FASER v1

Authenticated Encryption in a Feedback Shift Register

Designers: Faith Chaza, Cameron McDonald and Roberto Avanzi

Submitter: Cameron McDonald
cameronm@qti.qualcomm.com

March 14, 2014

1 Specification

1.1 Parameters

FASER is a family of authenticated ciphers. The family consists of two parent ciphers: FASER128 and FASER256. The nomenclature represents the maximum key length that can be used in each cipher. Both parent ciphers have four parameters: key length, secret message number length, public message number length, and tag length. Each parameter is an integer number of bytes.

FASER128 Parameters The key length is either 10 bytes (80 bits) or 16 bytes (128 bits). The secret message number length is either 0 bytes, 8 bytes (64 bits), 12 bytes (96 bits), or 14 bytes (112 bits). The public message number length is either 0 bytes, 8 bytes (64 bits), 12 bytes (96 bits), or 14 bytes (112 bits). The tag length is either 8 bytes (64 bits), 12 bytes (96 bits), or 16 bytes (128 bits).

FASER256 Parameters The key length is either 24 bytes (192 bits) or 32 bytes (256 bits). The secret message number length is either 0 bytes, 8 bytes (64 bits), 12 bytes (96 bits), or 14 bytes (112 bits). The public message number length is either 0 bytes, 8 bytes (64 bits), 12 bytes (96 bits), or 14 bytes (112 bits). The tag length is either 8 bytes (64 bits), 12 bytes (96 bits), 16 bytes (128 bits), or 20 bytes (160 bits).

1.2 Recommended parameter sets

The recommended parameter sets for FASER include:

- FASER128: 16 byte key, 8 byte secret message number, 8 byte public message number, and 8 byte tag.
- FASER128: 16 byte key, 12 byte secret message number, 12 byte public message number, and 12 byte tag.
- FASER256: 32 byte key, 8 byte secret message number, 8 byte public message number, and 8 byte tag.
- FASER256: 32 byte key, 12 byte secret message number, 12 byte public message number, and 12 byte tag.

1.3 Authenticated Encryption

The inputs to authenticated encryption are a plaintext P, associated data A, a key K, a secret message number SMN, and a public message number PMN. There are no limits imposed on the length of the plaintext or associated data.

The output of authenticated encryption is a ciphertext C which includes the tag T as its suffix (ie. output is $C \parallel T$). The length of the ciphertext is equal to the length of the plaintext plus the tag length.

FASER consists of three main tasks: *initialisation* - to initialise state based on the key and public message number, *update* - to process the input data, and *finalisation* - to compute the authentication tag. In FASER, the SMN is not used for any special purpose, it is concatenated to the plaintext P and encrypted using the same method. Hence, the input to be encrypted becomes $P \parallel SMN$.

1.4 State

FASER uses two state registers, identical in size, one for encryption and one for authentication, denoted E and A respectively. The register size for FASER128 is is 256, while the register size for FASER256 is 384. We use the notation X_i to denote the i-th 64-bit word of register X. Thus, in FASER128, the state registers will be represented as:

$$E = (E_3, E_2, E_1, E_0)$$
$$A = (A_3, A_2, A_1, A_0).$$

Similarly, in FASER256, the state registers will be represented as:

$$E = (E_5, E_4, E_3, E_2, E_1, E_0)$$
$$A = (A_5, A_4, A_3, A_2, A_1, A_0).$$

The state registers used in FASER are updated using a FSR (Feedback Shift Register). The encryption function uses the FSR for a stream cipher, while the authentication function uses the FSR for an accumulator. The FSR update is identical in the encryption function and the authentication function, it is just the initialisation where they differ.

1.5 FSR

The FSR is made up of n sub-FSRs, where n is 8 and 12 for FASER128 and FASER256 respectively. The sub-FSRs update different regions of the state independently. The sub-FSRs have been chosen so that: pairwise they are 64-bits in size; and all sub-FSR lengths are coprime. There are 2 sub-FSRs that operate on each 64-bit word. We use the notation $H_i(X)$ and $L_i(X)$ to represent the i-th most significant (High) bits and i-th least significant (Low) bits respectively.

