On format preserving encryption with nonce

Alexander Maximov and Jukka Ylitalo

Ericsson Research. Lund, Sweden and Jorvas, Finland
alexander.maximov,jukka.ylitalo@ericsson.com

Abstract.
In this short paper we consider a format preserving encryption when a nonce is available. The encryption itself mimics a stream cipher where the keystream is of a (non-binary) radix R. We give a few practical and efficient ways to generate such a keystream from a binary keystream generator.

Keywords: FPE · nonce

1 Introduction

A Format Preserving Encryption (FPE) is a way of encrypting plaintext message such that the resulting ciphertext keeps the same format as the plaintext. A complete FPE assumes to have a secret key as input to the algorithm, but the database containing the plaintext is not assumed to have a dedicated field for a Nonce. This way, an FPE algorithm can be seen as a keyed secret permutation over the complete (non-binary) plaintext, and this problem is, in general, challenging.

NIST (National Institute of Standards and Technology) has standardised two modes of FPE, namely, FF1 and FF3 [Dwo16]. The FF3 mode is known to be vulnerable for so-called small domain attacks [DV17]. Therefore, the focus has been on improving the performance of FF1 mode, e.g. [YWXL24], and finding new FPE algorithms that perform in higher speed rates than FF1, e.g. [DHHV21][PGSC20].

Our approach is similar to the FPE algorithm design presented by Pérez-Resa et al in [PGSC20]. They implement an FPE algorithm in a stream cipher fashion using a binary block cipher in CTR mode with a modulo operation. Pérez-Resa et al present two alternative ways for generating a key stream in their paper (Figure 2).

In this paper, we assume that a nonce is available, and discuss a few efficient alternatives for a (non-binary) keystream generation from a binary stream, while the encryption procedure is the same as in [PGSC20], where the plaintext and the keystream are added modulo the radix R. In the example given in Section VI of [PGSC20], for $R = 267$ and the secret key of 128 bits, it requires at least 149 bits from the binary generator, while in our scenario it is sufficient to have only around 72 binary bits to maintain the 128-bit security. I.e., one needs to collect around 2^{128} to be able to distinguish such a radix-R-distribution from random.

1.1 Preliminaries and notations

Let the plaintext be $p_1, p_2, \ldots, p_L$ and the ciphertext be $c_1, c_2, \ldots, c_L$, both of length L symbols where each symbol is from an alphabet of radix $R \geq 2$, i.e., $p_i, c_i \in \mathbb{Z}_R, i = 1..L$. Then we want to generate a keystream $k_1, k_2, \ldots, k_L$ from the same alphabet such that the encryption is done as in a stream cipher $c_i = (p_i + k_i) \mod R$, and the corresponding decryption is $p_i = (c_i - k_i) \mod R$.

Let us have a binary pseudo-random keystream generator KG that accepts a secret key Key and a nonce $Nonce$, and provides a pseudo-random binary stream, then we will consider ways of utilising that binary KG to produce a keystream of radix R, that is then used for encryption and decryption. For the binary keystream generator KG we could utilise e.g. the 256-bit wide block Rijndael cipher, where the nonce is combined with a counter and other parameters, and such input is then encrypted with Rijndael. The output is considered as the binary bit-stream. We define the function $KG.bits(n)$ that returns a pseudo-random and uniformly distributed integer value from the range $[0..2^n - 1]$, by utilising the next n bits of the binary keystream produced by KG.

Alternatively, a KG can be any secure binary stream cipher.

1.2 On KG and nonce

It is convenient to define a binary KG over the 256-bit wide block Rijndael cipher [DR02], as this not only allows to adopt a nonce value of a decent size, but also has bit-space for domain separation and yet another counter. Also, Rijndael can be implemented by using AES-NI SIMD instructions for the standard AES, with a fixed 32 bytes permutation between the rounds, while each round of 256-bit block Rijndael consists of two parallel calls to the AES round function. This makes Rijndael a fast and perfect candidate as a building block for a binary KG.

As an example, consider a secret key Key to be a 128/192/256-bit long, and we adopt the 256-bit wide block cipher $OUT = \texttt{Rijndael}_{Key}(IN)$. Then the 256-bit input block IN can be constructed as given in Table 1.

