Name of Submission:

FlexAEAD - A Lightweight Cipher with Integrated Authentication

Name of Submitters:

Eduardo Marsola do Nascimento[1], José Antônio Moreira Xexéo[2]

(1) edunasci@yahoo.com, +55(21)990357336, Petrobras Petróleo Brasileiro S.A.
Av. Dr. Altino Arantes, 895 – apt 102 – Vila Clementino
São Paulo – SP – 04042-034 – Brazil

(2) xexeo@ime.eb.br, +55(21)25467092, Instituto Militar de Engenharia
Praça General Tibúrcio, 80 – Praia Vermelha – Urca
Rio de Janeiro – RJ – 22290-270 – Brazil

Abstract. This paper describes a symmetrical block cipher family – FlexAEAD. It was engineered to be lightweight, consuming less computational resources than other ciphers and to work with different block and key sizes. Other important characteristic is to integrate the authentication on its basic algorithm. This approach is helps to reduce the resource needs. The algorithm capacity to resist against linear and different cryptanalysis attacks was evaluated. This algorithm is a variation of the FlexAE algorithm presented at IEEE ICC2017 (Paris – France) and SBSEG2018 (Natal – Brazil). The FlexAEAD also supports the authentication of the Associated Data (AD).

1. Algorithm Description

The FleaxAEAD algorithm uses as a main component a key dependable permutation function(PF_K). On this function, the block is XORed with a key K_A at the beginning and with a key K_B at the end of the process. This function (PF_K) is invertible ($INVPF_K$), so the process can be reversed.

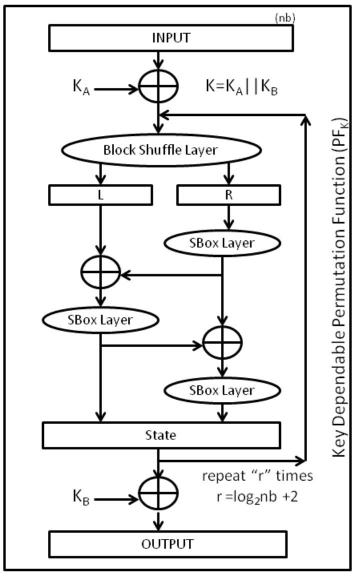

Figure 1. The permutation function PF_K diagram

On the (PF_K), after the XOR with K_A, the block is transformed by shuffle layer, where a 2^{nb} bytes input is divided in 4 $bits$ blocks ($b[0], b[1], \ldots, b[2^{nb+1}-1]$) and reordered as $(\mathrm{b}[0], \mathrm{b}[2^{\mathrm{nb}}], \mathrm{b}[1], \mathrm{b}[2^{\mathrm{nb}}+1], \ldots, \mathrm{b}[2^{\mathrm{nb}}-1],\ \mathrm{b}[2^{\mathrm{nb}+1}-1]$.

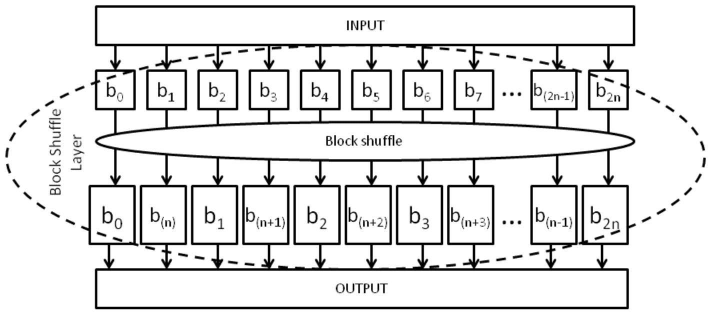

Figure 2. The BlockShuffle Layer

After the shuffle, the block is divided into two parts (L, R). The right part (R) suffers a non-linear transformation using a SBox Layer where each byte is translated by the AES

SBox table generating (R'). The left part (L) and (R') are XORed resulting in (L'). The (L') is applied to a SBox Layer generating (L''). The (L'') and (R') are XORed together generating (R'') which is applied to the SBox Layer to generate(R'''). The pair (L'', R''') are combined together $(state)$. Although this construction resembles a Feistel network, it needs the SBox Layer to be reversible. The main reason for this construction is to improve the resistance to cryptanalysis attacks by forcing the combination of two input bytes to be applied to an active SBox.

The SBox Layer can be inverted using the reverse AES SBox. On the appendices the AES SBox direct and reverse tables can be found.

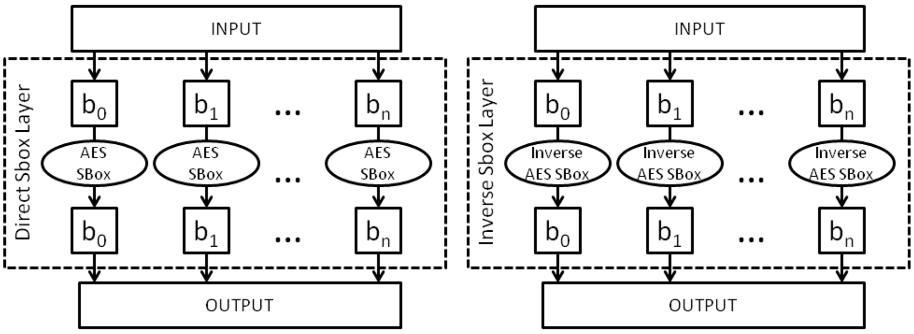

Figure 3. The SBox Layer

The number of rounds (r) on this construction is $r = \log_2 nb + 2$, where nb=block size in bytes. This number of rounds is the minimum to assure that any bit change on the input the block will affect all bits on the output. The number of rounds grows logarithmic with the block size, keeping the number of cpu cycles needed to process small even if working with bigger block sizes.

