VAES3 scheme is compliant to those requirements. VAES3 was designed to meet security goals and requirements beyond the original example instantiations, and its design goals are slightly different than those of FFX[radix]. One of the unique features of VAES3 is a subkey step that enhances security and lengthens the lifetime of the key.

VAES3 is an instance of the FE2 cipher from the FPE paper [1]. The extensive security evaluations of FE2 thus carry over to VAES3.

VAES3 addresses the problem of enciphering strings over an alphabet of an arbitrary radix from 2 to 256. Example choices are 2 (binary), 10 (decimal) and 26 (alpha), with 10, corresponding to the decimal digits used on credit card numbers, being the most important case.

[1] VeriFone Systems Inc., USA. e-mail: joachim.vance@verifone.com

2 The VAES3 Scheme

NOTATION. We assume the notation from the FFX spec [2] but, for the user's convenience, we recall the most relevant definitions here.

We let $\sum = \{0, 1, \ldots, \text{radix} - 1\}$ be an alphabet. Members of $\sum$ are referred to as characters or digits or symbols. Each character in the set has a representative numeric value. The size of $\sum$ is referred to as the radix. Example radix values are 2, 10, 26, corresponding to the binary alphabet, the alphabet of decimal digits, and the alphabet of English letters.

Plaintexts and ciphertexts are regarded as strings over the alphabet.

By $|X|$ we denote the length of string X, the number of characters in it. For example, if radix = 10, then X = 00326 is a string of length $|00326| = 5$. Note that leading zeros are counted.

The blockwise addition function $X \boxplus Y = Z$ is defined by
$a_1 \cdots a_n \boxplus b_1 \cdots b_n = c_1 \cdots c_n$ where $c_1 \cdots c_n$ is the unique string such that

$$\sum c_i \cdot \text{radix}^{n-i} = \left(\sum a_i \cdot \text{radix}^{n-i} + \sum b_i \cdot \text{radix}^{n-i}\right) \bmod \text{radix}^n.$$

By $Z \boxminus Y$ we mean the unique string X such that $X \boxplus Y = Z$. It is important to note that the "mod" function on most platforms does not calculate the modulus properly on negative numbers. Instead it calculates the remainder function, which for positive numbers is the same as the modulus. It is possible to correct this error and still use the mod function. The importance of this comes from the need to define $\boxplus$ and $\boxminus$ such that $(X \boxplus Y) \boxminus Y = X$. For example, given $X = 60$, $Y = 70$, radix = 10 and $n = 2$, then $60 \boxplus 70$ is calculated as $(60 + 70) \bmod 100 = 30$. Thus given the definition of $\boxminus$ above we know that $30 \boxminus 70$ results in $(30 - 70) \bmod 100 = 60$. This is correct for modulus function, but most computing platforms calculate $30 - 70 = -40$ and return the mod of that result as the positive integer 40. If X and Y are each less than the modulus radix^n (in the integer calculation) this can be corrected by adding radix^m to the equation to make the result of the subtraction a positive number that is congruent to the modulus of the original negative value.

Let BYTE denote $\{0, 1\}^8$, the set of 8-bit bytes.

By $\lfloor x \rfloor$ we mean the nearest integer of x rounded down.

By $\lceil x \rceil$ we mean the nearest integer of x rounded up.

By $[s]^i$ we mean the i-byte string that encodes the number $s \in [0 \,..\, 2^{8i} - 1]$. If the number is less than i bytes then the output is prepended by zeroes to take the output up to i bytes. If the number $s \geq 2^{8i}$ then the output will be the rightmost i bytes of s. This is also $s \bmod 2^{8i}$. For example, $[3]^1$ = the bit string 00000011. And $[259]^1$ is the rightmost byte of 259 and thus also equals the bit string 00000011.

By $s_1 \parallel s_2$ we mean the concatenation of string or binary data.

The function $\text{NUM}_{\text{radix}}(X)$ takes a nonempty string $X \in \{0, \ldots, \text{radix} - 1\}^*$ and converts it to the corresponding number, where the number is interpreted in the given radix, most-significant character first. The function returns the integer representation of the string, namely

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

where each value $X[i]$ is based on the value of the character in the alphabet. For example, if $\text{radix} = 10$ then $\text{NUM}_{\text{radix}}(\text{"00032"}) = 32$.