Table 1: A possible structure of the 256-bit input block to Rijndael for FPE.

bytes	24 bytes [0..23]	4 bytes [24..27]	4 bytes [28..31]
IN	$Nonce$	$FieldID$	$Counter$

Here, the $Nonce$ is allowed to be up to 24 bytes, which should be unique *per file*, then $FieldID$ is a fixed identifier to each field within that file to be encrypted with FPE. Each field may have a different radix and length. Then the last field $Counter$ is used by KG to generate a pseudo-random binary sequence. This way, every field can be as long as $\approx 256 \cdot 2^{32}/2$ bits, but in practical use cases these fields are much shorter. The output of Rijndael, i.e., $OUT = \texttt{Rijndael}_{Key}(IN)$, can be regarded as 256 bits of the binary stream, if more bits are needed, then KG constructs and then encrypts the same IN but with incremented $Counter$, and so on. The construction of KG can vary depending on the specific use case.

The remaining question is where to get the $Nonce$ from? As mentioned, in this paper we assume that the nonce is available, and it should be unique per file. If that file has other non-encrypted fields, such as a timestamp, user-ID, and/or other unique identifiers, then the $Nonce$ can be constructed as a Hash-based Message Authentication Code (HMAC) of the unencrypted part of the file.

However, the $Nonce$ can be stored to a database or carried in meta-data in a message together with the ciphertext. In this way, it is possible to distribute more easily the encryption with the same master key between nodes in a data center without synchronising nonces.

Finally, one should also consider the use of a message authentication code (MAC) to safeguard the file's integrity and ensure that no single bit has been altered during transmission. However, this topic falls outside the scope of this paper.

2 Keystream generation

2.1 Using $U[0,1)$

There exists many methods and libraries that generate and return a uniformly distributed pseudo-random real value in the range $[0,1)$, see e.g. [L'E97, L'E17]. A generator could be seeded with $(Key, Nonce)$ as it's initial state, so that the generated sequence would be the same for both encryption and decryption procedures. Then a keystream value k_i in the radix R can be computed as:

$$\begin{aligned} t_i &= U[0,1).\texttt{get\_next}() \\ k_i &= \lfloor R \cdot t_i \rfloor \end{aligned}$$

where $U[0,1).\texttt{get\_next}()$ is a function that returns the next pseudo-random value in the range $[0,1)$ from a generator $U[0,1)$ with a state. Mathematically, this seems an ideal approach that would generate a uniformly distributed pseudo-random sequence of integers in the range $U[0,R-1]$. However, in a real implementation the value t_i must have a certain number of precision bits to maintain a desired security level, not mentioning the details of $U[0,1)$ itself and it's cryptographic properties. We refer to Section 2.4 for more details on precision.

2.2 Monte Carlo method, non-constant time

Another standard approach is using the Monte Carlo method [Wik24], where we sample from a larger domain and drop those samples that do not fit in the smaller domain. Let us have two integer parameters $q \geq 1$ and $a \geq 1$ such that

$$q \cdot R \quad \leq \quad 2^a$$

Then, we would sample from the binary KG by requesting a bits repeatedly until we get a value x such that $x < q \cdot R$, then the resulting sample in radix R is computed as $k_i = x \mod R$. That would implement a uniformly distributed pseudo-random sequence of integers modulo R. Then the algorithm to get a new sample may look as follows:

```
int get_new_sample(int a, int q, int R) :=
while(1)
{   int x = KG.bits(a)
    if(x < q * R) return x % R;
}
```

The efficiency of the algorithm above can be measured in terms of the number of times the loop needs to be repeated, in average; and the average number of bits needed from KG to get a single sample in radix R. Obviously, the algorithm needs less cycles if we pick the smallest a:

$$a \quad = \quad \lceil \log_2(q \cdot R) \rceil$$

Then the number of loops to get a single sample in average is:

$$N^{Avr}_{time}(a,q,R) \quad = \quad \frac{2^a}{q \cdot R} \quad \text{loops per symbol.}$$

The average number of bits from KG needed to produce a single symbol is therefore

$$N^{Avr}_{bits}(a,q,R) \quad = \quad a \cdot \frac{2^a}{q \cdot R} \quad \text{bits per symbol.}$$