The key dependable permutation function and its inverse can also be described on the pseudo code on the Figure 4.

$dirPF(INPUT[nb], K[nk])$	$invPF(INPUT[nb], K[nk])$
$K_A = K\left[1..(\frac{nk}{2})\right]$	$K_A = K\left[1..(\frac{nk}{2})\right]$
$K_B = K\left[\left(\frac{nk}{2}+1\right)..nk\right]$	$K_B = K\left[\left(\frac{nk}{2}+1\right)..nk\right]$
$state = INPUT \oplus K_A$	$state = INPUT \oplus K_B$
`for ( i =1 to log(nb)+2]`	`for ( i =1 to log(nb)+2]`
`state = ShuffleLayer(state)`	`state = ShuffleLayer(state)`
$L = state\left[1..\left(\frac{nb}{2}\right)\right]$	$L = state\left[1..\left(\frac{nb}{2}\right)\right]$
$R = state\left[\left(\frac{nb}{2}\right)+1..nb\right]$	$R = state\left[\left(\frac{nb}{2}\right)+1..nb\right]$
$R' = SBoxLayer(R)$	$R' = invSBoxLayer(R)$
$L' = L \oplus R'$	$R'' = L \oplus R'$
$L'' = SBoxLayer(L')$	$L' = invSBoxLayer(L)$
$R'' = L'' \oplus R'$	$L'' = L' \oplus R''$
$R''' = SBoxLayer(R'')$	$R''' = invSBoxLayer(R'')$
$state\left[1..\left(\frac{nb}{2}\right)\right] = L''$	$state\left[1..\left(\frac{nb}{2}\right)\right] = L''$
$state\left[\left(\frac{nb}{2}\right)+1..nb\right] = R'''$	$state\left[\left(\frac{nb}{2}\right)+1..nb\right] = R'''$
`end for`	`end for`
$OUTPUT = state \oplus K_B$	$OUTPUT = state \oplus K_B$

Figure 4. The key dependable permutation function and its inverse

The FlexAEAD cipher uses four subkeys (K_1, K_2, K_3, K_4). They are created from a bit sequence generated by applying the permutation function three times using the main key

K (PF_K) until have enough bits for all subkeys. The initial value is a sequence of zeros $(0^{ks/2})$. Each subkey (K_1, K_2, K_3, K_4) size is $2 \times \mathrm{nb}$, which is double the block size in bytes (or $16 \times nb$ in bits). The main key K size is $128 \times 2^{\mathrm{x}}$ bits, where $\mathrm{x} \geq 0$. The maximum size of the main key is two times the blocksize. This limit was imposed to force each subkey to be composed by a sequence that went by the process at least twice. The number of times the permutation function is applied has been chosen to have the similar resistance to linear and differential cryptanalysis attacks on the subkey generation as on encrypting a block.

The FlexAEAD also uses a sequence of bits $(S_0 S_1 \ldots S_{n+m})$. This sequence is the same size of the associated data plus the message to be sent. It is generate by applying PF_{K3} over the NONCE to generate a base counter. The counter is divided in 32 bits chunks of data. Each chunk is treated as an unsigned number (little-endian) that is incremented for every block of the sequence by the function INC32. If the counter for a 64 bit block has the following bytes $(x01, x02, x03, x04, xFF, x01, x02, x03)$, after the INC32 function, the result is $(x02, x02, x03, x04, x00, x02, x02, x03)$.

The sequence will be unique for every NONCE. The chance of occurring overlapping sequences for two different NONCE is nonsignificant. Considering the maximum sie of the sequence is 2^{32}, for a 64 bits NONCE, there are 2^{32} non-overlapping sequences, so the probability of choosing two NONCEs with overlapping sequences is 2^{-64} $(p_{overlapping} = 2^{-32} \times 2^{-32} = 2^{-64})$. For a 128 bits NONCE, there are 2^{96} non-overlapping sequences, so the probability is 2^{-192}.

Another important characteristic is the fact that the sequence generation can run in parallel for every block. The function INC32 can add an arbitrary number to the base counter. On a multi-thread environment, the S_0 can be generate adding 1 to the base counter and in a parallel thread the S_{10} can be generate adding 11 to the base counter. Allow the cipher all available hardware. The sequence can be generated during the process of hashing the associate data or encrypting a data block, avoiding unnecessary memory allocation.