In FASER128, the FSR is:

$$FSR(X) = (FSR_3(X_3), FSR_2(X_2), FSR_1(X_1), FSR_0(X_0)),$$

where,

$$\begin{aligned}
FSR_0(X_0) &= FSR33(H_{33}(X_0)) \parallel FSR31(L_{31}(X_0)) \\
FSR_1(X_1) &= FSR35(H_{35}(X_1)) \parallel FSR29(L_{29}(X_1)) \\
FSR_2(X_2) &= FSR41(H_{41}(X_2)) \parallel FSR23(L_{23}(X_2)) \\
FSR_3(X_3) &= FSR47(H_{47}(X_3)) \parallel FSR17(L_{17}(X_3)).
\end{aligned}$$

In FASER256, the FSR is:

$$FSR(X) = (FSR_5(X_5), FSR_4(X_4), FSR_3(X_3), FSR_2(X_2), FSR_1(X_1), FSR_0(X_0)),$$

where,

$$\begin{aligned}
FSR_0(X_0) &= FSR33(H_{33}(X_0)) \parallel FSR31(L_{31}(X_0)) \\
FSR_1(X_1) &= FSR35(H_{35}(X_1)) \parallel FSR29(L_{29}(X_1)) \\
FSR_2(X_2) &= FSR37(H_{37}(X_2)) \parallel FSR27(L_{27}(X_2)) \\
FSR_3(X_3) &= FSR41(H_{41}(X_3)) \parallel FSR23(L_{23}(X_3)) \\
FSR_4(X_4) &= FSR43(H_{43}(X_4)) \parallel FSR21(L_{21}(X_4)) \\
FSR_5(X_5) &= FSR47(H_{47}(X_5)) \parallel FSR17(L_{17}(X_5)).
\end{aligned}$$

Here, $\parallel$ denotes concatenation. Note that the sub-FSRs used in FASER128 are a subset of the sub-FSRs used in FASER256. For each 64-bit pair of sub-FSRs, there is one nonlinear FSR and one linear FSR. We use the notation x_i to denote the i-th bit of variable X. The definition for all sub-FSRs is given below.

$$
\begin{aligned}
&FSR17(X) : y \leftarrow x_{16} + x_{15} + x_{14} \cdot x_{13}, && (x_{16}, \ldots, x_1, x_0) \leftarrow (x_{15}, \ldots, x_0, y)\\
&FSR21(X) : y \leftarrow x_{20} + x_{19} + x_{17} \cdot x_{15}, && (x_{20}, \ldots, x_1, x_0) \leftarrow (x_{19}, \ldots, x_0, y)\\
&FSR23(X) : y \leftarrow x_{22} + x_{21} + x_{12} \cdot x_{11}, && (x_{22}, \ldots, x_1, x_0) \leftarrow (x_{21}, \ldots, x_0, y)\\
&FSR27(X) : y \leftarrow x_{26} + x_{24} + x_{21} \cdot x_{16} + 1, && (x_{26}, \ldots, x_1, x_0) \leftarrow (x_{25}, \ldots, x_0, y)\\
&FSR29(X) : y \leftarrow x_{28} + x_{27} + x_{19} \cdot x_{12}, && (x_{28}, \ldots, x_1, x_0) \leftarrow (x_{27}, \ldots, x_0, y)\\
&FSR31(X) : y \leftarrow x_{30} + x_{11} + x_{21} \cdot x_{13}, && (x_{30}, \ldots, x_1, x_0) \leftarrow (x_{29}, \ldots, x_0, y)\\
&FSR33(X) : y \leftarrow x_{32} + x_{19}, && (x_{32}, \ldots, x_1, x_0) \leftarrow (x_{31}, \ldots, x_0, y)\\
&FSR35(X) : y \leftarrow x_{34} + x_{32}, && (x_{34}, \ldots, x_1, x_0) \leftarrow (x_{33}, \ldots, x_0, y)\\
&FSR37(X) : y \leftarrow x_{36} + x_{35} + x_{32} + x_{30}, && (x_{36}, \ldots, x_1, x_0) \leftarrow (x_{35}, \ldots, x_0, y)\\
&FSR41(X) : y \leftarrow x_{40} + x_{37}, && (x_{40}, \ldots, x_1, x_0) \leftarrow (x_{39}, \ldots, x_0, y)\\
&FSR43(X) : y \leftarrow x_{42} + x_{41} + x_{37} + x_{36}, && (x_{42}, \ldots, x_1, x_0) \leftarrow (x_{41}, \ldots, x_0, y)\\
&FSR47(X) : y \leftarrow x_{46} + x_{41}, && (x_{46}, \ldots, x_1, x_0) \leftarrow (x_{45}, \ldots, x_0, y).
\end{aligned}
$$