Note that for an optimal $a = \lceil \log_2(q \cdot R) \rceil$, we get the smallest number of bits per symbol $N^{Avr}_{bits}(a, q, R)$ when $q = 1$. Examples:

$$\begin{aligned} (a = 8, q = 25, R = 10) \quad &\Rightarrow \quad N^{Avr}_{time} = 1.024, \quad N^{Avr}_{bits} = 8.192 \\ (a = 4, q = 1, R = 10) \quad &\Rightarrow \quad N^{Avr}_{time} = 1.6, \quad N^{Avr}_{bits} = 6.4 \end{aligned}$$

In the first case KG is requested one byte each time, and almost every sample is accepted, but the number of bits per a keystream symbol needed is 8.192. In the second case the KG is requested only 4 bits each time and the loop needs to be repeated in average 1.6 times, but in the end the average number of bits needed to derive a single keystream symbol is much smaller – 6.4 bits. This may lead to a less number of invocations of Rijndael.

However, this approach is not deterministic and may have a variable time of encryption.

2.3 One keystream symbol at a time, sequentially

Let $b = \lceil \log_2 R \rceil$ and an integer $s > 0$. Assume we take a random X_1 of size $s + b$ bits from the binary KG and compute a keystream symbol as $k_1 = X_1 \mod R$. However, for a small s the distribution of such keystream symbols may have a very large bias and therefore such a keystream can be easily distinguished from random. On the other hand, for a sufficiently large s we can get a very small bias of the keystream distribution and, thus, a decent security level.

Let us derive the bias of the keystream symbols computed that way. X is uniformly distributed over the set of $\{0, 1, \ldots, 2^{s+b} - 1\}$ as it is originated from a perfect KG. We count the number of occurences for $(X \mod R = k)$ for each value of $k \in [0..R-1]$. It is clear that each occurence of k will be either $\lfloor 2^{s+b}/R \rfloor$ or $(1 + \lfloor 2^{s+b}/R \rfloor)$ times. Thus, for any value of $k \in [0..R-1]$ the difference between the probability $Pr\{X \mod R = k\}$ and the uniform probability $Pr\{U = k\} = 1/R$ is upper bounded by

$$\Delta \quad = \quad \left| \frac{0.5 \pm 0.5 + \lfloor 2^{s+b}/R \rfloor}{2^{s+b}} - \frac{1}{R} \right| \quad < \quad 2^{-(s+b)}.$$

The bias of the keystream distribution can be upper bounded in the form of the Squared Euclidean Imbalance (SEI) [BJV04] as follows:

$$\epsilon(k = X \mod R) \quad = \quad R \sum_{i=0}^{R-1} \Delta_i^2 \quad < \quad R \sum_{i=0}^{R-1} 2^{-2(s+b)} \quad = \quad 2^{2 \log_2 R - 2(s+b)}.$$

That keystream can be distinguished from random by having $O(1/\epsilon)$ number of samples. This way, if we want a t-bit security level, then we need to upper bound SEI as

$$2^{2 \log_2 R - 2(s+b)} \leq 2^{-t} \quad \Rightarrow \quad s + (b - \log_2 R) \geq t/2$$

and since $(b - \log_2 R) \geq 0$, it is sufficient to require $s \geq t/2$ to maintain the security level of t bits.

In a naïve approach, the next keystream symbol k_2 could be derived by requesting new $(s + b)$ bits from the GK. However, if we represent X_1 as $X_1 = r \cdot R + k_1$, then $k_1 = X_1 \mod R$ is the keystream symbol, and $r = \lfloor X_1/R \rfloor$ is the remaining randomness that is not used anywhere else, but we can reuse it to construct the next pseudo-random X_2 while only requesting b fresh random bits from the KG, as follows:

$$\begin{aligned} X_2 \quad &= \quad 2^b \cdot \lfloor X_1/R \rfloor + KG.bits(b) \\ k_2 \quad &= \quad X_2 \mod R \end{aligned}$$

This way, instead of requesting $(s + b) \cdot L$ bits from the GK we would only need to request $(s + b \cdot L)$ bits, which could save a lot of performance especially in case s is large and b is

small in practical instantiations. For example: let $b = 8, s = 64, L = 100$, then in the naïve approach we need 7200 pseudo-random bits that corresponds to 29 invocations of Rijndael, and in the optimised variant 864 bits are needed with only 4 calls to Rijndael.

Table 2: The results are simulated values of $\log_2(\epsilon(k_1, k_2, \ldots, k_T))$.