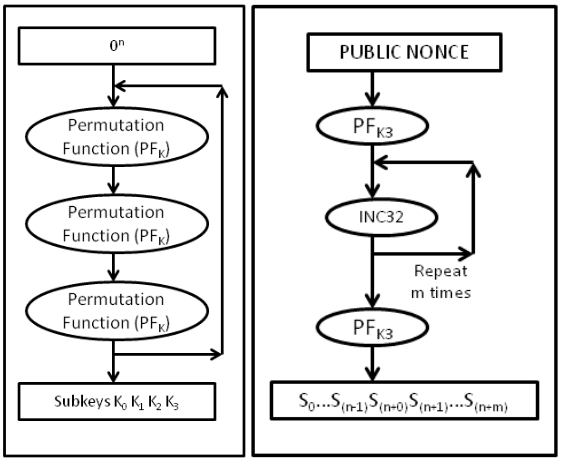

Figure 5. The K_0,K_1,K_2 and $S_0 S_1 \ldots S_m$ generation processes

To hash the associate data, first the associated data is divided in n blocks $(AD_0 AD_1 \ldots AD_n)$. The final block is padded with 0 bits. Each block (AD_x) is XORed with the correspondent (S_x) block and it is submitted to PF_{K2} to generate a intermediate state block (st_x). The process that each associated data block goes though is $(AD_x \rightarrow$

$XOR(s_x) \rightarrow PF_{K2} \rightarrow st_x)$.

To cipher the plain text message, it is broken into m plaintext blocks ($P_0 P_1 \ldots P_m$). The last block is padded with (10^{pb-1}), where pb is the number of padding bits to complete the block.

Each block (P_x) is XORed with the correspondent (S_x) block and it is submitted to PF_{K2} to generate a intermediate state block (st_x). The state (st_x) is submitted to PF_{K1}, XORed again with (S_x) and finally submitted to $PF_{K0}$to generate a ciphertext block (C_x). The process that each plaintext block goes though is ($P_x \rightarrow XOR(s_x) \rightarrow PF_{K2} \rightarrow st_x \rightarrow PF_{K1} \rightarrow XOR(S_x) \rightarrow PF_{K0} \rightarrow C_x$). It is important to observe that if the plaintext or associate data blocks are swapped in position, the generated checksum will be modified. This characteristic prevents reordering data attacks.

All intermediate state blocks are XORed together to generate a checksum. If the last message block was padded, the checksum is XORed with the bit sequence($1010 \ldots 10$). If there was no padding it is XORed with the bit sequence ($0101 \ldots 01$). After it the result is submitted to PF_{K0} function to generate the TAG used for authentication. The TAG length ($Tlen$) can be smaller than the block size, if it is adequate to the application. This is done by truncating the TAG on its $Tlen$ more significant bits (MSB_{Tlen}).