Note that + and $\cdot$ denote addition and multiplication over GF(2) (XOR and AND), respectively. The non-linear sub-FSRs have algebraic degree 2.

All sub-FSRs have been chosen so that, in software, they may easily be clocked at least 8 bits at a time. In one update of FASER, the FSR is clocked 8 times, where each clock of the FSR results in each sub-FSR being clocked. Consequently, there are 64-bits and 96-bits updated in each state register of FASER128 and FASER256 respectively.

1.6 MIX

The MIX function combines information from across the state register. This function is used in the NLF (described below). It is also used to diffuse information directly in the authentication state register.

In FASER128, the MIX function input is the entire state (X_3, X_2, X_1, X_0) and the output is three 64-bit words such that:

$$
\begin{aligned}
Y_0 &= (X_0 \lll \ \ 3) \oplus (X_1 \lll 12) \oplus (X_2 \lll 43) \oplus (X_3 \lll 27)\\
Y_1 &= (X_0 \lll 22) \oplus (X_1 \lll 54) \oplus (X_2 \lll \ \ 5) \oplus (X_3 \lll 30)\\
Y_2 &= (X_0 \lll 50) \oplus (X_1 \lll 35) \oplus (X_2 \lll 14) \oplus (X_3 \lll 60).
\end{aligned}
$$

In FASER256, the MIX function input is the entire state $(X_5, X_4, X_3, X_2, X_1, X_0)$ and the output is three 64-bit words such that:

$$Y_0 = (X_0 \lll 3) \oplus (X_1 \lll 11) \oplus (X_4 \lll 26) \oplus (X_5 \lll 36)$$
$$Y_1 = (X_1 \lll 49) \oplus (X_2 \lll 16) \oplus (X_3 \lll 24) \oplus (X_4 \lll 59)$$
$$Y_2 = (X_0 \lll 30) \oplus (X_2 \lll 41) \oplus (X_3 \lll 51) \oplus (X_5 \lll 2).$$

Here, $\lll$ denotes bitwise rotation to the left. The rotations chosen for the MIX are described in the design rationale section.

1.7 NLF

The MAJ function is the bitwise majority function. In FASER, the MAJ function operates on 64-bit words. The output of MAJ is Z:

$$Z = (Y_0 \wedge Y_1) \vee (Y_0 \wedge Y_2) \vee (Y_1 \wedge Y_2)$$

The NLF (nonlinear filter) is the composition of the MIX function and the MAJ function applied to the entire state register. That is, $NLF(X) = MAJ(MIX(X))$. The NLF is used in the encryption function to produce the stream. It is also used in the authentication finalisation to produce the tag. An interesting property of the bitwise majority function is that the exclusive or $\oplus$ can be used in place of the inclusive or $\vee$, and the resulting function is the same. This function has degree 2.

1.8 Update

Each clock of FASER invokes two operations: the encryption function for producing ciphertext, and the authentication function for accumulating the ciphertext. The encryption function of FASER is a synchronous stream cipher that outputs a 64-bit word for each cipher clock. The stream is bitwise xor'ed with a plaintext word to produce a ciphertext word. If the plaintext is less than a full 64-bit word, the most significant n bytes of stream are xor'ed with the plaintext to produce the ciphertext. Thus, the encryption function is length preserving. The authentication function of FASER takes one input, the 64-bit ciphertext word output from the encryption function.