R	b	s	Exp.SEI	T=1	T=2	T=3	T=4	T=5	T=6	T=7	T=8	T=9	T=10
X is bounded by $s+b$ bits													
3	2	5	-10	-13.00	-11.39	-10.70	-10.26	-10.00	-9.77	-9.54	-9.32	-9.13	-8.97
3	2	10	-20	-23.00	-21.79	-21.09	-20.63	-20.33	-20.04	-19.78	-19.55	-19.34	-19.17
5	3	10	-20	-23.42	-22.35	-21.84	-21.46	-21.19	-20.94	-20.72	-20.54	-20.38	
7	3	10	-20	-22.68	-21.47	-20.63	-19.88	-19.30	-18.85	-18.49	-18.19	-17.95	
9	4	8	-16	-21.00	-19.95	-19.24	-18.78	-18.41	-18.12	-17.88			
15	4	8	-16	-20.19	-17.82	-16.29	-15.19	-14.34	-13.66	-13.13			
X is bounded by $s+b+1$ bits													
3	2	5	-10	-13.00	-12.54	-12.37	-12.02	-11.72	-11.49	-11.30	-11.16	-11.03	-10.89
3	2	10	-20	-23.00	-22.75	-22.48	-22.07	-21.67	-21.38	-21.19	-21.07	-20.95	-20.79
5	3	10	-20	-23.42	-23.02	-22.70	-22.49	-22.36	-22.22	-22.10	-22.00	-21.90	
7	3	10	-20	-22.68	-22.05	-21.91	-21.55	-21.32	-21.23	-20.90	-20.60	-20.22	
9	4	8	-16	-21.00	-20.56	-20.29	-19.74	-19.37	-19.11	-18.90			
15	4	8	-16	-20.19	-18.47	-17.54	-17.10	-16.98	-16.69	-16.59			
X is bounded by $s+b+2$ bits													
3	2	5	-10	-13.00	-12.54	-12.37	-12.25	-12.19	-12.07	-11.96	-11.87	-11.80	-11.74
3	2	10	-20	-23.00	-22.75	-22.48	-22.39	-22.32	-22.20	-22.10	-22.01	-21.94	-21.89
5	3	10	-20	-23.42	-23.02	-22.90	-22.74	-22.65	-22.58	-22.53	-22.48	-22.44	
7	3	10	-20	-22.68	-22.05	-21.91	-21.55	-21.32	-21.23	-21.16	-21.09	-21.04	
9	4	8	-16	-21.00	-20.56	-20.38	-20.26	-20.17	-20.06	-19.97			
15	4	8	-16	-20.19	-18.47	-17.54	-17.10	-16.98	-16.69	-16.59			
X is a 64-bit variable													
3	2	5	-10	-13.00	-12.54	-12.37	-12.25	-12.19	-12.16	-12.15	-12.14	-12.13	-12.13
3	2	10	-20	-23.00	-22.75	-22.48	-22.39	-22.32	-22.28	-22.27	-22.26	-22.25	-22.25
5	3	10	-20	-23.42	-23.02	-22.90	-22.86	-22.84	-22.84	-22.84	-22.84	-22.84	
7	3	10	-20	-22.68	-22.04	-21.91	-21.55	-21.31	-21.22	21.16	-21.09	-21.04	
9	4	8	-16	-21.00	-20.56	-20.38	-20.31	-20.31	-20.30	-20.30			
15	4	8	-16	-20.19	-18.47	-17.53	-17.10	-16.98	-16.69	-16.59			

Note that here $r = \lfloor X_1/R \rfloor$ is the remaining randomness from X_1, and it may actually be larger than s bits, resulting X_2 to contain more than $s+b$ bits. If X_2 has a capacity to store more than $s+b$ bits then it is only better for randomness; in another scenario X_2 can be truncated. We investigate the effect of truncation on security further.

Multidimensional SEI. We perform a set of simulations where by using the above algorithm compute the multidimensional distribution of T-tuples $(k_1, k_2, \ldots, k_T)$, given a few cases, to see how the dependency grows.