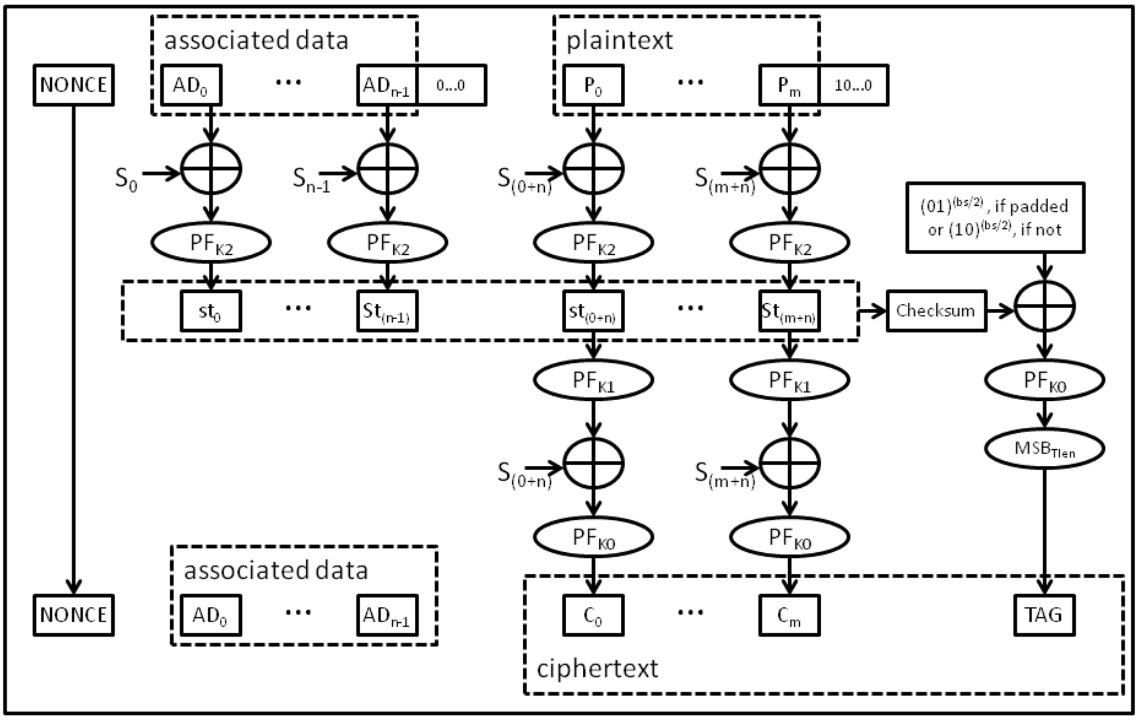

Figure 6. The FlexAEAD encryption diagram

For decryption, first the Associated Data is submitted to the same process as in encryption ($AD_x \rightarrow XOR(s_x) \rightarrow PF_{K2} \rightarrow st_x$). The Ciphertext is broken into blocks and the TAG is separated (as its size is known, the last part of the ciphertext is the TAG). The cipher text blocks are submitted to a reverse process ($C_x \rightarrow INVPF_{K0} \rightarrow XOR(S_x) \rightarrow INVPF_{k1} \rightarrow st_x \rightarrow INVPF_{K2} \rightarrow P_x$). During the process all (st_x) are XORed together. This checksum is XORed with bit sequence ($1010 \ldots 10$) then submitted to (PF_{K0}) to generate a TAG'. If the TAG' is equal to the received TAG, the message is valid and the original plaintext was not padded. If it is different the checksum is XORed with bit sequence $(0101 \dots 01)$ then submitted to (PF_{K0}) to generate a TAG''. If the TAG'' is equal to the received TAG, the message is valid and the original plaintext was padded. If neither calculated TAGs are equal to the received TAG, the message is invalid and it is discarded.

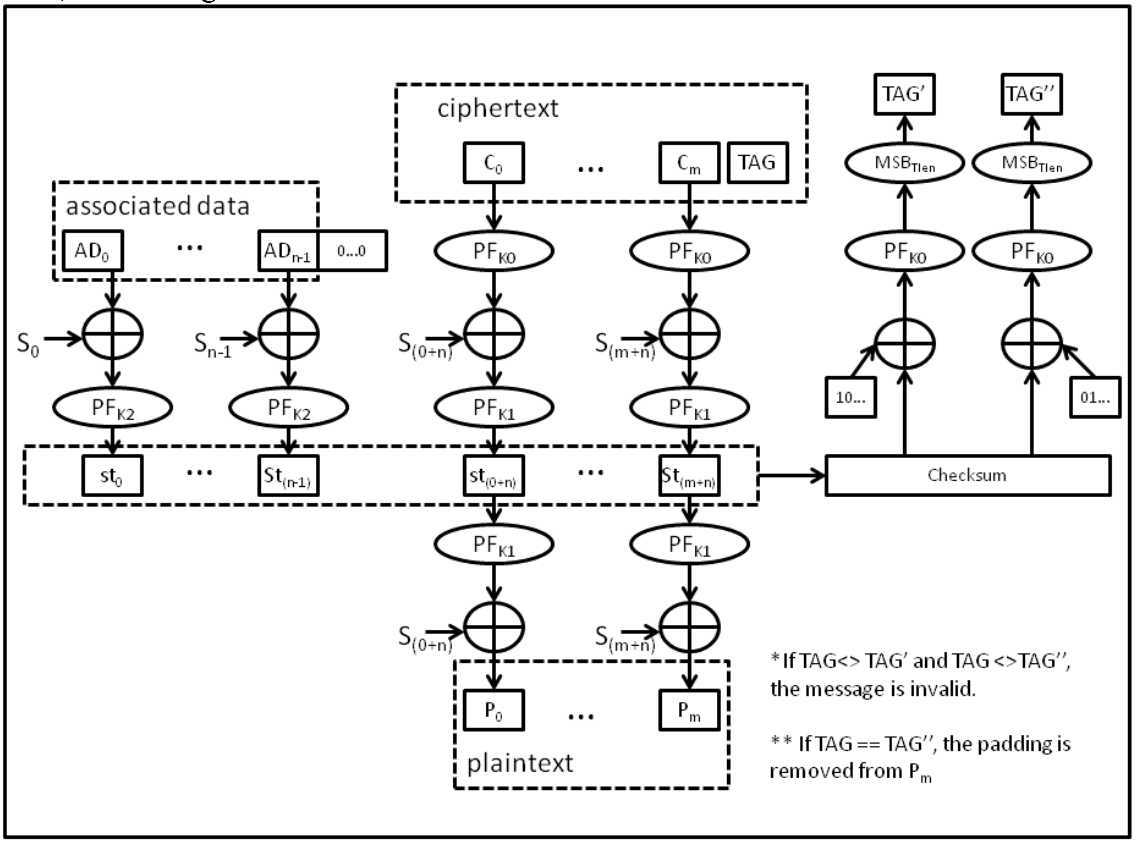

Figure 7. The FlexAEAD decryption diagram

2. Key and Block Size Selection

Although the FlexAEAD algorithm family allows several block and key size. A few variant were selected as concrete examples for this contest.

The family also allows the user to select the tag, used to validate the message, and nonce size. For this contest they will be the maximum allowed, depending on the variant. The maximum for them is the same as the block size for each variant.

The chosen variants are:

FlexAEAD128b064 – 128 bits key, 64 bits block, 64 bits nonce and 64 bits tag sizes

FlexAEAD128b128 – 128 bits key, 128 bits block, 128 bits nonce and 128 bits tag sizes

FlexAEAD256b256 – 256 bits key, 256 bits block, 256 bits nonce and 256 bits tag sizes

These variants were implemented and the NIST test vectors were successfully generated for them.