The FASER update function is designed so that the bulk of the encryption function and the authentication function (FSR and MIX) can be done in parallel. The following describes one update of FASER to process one 64-bit plaintext word P_i. FASER continues to clock until all the input has been processed.

The pseudo-code for the $UPDATE$ process is:

$$
\begin{aligned}
&UPDATE(E, A, P_i): \\
&\quad E = FSR(E), && A = FSR(A) \\
&\quad (Y_2, Y_1, Y_0) = MIX(E), && (M_2, M_1, M_0) = MIX(A) \\
&\quad C_i = P_i \oplus MAJ(Y_2, Y_1, Y_0), && A = (A_3 \oplus C_i, A_2 \oplus M_2, A_1 \oplus M_1, A_0 \oplus M_0) \\
& && A = (A_2, A_1, A_0, A_3)
\end{aligned}
$$

The high level design of the update process is displayed in Figure 1.

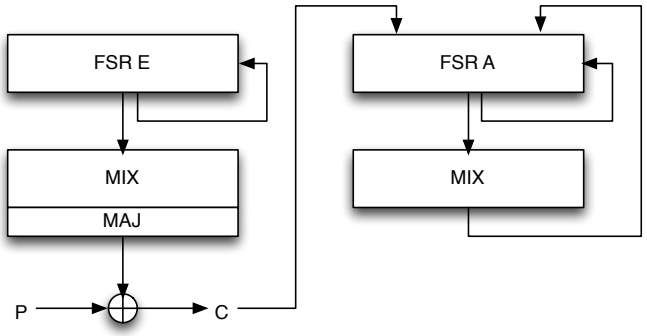

Fig. 1. Faser Update Process

While processing associated data, the encryption function is not required and does not need to be clocked. The plaintext data is directly input to the authentication function (ie. $C_i = P_i$).

Similarly to the encryption of partial words, if the input to the authentication function is not a full 64-bit word, the input bytes fill the most significant bytes of a 64-bit word and the remaining bytes are padded with zero. For example, if the plaintext is 3 bytes long such that $P = \{0x12, 0x34, 0x56\}$, the input word to be processed is $P_i = 0x1234560000000000$ (where the most significant bytes are written on the left).

1.9 Initialisation

This process intitialises the two state registers using the key K and the public message number PMN. Initially, the inputs are simply copied directly into the register, least significant byte first. The remaining bytes are filled with a constant, $0x5A$ and $0xA5$ for the encryption and authentication register respectively. The different constants ensure the registers E and A will begin in different starting states. The register contents are then diffused so that the inputs (K and PMN) affect the entire state. This is achieved using the FSR and MIX function on each of the registers. To avoid fixed points in the FSR, the register contents are tweaked for each 64-bit word using the following process:

$$TWEAK(E_i) : (x_{63}, x_{62}, \ldots, x_2, x_1, x_0) \leftarrow (1, x_{62}, \ldots, x_2, 0, 1)$$
$$TWEAK(A_i) : (x_{63}, x_{62}, \ldots, x_2, x_1, x_0) \leftarrow (1, x_{62}, \ldots, x_2, 1, 0).$$

This ensures each linear sub-FSR is not all 0's and each nonlinear FSR is not all 0's and not all 1's. The FSR fixed points are discussed in the design rationale section. Additionally, the registers E and A have subtly different tweak constants to ensure the register start states remain different. Further diffusion is applied to hide the tweaked constants. This is achieved using the FSR, where each sub-FSR is updated independently and we are guaranteed to not end up with fixed points.