The function $\text{STR}^m_{\text{radix}}(x)$ takes a number $x \in [0 .. \text{radix}^m - 1]$ and returns the m-character string that represents it in the given radix, most significant character first. If the number of digits of the output string in radix is less than the length m, then the resulting string should be prepended with the character representing zero in that alphabet to bring the length up to m. For example $\text{STR}^5_{10}(32)$ = "00032".

The function EVEN(i) takes a number i and returns true if i is even and false if it is not and can be implemented as (i mod 2 == 0).

The function AES(K,P) is always the AES encrypt function of a single 16-byte block (ECB) for both FFX encrypt and FFX decrypt. K is a 128-bit key. P is a single 128-bit plaintext block. The AES decrypt function is never used.

We repeat the definition of the FFX encryption and decryption algorithms from the FFX proposal [2] specifying only the alternating Feistel (method = 2). This is followed by the VAES3 parameter set and definition of the VAES3 round function.

```
10 algorithm FFX.Encrypt(K, T, X)
11 if K ∉ Keys or T ∉ Tweaks or
12    X ∉ Chars* or |X| ∉ Lengths
13    then return ⊥
14 n ← |X|; ℓ ← split(n); r ← rnds(n)
15 A ← X[1 .. ℓ]; B ← X[ℓ+1 .. n]
16 for i ← 0 to r − 1 do
17    C ← A ⊞ F_K(n, T, i, B)
18    A ← B; B ← C
19 return A ‖ B
```

```
20 algorithm FFX.Decrypt(K, T, Y)
21 if K ∉ Keys or T ∉ Tweaks or
22    Y ∉ Chars* or |Y| ∉ Lengths
23    then return ⊥
24 n ← |Y|; ℓ ← split(n); r ← rnds(n)
25 A ← Y[1 .. ℓ]; B ← Y[ℓ+1 .. n]
26 for i ← r − 1 downto 0 do
27    C ← B; B ← A
28    A ← C ⊟ F_K(n, T, i, B)
29 return A ‖ B
```

Parameter	Value	Comment
radix	a number radix ∈ [2 .. 2^8]	alphabet is Chars = ∑ = {0,1, ..., radix −1}
Lengths	[2,3, … , N(radix)] where N(radix) = 2 · ⌊120/lg (radix)⌋	Permitted message lengths. lg( ) denotes the logarithm in base 2
Keys	$\{0,1\}^{128}$	128-bit AES keys
Tweaks	a string over Chars = {0,1, ..., radix −1} of length 0 to a maximum length of ⌊104 / lg(radix)⌋	Tweaks are input as strings of radix converted to a byte string. The radix of the tweak is allowed to be different than that of the plaintext and ciphertext
addition	1	Blockwise addition
method	2	Alternating Feistel
split(n)	$\lfloor n/2 \rfloor$	Maximally balanced Feistel
rnds(n)	10	Number of rounds is fixed
F	The VAES round function is given below	AES-based round function

```
30 algorithm F_K(n, T, i, B)
31 t ← |T|; i ← i+1;
32 if EVEN(i) then m ← ⌊n/2⌋ else m ← ⌈n/2⌉

33 P ← [radix]^1 ‖ [t]^1 ‖ [n]^1 ‖ [ NUM_radix( T ) ]^13
34 J ← AES(K, P)

35 Q ← [i]^1 ‖ [ NUM_radix( B ) ]^15
36 Y ← AES(J, Q)
37 y ← NUM_2( Y )

38 z ← y mod radix^m
39 return STR^m_radix( z )
```