3. Differential Cryptanalysis

The differential cryptanalysis (BIHAM and SHAMIR, 1991) technique consists on analyzing of the probabilities of the differences on the cipher SBoxes inputs and outputs.

The differential and the linear cryptanalysis are almost the same as performed for the algorithm FlexAE (NASCIMENTO and XEXEO, 2018). The difference is the number of rounds that were incremented for better security.

The difference distribution table for AES SBox shows that the maximum probability for any pair $(\Delta X \neq 0, \Delta Y \neq 0)$ is $p = \frac{4}{256} = 2^{-6}$.

To encrypt each ciphertext block the PF_K is executed at least 3 times $(P_x \rightarrow XOR(s_x) \rightarrow PF_{K2} \rightarrow st_x) \rightarrow PF_{K1} \rightarrow XOR(S_x) \rightarrow PF_{K0} \rightarrow C_x)$. The number of rounds depends on the block size in bytes $(r = \log_2 nb + 2)$. The total of rounds for block sizes of 64, 128 and 256 bits are respectively 15, 18 and 21.

Due to the cipher architecture, the minimum number of active SBoxes in each round on the PF_K function is 2. The maximum probability can be calculated by $p_D = \prod_{i=1}^{(2\times(r-1))} 2^{-6}$ and the difficult of an attack based on differential cryptanalysis is $N_D \cong \frac{1}{P_D}$ (Heys, 2001).

Table 1. Difficult to perform a differential cryptanalysis attack

Block Size	Rounds (r-1)	Active SBoxes	p_D	N_D
64	14	28	2^{-168}	2^{168}
128	17	34	2^{-204}	2^{204}
256	20	40	2^{-240}	2^{240}

An attack based on a differential cryptanalysis is more difficult than a brute force attack when the cipher uses a 64 bit block size / 128 key size or 128 bit block size / 128 key size.

For the 256 bit block size / 256 key size the attack is easier than a brute force attack although it is not feasible.

4. Linear Cryptanalysis

The linear cryptanalysis (MATSUI, 1993) technique consists in evaluating the cipher using linear expressions to approximate the cipher results and calculating their biases of being true or false. The higher the bias, the easier is to uncover the key bits.

For AES SBox there are a total of 65025 possible linear expressions. The maximum bias on these expression is $\epsilon = \frac{16}{256} = 2^{-4}$.

After calculating the bias for every SBox, the next step is to verify the cipher structure effect and determine the best linear expressions for each round. In this stage it is easier to represent the linear expressions in graphic way. The following has a graphical representation of a linear approximation for all 5 rounds of the PF_K using 64 bits block size.

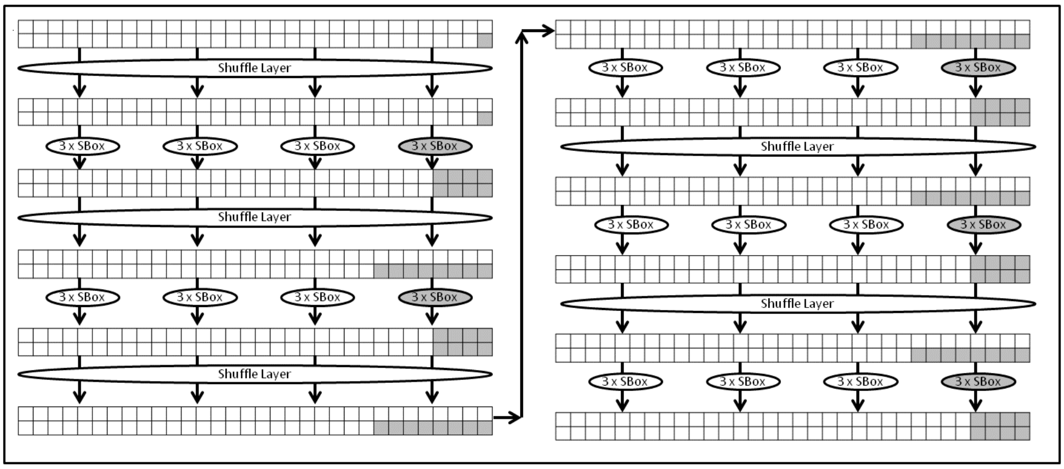

Figure 8. The linear expression graphical representation for FlexAEAD

The complexity of an attack is determined by the number of chosen plaintext pair (N_L) which can be calculate from the bias $N_L = \frac{1}{\epsilon^2}$ (HEYS, 2001). On the linear cryptanalysis, if the number of active SBox is known (n), the bias (ϵ) can be determined subtracting (0.5) from the probability (p) calculated using the Piling-up Lemma $\mathrm{p} = \frac{1}{2} + 2^{\mathrm{n}-1} \prod_{\mathrm{i}=1}^{\mathrm{n}} \left(\mathrm{p_i} - \frac{1}{2}\right)$ (MATSUI,1993): $\epsilon = p - 0.5$.

Table 2. Difficult to perform a linear cryptanalysis attack

Block Size	Rounds (r)	Active SBox	Maximum Bias	$N_L = \frac{1}{\epsilon^2}$
64	15	45	$\epsilon = 2^{-136}$	$N_L = 2^{272}$
128	18	54	$\epsilon = 2^{-163}$	$N_L = 2^{326}$
256	21	63	$\epsilon = 2^{-190}$	$N_L = 2^{380}$

An attack based on a linear cryptanalysis is more difficult than a brute force attack making it impractical.