The pseudo-code for the $INIT$ process is:

$INIT(E, A, K, PMN)$:

$E = 0x5A \ldots 0x5A \parallel PMN \parallel K$	$A = 0xA5 \ldots 0xA5 \parallel PMN \parallel K$
for $i = 1$ to 8 **do**	
$E = FSR(E)$,	$A = FSR(A)$
$(Y_2, Y_1, Y_0) = MIX(E)$,	$(M_2, M_1, M_0) = MIX(A)$
$E = (E_3, E_2 \oplus Y_2, E_1 \oplus Y_1, E_0 \oplus Y_0)$,	$A = (A_3, A_2 \oplus M_2, A_1 \oplus M_1, A_0 \oplus M_0)$
$E = (E_2, E_1, E_0, E_3)$,	$A = (A_2, A_1, A_0, A_3)$
end for	
$TWEAK(E)$,	$TWEAK(A)$
do $i = 1$ to 8 **do**	
$E = FSR(E)$,	$A = FSR(A)$
end for	

1.10 Finalisation

This process generates the authentication tag based on contents of the authentication register. The authentication register is updated 16 more times without any ciphertext input. The authentication register is then updated and outputs 64-bit words in a similar fashion to the $UPDATE$ function. This continues to happen until the required length of tag has been output. If the required tag length is not a multiple of 8 bytes (full 64-bit words), then only the most significant bytes of T_i are used.

The pseudo-code for the $FINAL$ process is:

$FINAL(A)$:
  for $i = 1$ to 16 **do**
    $A = FSR(A)$
    $(M_2, M_1, M_0) = MIX(A)$
    $A = (A_3, A_2 \oplus M_2, A_1 \oplus M_1, A_0 \oplus M_0)$
    $A = (A_2, A_1, A_0, A_3)$
  end for
  do $i = 1$ to n **do**
    $A = FSR(A)$
    $(M_2, M_1, M_0) = MIX(A)$
    $T_i = MAJ(M_2, M_1, M_0)$
  end for

2 Security Goals

Goal	FASER128v1 bits of security	FASER256v1 bits of security
Confidentiality for the plaintext	128	256
Confidentiality for the secret message number	128	256
Integrity for the plaintext	64	64
Integrity for the associated data	64	64
Integrity for the secret message number	64	64
Integrity for the public message number	64	64

Table 1. Security goals for a cipher with an 8 byte secret message number, an 8 byte public message number and an 8 byte tag.

Goal	FASER128v1 bits of security	FASER256v1 bits of security
Confidentiality for the plaintext	128	256
Confidentiality for the secret message number	128	256
Integrity for the plaintext	96	96
Integrity for the associated data	96	96
Integrity for the secret message number	96	96
Integrity for the public message number	96	96

Table 2. Security goals for a cipher with a 12 byte secret message number, a 12 byte public message number and a 12 byte tag.

The numbers in Table 1 and in Table 2 are on different scales: the confidentiality measure is the expected number of key guesses to find the secret key and the integrity measure is the expected number of online forgery attempts for a successful forgery. Any successful forgery or successful key guess should be assumed to completely compromise confidentiality and integrity of all messages.

The public message number is a nonce and should only be used once under each key. The secret message number is not a nonce and can be used multiple times under the same key, provided different public message numbers are used each time.

The cipher does not promise any integrity or confidentiality if the legitimate key holder uses the same (public message number, secret message number) pairs to encrypt two different (plaintext,associated data) pairs under the same key.

3 Security Analysis

3.1 Avalanche Criterion

The initialisation process satisfies the strict avalanche condition, which is to say that if two initial states (key and PMN) differ by a single bit, then the difference between the corresponding keystream sequences will appear random in nature. This was ensured by first checking that every bit of the state at completed initialisation is a non-linear function of every bit of the key and of the PMN, and then by probabilistic tests. Furthermore, changing either the public message number or the key by a single bit lead to seemingly random changes in the authentication tag.

3.2 Correlations

Divide and conquer attacks are of particular concern with our design. This is primarily due to the fact that the FSR is constructed from several smaller FSRs. However, we have chosen the NLF such that the output is not correlated to any one input word (or any one FSR).

The key and the keystream do not exhibit significant correlation. Nor do the public message number and the keystream. There also appears to be no correlation between keystream sequences generated with similar public message numbers. After the initialisation, the relationship between the original key and the internal state of the FSR was investigated and it was found that there was no significant correlation between the two. Similarly, there was no significant correlation present between the public message number and the internal state. The key and the authentication tag do not exhibit significant correlation. Nor do the public message number and the tag, or the secret message number and the tag.