The results are given in Table 2, from where we see that the growth of SEI is not significant for a multidimensional distinguisher. However, simulations also show that if the size of X is tight $(s+b)$ bits, the scheme cannot maintain full security as SEI tends to increase larger than the expected SEI, see the red-marked valued. When the size of X is at least $\sim (s+b+1)$ bits the expected security is generally maintained, though slightly degrading with a larger T. E.g., we double checked an affordable case $R = 3, b = 2, s = 5$ and $T = 19$, i.e. considering the multidimensional distribution $D(k_1, k_2, \ldots, k_{19})$, with the bound of $(s+b+1)$ bits for X and the SEI is $2^{-9.96}$ (for $T = 11..19$, $\log_2(\text{SEI})$ is $\{-10.75, -10.62, -10.50, -10.40, -10.30, -10.21, -10.12, -10.03, -9.96\}$, respectively) that is converging to slightly above the expected 2^{-10}. This result is expected since e.g. the full entropy of $\lfloor X_1/R \rfloor$ in X_2 can be maintained by utilising one extra bit to the original $(s+b)$ bits of X_1. The more bits X has, the better security.

Exampled instantiation. If we only want to use the type `uint64_t` for X, then we can pick $b = 8$ and $s = 64 - b - 1 = 55$, which allows all $R \in [2..256]$ and gives the security close to 2^{110}. The algorithm is therefore as follows.

```
int b = \lceil log_2(R) \rceil // this can be replaced with b=8
uint64_t X = KG.bits(64 - b);
```

```
for(int i = 0; i < L; i++)
{   X = (X << b) | KG.bits(b);
    k[i] = X % R;
    X /= R;
}
```

2.4 All keystream symbols in parallel via a long division

The good point of the previous method is that it produces keystream symbols one by one, has a small state, and can process very large plaintexts. The negative point is that the security is only partially proven and partly relies on our own simulations on multidimensional distributions. Here we present yet another method that generates all keystream symbols in parallel, relies on the security proof as given in Section 2.3. The drawback of this method is that it needs to compute all keystream symbols in parallel, thus a memory storage of the full plaintext length L is needed. This method is also more time consuming (quadratic time in this method vs. linear time in the previous method).

Represent all keystream symbols as a (very) large integer K:

$$K = k_L \cdot R^{L-1} \ldots + k_3 \cdot R^2 + k_2 \cdot R^1 + k_1 \cdot R^0 \in [0, R^L - 1].$$

Then we want to get K by taking $K = X \mod R^L$ where X has a sufficient number of binary bits from GK, in order to fullfill the required security level, following the security proof as given in Section 2.3. Thus, if we want the security level of t bits then the total number of needed pseudo-random bits for X is

$$w = \lceil \underbrace{(t/2 - \log_2 R)}_{\text{security level } t} + \underbrace{(L \cdot \log_2 R)}_{\text{for } K} \rceil = \lceil t/2 + (L-1)\log_2 R \rceil$$

Then the algorithm of generating the complete keystream is as follows:

$$\begin{aligned} k_1 &= \lfloor X/R^0 \rfloor \mod R \\ k_2 &= \lfloor X/R^1 \rfloor \mod R \\ k_3 &= \lfloor X/R^2 \rfloor \mod R \\ &\vdots \end{aligned} \tag{1}$$

Let X be represented in u chunks of q-bit integers, such that $u \cdot q \geq w$, i.e.

$$X = X_{u-1} \cdot 2^{(u-1)\cdot q} + \ldots + X_2 \cdot 2^{2\cdot q} + X_1 \cdot 2^{1\cdot q} + X_0 \cdot 2^{0\cdot q}$$

Then the computation of $k_1 = (X \mod R)$ and $Y = \lfloor X/R \rfloor$, where Y is also represented as a vector of u q-bit integers similarly to X above, can be implemented *iteratively* as follows:

$$\begin{aligned} &k_1 = 0 \\ &\textbf{for } i = u-1 \textbf{ downto } 0 \\ &\quad Y_i = k_1 \cdot 2^q + X_i \\ &\quad k_1 = Y_i \mod R \\ &\quad Y_i = \lfloor Y_i/R \rfloor \end{aligned}$$

Note that the next keystream word k_2 is computed as $k_2 = (Y \mod R)$ in a similar way. Thus, we can merge the computation of the remainders in a combined loop, while

dropping the results of the divisors Y as not needed for further computation:

$k_1, k_2, \ldots, k_L = 0$
for $i = u - 1$ **downto** 0
  $x = X_i$ « x is a q-bit integer
  for $j = 1$ **to** L
    $y = k_j \cdot 2^q + x$ « y is at most $(q + b)$-bit integer
    $k_j = y \mod R$
    $x = \lfloor y/R \rfloor$