Figure 1: Definition of VAES3

3 Implementation Notes

There are certain limits put on the lengths of the inputs. For the tweaks, length t of T must satisfy $\text{radix}^t \leq 8^{13}$ so that T may be encoded as a 104 bit binary string. Depending on the radix of T, this means the maximum length is determined by $\lfloor 104 / \lg(\text{radix}) \rfloor$. A tweak of length zero (empty string) is assumed to have the value of $[0]^{13}$.The length n of the plaintext or ciphertext must be 2 or more (single-digit plaintexts are not allowed). Although the algorithm can technically support a length n to a maximum of $2 \cdot \lfloor 120/\lg(\text{radix}) \rfloor$ there are reasons to limit this. When the radix is not a power of 2, modular arithmetic causes the output of line 38 to not have uniform distribution in the output space. The question here is, assuming AES has a random output, what is the bias, or statistical distance from random, of 128 bits of this output taken modulo radix^m? Corollary A.2 with $N = 2^{128}$ and $M = \text{radix}^m$ says the bias is at most $2^{-128} \cdot \text{radix}^m/4$. Therefore, when the radix is not a power of two, to limit the maximum statistical distance between the modular distribution and the expected random distribution to at most 2^{-32}, which is very close to uniform, the maximum input length should be limited to $2 \cdot \lfloor 98/\lg(\text{radix}) \rfloor$. For $\text{radix} = 10$ this means the plaintext/cipher text should not exceed 58 digits in length. For more information about modular distribution see appendix A and the Modular Uniformity Lemma which is repeated here from [4].

On line 33 the 128-bit string P is formed by concatenating the following strings: a one byte representation of the radix; a one byte representation of the number t of digits in the tweak; a one byte representation of the number n of characters in the plaintext; and a 104 bit (13 byte) representation of the tweak. On line 34 $\text{AES}(K, \cdot)$ is then applied to P to produce the 128-bit AES round key J. This is referred to as the key-derivation step. We believe that this enhances security and lengthens the lifetime of the key.

The key-derivation step on lines 33 and 34 need only be done once across all rounds. This optimization simplifies the number of AES calculations per encrypt or decrypt to 11, just one more than the number of rounds.

There are other optimizations that can be done to improve performance. The modulus calculation in line 38 can be removed because of the modulus calculation in the addition and subtraction function, as long as the addition and subtraction function is designed to handle the unequal length operands. Additionally the conversion between number and string during the rounds can be eliminated thus improving performance by operating on integers alone. This is a programming choice that does not change the functionality of the algorithm, meaning that the inputs and outputs produced are the same. (Note that the inputs and outputs are still strings). The conversion from number to string would occur after the last round.

On line 31 the round counter i is increased by one within the round function. Previous to the proposed FFX standard VAES used a one based round counter. But the FFX proposal uses **for** loops in the encryption and decryption algorithm that are zero-based. Increasing the round counter by one maintains compatibility with existing deployments.

Security depends very crucially on the number of rounds. The choice $\text{rnds}(n) = 10$ for all n is more than enough to preclude known attacks, and is greater than the minimum of 8 required to be compliant with the proposed standard FFX [2]. Older versions of VAES used 16 or greater rounds to induce 128 bits of entropy based on the input length as suggested in [2]. But the performance and complexity cost was too high and as addressed in [3] there seems to be no significant justification for using more rounds once the “meet-in-the-middle”

attack has been overcome at 6 rounds. Therefore a round count of 10 seems to provide both performance benefits and a margin of safety.

For the implementation of the function $\mathrm{NUM}_{\mathrm{radix}}(X)$, in software development it is well understood how data types such as decimal data, hexadecimal data, or base64 data marshal characters to numeric values in those alphabets in the range 0 to radix-1. This concept of data marshalling can be extended to support any set of input data, such as alpha-numeric data, where the radix is the number of characters in the source data set. The length of the input string and the radix of the string is the format that is being preserved by the algorithm.

All string to number and number to string conversions assume the strings represent numbers in a certain radix. Each character in the string represents a specific integer value based on the radix and the character set representing the radix. For example, decimal numbers are represented by strings of the character set $\{'0','1','2','3','4','5','6','7','8','9'\}$. Each ASCII character in the set represents a specific value in the radix. I.e., '0' = 0, '1' = 1, '2' = 2, etc. Even though the character '0' has the ASCII value 0x30 in hex, we give it an integer value of 0, which is also equivalent to its position in the array. There are 10 characters in the array and thus the radix of the numbers represented by the strings made from this character set is 10 (aka "base 10").

For another example, numbers in radix 64 can be represented by the character set {ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789 +/}. In this character set 'A' = 0, 'B' = 1, 'C' = 2 and so forth all the way up to '9' = 61, '+' = 62, '/' = 63.