3.3 Weak or Equivalent Keys

Weak keys could only arise, by the design, if any of the FSR entered into a steady state. Therefore we have devised a mechanism to avoid this situation, i.e. the use of the TWEAK functions described in Subsection 1.9. However, the TWEAK basically discards 3 bits of information for each 64 bit block of the internal state, replacing them with fixed bits, different for the E and A FSRs. This may give rise to equivalent keys. However we observe that the key and the PMN provide, for FASER128, up to 240 bits of information, and by the way these are mixed by the 8 rounds of FSR and MIX functions, the discarded bits are non linear combinations of bits of both. Hence, we are aware that, with extremely low probability, key/PMR combinations K1/P1 and K2/P2 may occur that give the same encrypted stream. However, by the fact that different TWEAKs are applied to the E and A FSRs, the authentication tag would be computed with a different state. (The computation of the exact likelihoods for these occurrences is a planned task, however we have not been able to construct equivalent key/PMR combinations. Also, an attacker that were able to manipulate the system to feed an equivalent key/PMR pair would need access to a flexible encryption oracle and thus to decrypt the stream anyway).

3.4 Algebraic Attacks

For FASER128, the encryption state contains 256 unknown variables. Each update of the FSR produces 64 new variables and 64 equations. Of the 64 equations, 32 are linear and 32 are quadratic. The stream output from the NLF produces another 64 quadratic equations. So, after n rounds, the number of variables is $v = 256 + 64 \cdot n$ and the number of equations is $e = 128 \cdot n$. We can assume that, to solve the system and recover the state, one would require $e \geq v$. That is, a minimum of 4 output words (256 bits) must be observed. This system has 512 state variables, 128 linear equations and 384 quadratic equations. It has more variables and at least 2^{13} quadratic terms, thus vastly larger than the system arising from Trivium, that is estimated to be solvable in time 2^{164} [2]. We think this system has sufficient complexity to not be solvable, in a reasonable amount of time, using any known algebraic techniques. For FASER256, after 6 updates, the system of equations has 768 variables, 192 linear equations and 576 quadratic equations. We do not consider systems of overdefined equations.

3.5 Side channel attacks

Timing attacks and power attacks can be mitigated in standard ways; there is no data-dependent conditional execution after initial keying, no table lookups, nor are the operations used data-dependent in execution time on most CPUs.

4 Features

The main features of our design are given below:

1. security - the FSR design ensures a very large period, the MIX design ensures good diffusion.
2. unique - the design does not rely on the security of another crypto primitive.
3. fast - the bulk of the FASER update can be parallelized in software, taking advantage of modern processors SIMD instructions.
4. compact - in hardware, the FSR and MIX circuitry can be reused for encryption and authentication.
5. simple - easy design to implement and analyse.

We expect an optimized version of the software will exceed the speed of current AES-GCM implementations.

5 Design Rationale

FASER is designed to be fast in software and hardware, allowing the majority of the computation to be done in parallel. It is also possible to make a compact implementation where common circuitry can be used for both the encryption and authentication functions. The operations used in FASER are all basic lightweight operations, including: XOR, OR, AND, NAND and ROTATE. These are generally native operations on most processors.

The designer/designers have not hidden any weaknesses in this cipher.

5.1 FSR

The large nonlinear FSR is made up of several smaller sub-FSRs. The main reason for this design is that it is difficult to construct large single nonlinear FSRs where the period is known. The size of the sub-FSRs have been chosen for two reasons: pairwise they are 64-bits in size; and all sizes are coprime.

The property that they are pairwise 64-bits in size has a memory benefit, a pair fits in one word (on 64-bit processors). Also, it allows a for a small reduction in update operations, where it is possible to perform the shift operation, on both FSRs in the pair, at the same time.

The property that the sizes are all coprime, potentially allows one to construct a large FSR that has a very large period. It is well known that linear FSRs of size n have a maximum period of $2^n - 1$ when the feedback polynomial is primitive [1]. It follows, that if $(n_0, n_1, \ldots, n_l)$ are coprime, then $(2^{n_0} - 1, 2^{n_1} - 1, \ldots, 2^{n_l} - 1)$ are also coprime and the period is $(2^{n_0} - 1) \cdots (2^{n_l} - 1)$.