Verification of the above algorithm can be done easily by constructing X from those X_i-values added as q fresh least significant bits, and then computing k_i as in the algorithm above, and compare the resulting keystream against Equation 1. For example, let $L = 10$, $R = 19$, and we represent X as a polynomial over the powers of R, having $u = 12$ coefficients each $q = 5$ bits long, and assume KG produces 12 5-bit random values $X_i =$ `{9,3,30,4,1,12,22,14,18,16,9,17}`, i.e. $X =$ `0x08a6127598127869`. Both methods give the same result of $L = 10$ keystream symbols modulo $R = 19$: $k_i =$ `{0,1,18,4,6,14,6,11,6,4}`.

Using 64-bit wide integers, an implementation of the above idea could be as follows:

```
uint8_t k[1..L] = {0,...,0}
w = \lceil t/2 + (L-1)\log_2 R \rceil
b = \lceil \log_2(R) \rceil
cap = 64 - b;
for(int bits = 0; bits < w; bits += cap)
{   uint64_t x = KG.bits(cap);
    for(int j = 1; j <= L; j++)
    {   uint64_t y = x + ((uint64_t)k[j] << cap);
        k[j] = y % R;
        x = y / R;
    }
}
```

Note that there is no connection between the security level t and the size of the available type (`uint64_t`) for the intermediate integer x, thus a standard type like `uint64_t` can be used for that long division. The advantage here is that each call to KG requests only a manageable amount of bits, so that for the purpose of a long division there is no need to know the complete X in advance. Also note that in the above exampled implementation X may have more bits than needed minimum w bits (in case $(w \mod cap) \neq 0$), but this only increases the security level while keeping the implementation simpler.

Possible modifications. If *cap* is not a multiple of 8 bits, it might be more difficult to handle bit-level random values from KG. In this case, we can propose two possible modifications, as follows.

1. Reduce *cap* to the nearest multiple of 8, so that only full bytes are requested from KG, i.e. `cap -= cap % 8`.

2. In order to make the implementation simpler, we can request a fixed number of bits $KG.bits(64)$, instead of $KG.bits(cap)$, and either skip the upper b bits, or keep them as an additional noise that does not have an effect on the resulting bias, but only "rotates" the remainders when added as `x + (k[j] « cap)`. In the above exampled implementation, the number of 64-bit blocks needed is $\lceil w/cap \rceil$.

For example, $R = 256 - 1$, $L = 100$, $t = 128$, then we derive that $w = 856$ random bits are needed; with $b = 8$ and capacity of $64 - b = 56$ bits, we then need 16 64-bit random values from KG, which corresponds to 4 invocations of Rijndael. On the other hand, if

$R = 128 + 1,\ L = 100$, we then compute $b = 8,\ w = 759,\ cap = 56$ – this needs 14 64-bit random values. If we do not apply the optimised call to KG and each time request exactly cap bits, then we need only 12 64-bit random values for the latter case, that corresponds to only 3 invocations of Rijndael. However, managing non-64-bit random variables might be costly and it could indeed be simpler to use an optimised version with the replaced line `uint64_t x = KG.bits(64).`

2.5 Aggregated mode

All the previously discussed techniques can be speeded up by utilising a so-called "*aggregated mode*" of encryption, where the plaintext is split into n chunks each of which is then encrypted independently. The KG for each chunk needs to be initialised with the triple $(Key, Nonce, Index)$, where $Index \in [0..n-1]$ is a hard-coded index of a corresponding encryption stream.

3 Encryption/Decryption with SIMD

Encryption. Once the keystream in radix R is produced, the encryption can be done in parallel over blocks of plaintext symbols. For example, assume $R \leq 256$ thus a plaintext symbol p_i can fit into a single byte (i.e., `uint8_t`), and the same for k_i. Now using e.g. a 64-byte ZMM register, available in AVX-512, we want to compute 64 expressions $c_i = (p_i + k_i) \mod R$ in parallel, by utilising SIMD instructions, see e.g. [Int24].