5. Using the cipher to generate a pseudorandom sequence

The cipher was used to encrypted a block full of zeros again and again with the same key. The resulted were submitted to the dieharder toll. The sequence passed on all tests except on a few that it randomly presented as “WEAK”. If the NONCE or the KEY is changed or only that test is repeated, the test returned PASSED. This indicates that it is not possible to infer any pattern from the generated sequence. The test was performed on all four variants of the cipher presented on this document (FlexAEAD128b064, FlexAEAD128b128 and FlexAEAD256b256). The testing results example and the code used to generate the sequence for the dieharder tool are on the appendices.

6. Cipher family performance

The FlexAEAD family has inherited several functions from the FlexAE family, which presented good time performance in CPU cycles and RAM (NASCIMENTO and XEXEO,2017), when compared to other cipher. Although it is expected the FlexAEAD performance won’t be as good as to FlexAE, new tests will be necessary to evaluate the new family performance.

The main reason for the difference was the inclusion of a second XOR of the encrypting block with the Sx and another execution of the PF_K function. These modifications were necessary to avoid a reordering data attack.

The FlexAEAD cipher family uses only simple function like XOR, lookup table, for SBox Layer, or bits reorganization, for block shuffle layer. The block shuffle layer is simple to be implemented in hardware and it is expected to have a great performance (basically only wires changing the bits positions). The function in software is not optimized for large word processors like 64 bits. But these high end processors normally have multiples cores that can be used in parallel due to the cipher characteristics, compensating the deficiency.

For the FlexAE, the FELICS framework from CRYPTOLUX research group were used, but it was compared to non-authenticated block ciphers like AES. This time the SUPERCOP tool (BERNSTEIN and LANGE) was used and the FlexAEAD implementations were compared to the following CAESAR (BERNSTEIN) finalist implementations that were available at the SUPERCOP package: ascon128v11 (ASCON cipher), acorn128v3 (ACORN cipher), aegis128l (AEGIS-128 cipher) and deoxysi128v141 (Deoxys-II cipher).

To perform the tests, a Linux Ubuntu 18.04.2 LTS machine with the processor Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz were used. The results have shown that the actual FlexAEAD implementation uses more CPU cycles than the other ciphers.

6. Conclusion and future works

This paper describes the FlexAEAD cipher family. This cipher was tailored to be lightweight and flexible. Its security was analyzed for three variants with concrete values against linear and differential cryptanalysis attacks. The result is summarized on Table 3. Their capacity to generate a pseudorandom sequence was also confirmed.

Table 3. Variant parameters and cryptanalysis difficulty

Variant	Parameters sizes (in bits)				Cryptanalysis difficulty	
	Key	Block	Nonce	Tag	Linear	Differential
FlexAEAD128b064	128	64	64	64	2^{272}	2^{168}
FlexAEAD128b128	128	128	128	128	2^{326}	2^{204}
FlexAEAD256b256	256	256	256	256	2^{380}	2^{240}

An optimized version of the cipher will be implemented to compare its performance against the other participants. One performance advantage is its capacity to allow parallel computing, each block can be calculated by a different thread in any order. This characteristic is an advantage when using multicore processors.

References

BERNSTEIN, D. J.; LANGE, T. eds. eBACS: ECRYPT Benchmarking of Cryptographic Systems. URL: <https://bench.cr.yp.to> Access Date: Feb 28th 2019.

BERNSTEIN, D. J. Cryptographic competitions. URL: < https://competitions.cr.yp.to>

Access Date: Feb 28th 2019.

BIHAM, E.; SHAMIR, A. Differential cryptanalysis of DES-like cryptosystems. Journal of CRYPTOLOGY, 4, n. 1, 1991. 3-72.

CRYPTOLUX RESEARCH GROUP - UNIVERSITY OF LUXEMBOURG. Lightweight Block Ciphers, 2016. URL: <https://www.cryptolux.org/index.php/Lightweight_Block_Ciphers>. Access Date: Feb 28th 2019.

DAEMEN, J.; RIJMEN, V. Specification for the advanced encryption standard (AES). Federal Information Processing Standards Publication, 2001.

DINU, D. et al. FELICS – Fair Evaluation of Lightweight Cryptographic Systems, jul. 2015. URL: <http://csrc.nist.gov/groups/ST/lwc-workshop2015/papers/session7-dinu-paper.pdf>. Access Date: Feb 28th 2019.

EVEN, S.; MANSOUR, Y. A construction of a cipher from a single pseudorandom permutation. Journal of Cryptology, 10, 1997. 151-161.

JUTLA, C. S. Encryption modes with almost free message integrity. International Conference on the Theory and Applications of Cryptographic Techniques, 2001. 529-544.

MATSUI, M. Linear cryptanalysis method for DES cipher. Workshop on the Theory and Application of of Cryptographic Techniques, 1993. 386-397.