However, the construction and properties of nonlinear FSRs is not as well understood. Finding nonlinear FSRs that have maximum period and are easily computable is a non-trivial task. We searched for nonlinear FSRs that have the following properties:

1. Maximum shortest cycle - of all the possible FSRs. Ideally, the period is $2^n - 1$.
2. Lightweight - small number of terms in its feedback polynomial. Ideally, only one quadratic term and two linear terms.
3. Ability to clock several times. Ideally, it should be easy to compute a minimum of 8 new bits in one update.

Unfortunately, the FSRs that have maximal period are not lightweight or easy to compute. We have sacrificed period length for compactness. Also, most of the nonlinear FSRs found have two fixed points, the all zero state and the all one state. We avoid these states by tweaking values during the initialisation.

The FSRs chosen are listed in the specification section. The cycles for each nonlinear FSR are given below:

FSR17: 2088, 2970, 8108, 31991, 39628, 46285
FSR21: 10793, 20273, 25261, 37303, 85100, 1918420
FSR23: 37637, 48986, 64669, 2353785, 2363712, 3519817
FSR27: 32927, 56106, 140044, 161956, 353113, 441724, 921481, 2208102, 11754280, 118147995
FSR29: 387669, 489905, 1103550, 6255276, 7016117, 25530555, 35662528, 89314799, 185300784, 185809727
FSR31: 214199, 1707514, 2199359, 2521919, 8033832, 25562983, 49854433, 62432904, 402128512, 1592827991

Since each FSR is clocked 8 times per update, the update cycle lengths for each nonlinear FSR are:

FSR17: 261, 1485, 2027, 31991, 9907, 46285
FSR21: 10793, 20273, 25261, 37303, 21275, 479605
FSR23: 37637, 24493, 64669, 2353785, 295464, 3519817
FSR27: 32927, 28053, 35011, 40489, 353113, 110431, 921481, 1104051, 1469285, 118147995
FSR29: 387669, 489905, 551775, 1563819, 7016117, 25530555, 4457816, 89314799, 23162598, 185809727
FSR31: 214199, 853757, 2199359, 2521919, 1004229, 25562983, 49854433, 7804113, 50266064, 1592827991

The linear FSRs chosen all have maximal period. The minimum FSR period is then the lowest common multiple of all possible cycle lengths. For FASER128, the minimum period is:

$$P_{128} = 261 \cdot 2353785 \cdot 551775 \cdot 214199 \cdot (2^{33}-1) \cdot (2^{35}-1) \cdot (2^{41}-1) \cdot (2^{47}-1) \approx 2^{201}.$$

For FASER256, the minimum period is:

$$P_{256} = 261 \cdot 21275 \cdot 2353785 \cdot 35011 \cdot 25530555 \cdot 214199 \cdot (2^{33}-1) \cdots (2^{47}-1) \approx 2^{292}.$$

5.2 MIX

While the FSR introduces some diffusion within each sub-FSR state, the MIX function is designed to diffuse across all 64-bit register words. The MIX function is a linear code compressing the state down to three 64-bit words. The rotations have been chosen such that:

1. Each output bit is affected by at least one input bit from a nonlinear FSR.
2. The new bits resulting from the FSR update are dispersed across the output words.
3. The rotates between common words form a full positive difference set.

5.3 Update

The bulk of the encryption and authentication functions are the FSR update and the MIX function. These operations are independent for encryption and authentication and may be computed at the same time (in parallel). It is also possible to reuse the same hardware to accomplish these tasks.

6 Intellectual Property

US Provisional 61/952,629 has been filed.

If any of this information changes, the submitter/submitters will promptly (and within at most one month) announce these changes on the crypto-competitions mailing list.

7 Consent

The submitter/submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitter/submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitter/submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitter/submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitter/submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitter/submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

References

1. Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone, and R. L. Rivest. Handbook of applied cryptography, 1997.
2. Håvard Raddum. Cryptanalytic results on trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/039 (2006), http://www.ecrypt.eu.org/stream/papersdir/2006/039.ps.