Assume the following 64-byte registers are (pre-)loaded:

```
__m512i KS = _mm512_loadu_si512(keystream);
__m512i PT = _mm512_loadu_si512(plaintext);
__m512i RAD= _mm512_set1_epi8(Radix);
```

In case both $p_i, k_i \in \mathbb{Z}_R$, and $R \leq 127$ we get the 64-byte vector of ciphertext symbols `CT` in the radix R as follows:

```
__m512i CT = _mm512_add_epi8(PT, KS);
__mmask64 m = _mm512_cmpge_epu8_mask(CT, RAD); // when c>=R
CT = _mm512_mask_sub_epi8(CT, m, CT, RAD);
```

A similar result can be achieved with e.g. AVX (applied to 32-byte vectors):

```
  Q = _mm256_andnot_si256(_mm256_cmpeq_epi8(CT,
                          _mm256_max_epu8(CT, PT)), RAD);
  CT = _mm256_sub_epi8(CT, Q);
```

In case $R \in [128..255]$ we may have a byte-overflow and thus a modified procedure should apply (but this procedure also works for $R \leq 127$):

```
__m512i CT = _mm512_add_epi8(PT, KS);
__mmask64 m = _kor_mask64(_mm512_cmplt_epu8_mask(CT, PT),   // when c>=2^8
                          _mm512_cmpge_epu8_mask(CT, RAD)); // when c>=R
CT = _mm512_mask_sub_epi8(CT, m, CT, RAD);
```

In case $p_i \in \mathbb{Z}_R$, but $k_i \in \mathbb{Z}_{q \cdot R}$, where $R < 256,\ q \cdot R \leq 256$, we may need to use a function for taking the actual remainder, i.e.:

```
__m512i CT = _mm512_add_epi8(PT, KS);
__mmask64 m = _mm512_cmplt_epu8_mask(CT, PT); // when c>=2^8
CT = _mm512_mask_sub_epi8(CT, m, CT, RAD);
CT = _mm512_rem_epu8(CT, RAD);    // c%=R, SVML
```

However note that the instruction `_mm512_rem_epu8()` is a sequence of other SIMD instructions, and the function is implemented by an SVML library – not all compilers support that.

Decryption. The decryption procedure may be done through the encryption procedure, preceded by the negation of the keystream symbols, since $Dec(\{k_i\}, CT) = Enc(\{q \cdot R -$

$k_i\}, CT$). Thus, we simply modify the keystream symbols as below, and push the ciphertext into the encryption engine.

```
KS_dec = _mm512_sub_epi8(_mm512_set1_epi8(q*R), KS_enc)
```

References

[BJV04] Thomas Baignères, Pascal Junod, and Serge Vaudenay. How Far Can We Go Beyond Linear Cryptanalysis? In Pil Joong Lee, editor, *ASIACRYPT 2004*, volume 3329 of *LNCS*, pages 432–450, December 2004.

[DHHV21] F Betül Durak, Henning Horst, Michael Horst, and Serge Vaudenay. FAST: secure and high performance format-preserving encryption and tokenization. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 465–489. Springer, 2021.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael*, volume 2. Springer, 2002.

[DV17] F Betül Durak and Serge Vaudenay. Breaking the FF3 format-preserving encryption standard over small domains. In *Annual international cryptology conference*, pages 679–707. Springer, 2017.

[Dwo16] M Dworkin. Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption. *NIST Special Publication 800-38G*, 2016.

[Int24] Intel. Intel Intrinsics Guide, 2024. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[L'E97] Pierre L'Ecuyer. Uniform random numbers generators: a review. In S. Andradóttir and K. J. Healy and, editors, *Proceedings of the 29th conference on Winter Simulation Conference*, pages 127–134, New York, NY, USA, 1997. ACM Press.

[L'E17] Pierre L'Ecuyer. History of uniform random number generation. In *2017 Winter Simulation Conference (WSC)*, pages 202–230, 2017.

[PGSC20] Adrián Pérez-Resa, Miguel Garcia-Bosque, Carlos Sánchez-Azqueta, and Santiago Celma. A New Method for Format Preserving Encryption in High-Data Rate Communications. *IEEE Access*, 8:21003–21016, 2020.

[Wik24] Wikipedia. Monte Carlo method — Wikipedia, The Free Encyclopedia, 2024. Accessed: 2024-10-29.

[YWXL24] Xian-Wei Yang, Lan Wang, Ma-Li Xing, and Qiang Li. Improved Execution Efficiency of FPE Scheme Algorithm Based on Structural Optimization. *Electronics*, 13(20), 2024.