NASCIMENTO, E. M.; XEXÉO, J.A.M. "A flexible authenticated lightweight cipher using Even-Mansour construction". 2017 IEEE International Conference on Communications (ICC), Paris, 2017, pp. 1-6. (doi: 10.1109/ICC.2017.7996734). URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7996734&isnumber=7996317>. Access Date: Feb 28th 2019.

NASCIMENTO, E. M. “Algoritmo de Criptografia Leve com Utilização de Autenticação”. 2017. 113p. Dissertação (mestrado) - Instituto Militar de Engenharia, Rio de Janeiro, 2017. URL: <http://www.comp.ime.eb.br/pos/arquivos/publicacoes/dissertacoes/2017/2017-Eduardo.pdf>. Access Date: Feb 28th 2019.

NASCIMENTO, E. M.; XEXÉO, J.A.M. A Lightweight Cipher with Integrated Authentication. In: CONCURSO DE TESES E DISSERTAÇÕES - SIMPÓSIO BRASILEIRO EM SEGURANÇA DA INFORMAÇÃO E DE SISTEMAS COMPUTACIONAIS (SBSEG), 18., 2018, 1. Anais Estendidos do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais. Porto Alegre: Sociedade Brasileira de Computação, oct. 2018 . p. 25 - 32.

APPENDICE A – Direct and Inverse AES SBox

Table 3. Direct AES SBox

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0	0x63	0x7C	0x77	0x7B	0xF2	0x6B	0x6F	0xC5	0x30	0x01	0x67	0x2B	0xFE	0xD7	0xAB	0x76
0x1	0xCA	0x82	0xC9	0x7D	0xFA	0x59	0x47	0xF0	0xAD	0xD4	0xA2	0xAF	0x9C	0xA4	0x72	0xC0
0x2	0xB7	0xFD	0x93	0x26	0x36	0x3F	0xF7	0xCC	0x34	0xA5	0xE5	0xF1	0x71	0xD8	0x31	0x15
0x3	0x04	0xC7	0x23	0xC3	0x18	0x96	0x05	0x9A	0x07	0x12	0x80	0xE2	0xEB	0x27	0xB2	0x75
0x4	0x09	0x83	0x2C	0x1A	0x1B	0x6E	0x5A	0xA0	0x52	0x3B	0xD6	0xB3	0x29	0xE3	0x2F	0x84
0x5	0x53	0xD1	0x00	0xED	0x20	0xFC	0xB1	0x5B	0x6A	0xCB	0xBE	0x39	0x4A	0x4C	0x58	0xCF
0x6	0xD0	0xEF	0xAA	0xFB	0x43	0x4D	0x33	0x85	0x45	0xF9	0x02	0x7F	0x50	0x3C	0x9F	0xA8
0x7	0x51	0xA3	0x40	0x8F	0x92	0x9D	0x38	0xF5	0xBC	0xB6	0xDA	0x21	0x10	0xFF	0xF3	0xD2
0x8	0xCD	0x0C	0x13	0xEC	0x5F	0x97	0x44	0x17	0xC4	0xA7	0x7E	0x3D	0x64	0x5D	0x19	0x73
0x9	0x60	0x81	0x4F	0xDC	0x22	0x2A	0x90	0x88	0x46	0xEE	0xB8	0x14	0xDE	0x5E	0x0B	0xDB
0xA	0xE0	0x32	0x3A	0x0A	0x49	0x06	0x24	0x5C	0xC2	0xD3	0xAC	0x62	0x91	0x95	0xE4	0x79
0xB	0xE7	0xC8	0x37	0x6D	0x8D	0xD5	0x4E	0xA9	0x6C	0x56	0xF4	0xEA	0x65	0x7A	0xAE	0x08
0xC	0xBA	0x78	0x25	0x2E	0x1C	0xA6	0xB4	0xC6	0xE8	0xDD	0x74	0x1F	0x4B	0xBD	0x8B	0x8A
0xD	0x70	0x3E	0xB5	0x66	0x48	0x03	0xF6	0x0E	0x61	0x35	0x57	0xB9	0x86	0xC1	0x1D	0x9E
0xE	0xE1	0xF8	0x98	0x11	0x69	0xD9	0x8E	0x94	0x9B	0x1E	0x87	0xE9	0xCE	0x55	0x28	0xDF
0xF	0x8C	0xA1	0x89	0x0D	0xBF	0xE6	0x42	0x68	0x41	0x99	0x2D	0x0F	0xB0	0x54	0xBB	0x16