The algorithm can preserve strings in arbitrary data sets as long as the character set of the alphabet is provided to the functions $\mathrm{NUM}_{\mathrm{radix}}(X)$ and $\mathrm{STR}^{m}_{\mathrm{radix}}(x)$ so that the strings can be converted to numbers and back to strings in the same character set again.

4 References

[1] **M. Bellare, T. Ristenpart, P. Rogaway, T. Stegers.** *Format Preserving Encryption.* November 3, 2009. Proceedings of SAC 2009 (Selected Areas in Cryptography), Springer LNCS Vol. 5867, 2009. IACR's ePrint http://eprint.iacr.org/2009/251 Version: 20091103:183438.

[2] **M. Bellare, P. Rogaway and T. Spies.** *Format-Preserving, Feistel-based Encryption Mode.* NIST, Proposed modes of operation, encryption mode. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ffx/ffx-spec.pdf.

[3] **M. Bellare, P. Rogaway and T. Spies.** *Addendum to "The FFX Mode of Operation for Format-Preserving Encryption".* Draft 1.0. September 3, 2010. Manuscript, available on the NIST website. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ffx/ffx-spec2.pdf.

[4] **M. Bellare.** *The FPEX Format Preserving Encryption Algorithms.* January 5, 2010, Unpublished report prepared for Semtek.

A Modular Uniformity Lemma

Consider, on the one hand, the uniform distribution on $\mathbb{Z}_M$. Consider, on the other hand, the distribution on $\mathbb{Z}_M$ that is obtained by picking a random point x in $\mathbb{Z}_N$ and returning $x \bmod M$. What is the statistical difference between these distributions? To answer this, let IntDiv denote the integer division algorithm, which on inputs N, M returns a quotient q and remainder r satisfying $N = Mq + r$ and $0 \le r < M$. Then, we claim the following.

Lemma A.1 Let $N \ge M \ge 1$ be integers, and let $(q, r) \leftarrow \text{IntDiv}(N, M)$. For $z \in \mathbb{Z}_M$ let

$$P_{N,M}(z) = Pr\left[x \bmod M = z : x \xleftarrow{\$} \mathbb{Z}_N\right].$$

Then for any $z \in \mathbb{Z}_M$,

$$P_{N,M}(z) = \begin{cases} \dfrac{q+1}{N} \; if \; 0 \le z < r \\ \dfrac{q}{N} \; if \; r \le z < M. \end{cases}$$

Proof of Lemma A.1: Let the random variable X be uniformly distributed over $\mathbb{Z}_N$. Then

$$\begin{aligned} P_{N,M}(z) &= Pr[X \bmod M = z] \\ &= Pr[X < Mq] \cdot Pr[X \bmod M = z \mid X < Mq] \\ &\quad + Pr[Mq \le X < N] \cdot Pr[X \bmod M = z \mid Mq \le X < N] \\ &= \frac{Mq}{N} \cdot \frac{1}{M} \cdot \frac{N - Mq}{N} \cdot \begin{cases} \dfrac{1}{N - Mq} & if \; 0 \le z < N - Mq \\ 0 & if \; N - Mq \le z < M \end{cases} \\ &= \frac{q}{N} \cdot \frac{r}{N} \cdot \begin{cases} \dfrac{1}{r} & if \; 0 \le z < r \\ 0 & if \; r \le z < M. \end{cases} \end{aligned}$$

Simplifying yields the claimed equation. ■

Corollary A.2 Let $N \ge M \ge 1$ be integers. Then the statistical distance between the uniform distribution on $\mathbb{Z}_M$ and the distribution obtained by picking a random point x in $\mathbb{Z}_N$ and returning $x \; mod \; M$ is at most $M/4N$. ■

Proof of Corollary A.2: Let $(q, r) \leftarrow \text{IntDiv}(N, M)$. By Lemma A.1 the statistical distance is

$$\frac{1}{2}\sum_{z=0}^{r-1}\left|\frac{q+1}{N} - \frac{1}{M}\right| + \frac{1}{2}\sum_{z=r}^{M-1}\left|\frac{q}{N} - \frac{1}{M}\right| = \frac{r(M-r)}{NM} \le \frac{1}{4}\frac{M}{N}$$

as claimed. ■