Table 4. Reverse AES SBox

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0	0x52	0x09	0x6A	0xD5	0x30	0x36	0xA5	0x38	0xBF	0x40	0xA3	0x9E	0x81	0xF3	0xD7	0xFB
0x1	0x7C	0xE3	0x39	0x82	0x9B	0x2F	0xFF	0x87	0x34	0x8E	0x43	0x44	0xC4	0xDE	0xE9	0xCB
0x2	0x54	0x7B	0x94	0x32	0xA6	0xC2	0x23	0x3D	0xEE	0x4C	0x95	0x0B	0x42	0xFA	0xC3	0x4E
0x3	0x08	0x2E	0xA1	0x66	0x28	0xD9	0x24	0xB2	0x76	0x5B	0xA2	0x49	0x6D	0x8B	0xD1	0x25
0x4	0x72	0xF8	0xF6	0x64	0x86	0x68	0x98	0x16	0xD4	0xA4	0x5C	0xCC	0x5D	0x65	0xB6	0x92
0x5	0x6C	0x70	0x48	0x50	0xFD	0xED	0xB9	0xDA	0x5E	0x15	0x46	0x57	0xA7	0x8D	0x9D	0x84
0x6	0x90	0xD8	0xAB	0x00	0x8C	0xBC	0xD3	0x0A	0xF7	0xE4	0x58	0x05	0xB8	0xB3	0x45	0x06
0x7	0xD0	0x2C	0x1E	0x8F	0xCA	0x3F	0x0F	0x02	0xC1	0xAF	0xBD	0x03	0x01	0x13	0x8A	0x6B
0x8	0x3A	0x91	0x11	0x41	0x4F	0x67	0xDC	0xEA	0x97	0xF2	0xCF	0xCE	0xF0	0xB4	0xE6	0x73
0x9	0x96	0xAC	0x74	0x22	0xE7	0xAD	0x35	0x85	0xE2	0xF9	0x37	0xE8	0x1C	0x75	0xDF	0x6E
0xA	0x47	0xF1	0x1A	0x71	0x1D	0x29	0xC5	0x89	0x6F	0xB7	0x62	0x0E	0xAA	0x18	0xBE	0x1B
0xB	0xFC	0x56	0x3E	0x4B	0xC6	0xD2	0x79	0x20	0x9A	0xDB	0xC0	0xFE	0x78	0xCD	0x5A	0xF4
0xC	0x1F	0xDD	0xA8	0x33	0x88	0x07	0xC7	0x31	0xB1	0x12	0x10	0x59	0x27	0x80	0xEC	0x5F
0xD	0x60	0x51	0x7F	0xA9	0x19	0xB5	0x4A	0x0D	0x2D	0xE5	0x7A	0x9F	0x93	0xC9	0x9C	0xEF
0xE	0xA0	0xE0	0x3B	0x4D	0xAE	0x2A	0xF5	0xB0	0xC8	0xEB	0xBB	0x3C	0x83	0x53	0x99	0x61
0xF	0x17	0x2B	0x04	0x7E	0xBA	0x77	0xD6	0x26	0xE1	0x69	0x14	0x63	0x55	0x21	0x0C	0x7D

APPENDICE B – encrypt-dieharder.c code to generate pseudorandom sequence

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "encript.c"

int main (  ) {
   unsigned char *npub;
   unsigned char *k;
   unsigned char *state;
   struct FlexAEADv1 flexaeadv1;
   k = malloc(KEYSIZE);
   memset( k, 0x00, KEYSIZE);
   npub = malloc(BLOCKSIZE);
   memset( npub, 0x00, BLOCKSIZE);
   FlexAEADv1_init( &flexaeadv1, k );
   fprintf(stderr, "FlexAEADv1 ZERO %d %d\n", BLOCKSIZE*8, KEYSIZE*8 );
   // ### reset the counter and checksum
   memcpy( flexaeadv1.counter, npub, NONCESIZE);
   dirPFK(     flexaeadv1.counter,     flexaeadv1.nBytes,     (flexaeadv1.subkeys     +
   (4*flexaeadv1.nBytes)),  flexaeadv1.nRounds, flexaeadv1.state );
   state = malloc(BLOCKSIZE);
   while(1)
   {
        memset( state, 0x00, BLOCKSIZE );
        inc32( flexaeadv1.counter, flexaeadv1.nBytes, 1 );
        encryptBlock( &flexaeadv1, state);
        fwrite(state, 1, flexaeadv1.nBytes, stdout);
   }
   free(state);
}

// execution example: ./encrypt-dieharder | dieharder -a -g 200
```

APPENDICE C – dieharder tool results example for FlexAEADv256b256

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  5.91e+05  |3518119865|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.53243263|  PASSED
      diehard_operm5|   0|   1000000|     100|0.92541253|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.15594265|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.97400698|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.34139275|  PASSED
        diehard_opso|   0|   2097152|     100|0.32834173|  PASSED
        diehard_oqso|   0|   2097152|     100|0.91056284|  PASSED
         diehard_dna|   0|   2097152|     100|0.38464814|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.34100720|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.96884054|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.96913730|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.20717814|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.09572503|  PASSED
```

diehard_squeeze	0	100000	100	0.49830589	PASSED
diehard_sums	0	100	100	0.42558220	PASSED
diehard_runs	0	100000	100	0.03886906	PASSED
diehard_runs	0	100000	100	0.38309375	PASSED
diehard_craps	0	200000	100	0.11990794	PASSED
diehard_craps	0	200000	100	0.71676496	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.54813906	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.96626464	PASSED
sts_monobit	**1**	**100000**	**100**	**0.99996188**	**WEAK**
sts_runs	2	100000	100	0.24298167	PASSED
sts_serial	1	100000	100	0.77122722	PASSED
sts_serial	2	100000	100	0.98176924	PASSED
sts_serial	3	100000	100	0.69443393	PASSED
sts_serial	3	100000	100	0.26827062	PASSED
sts_serial	4	100000	100	0.68843008	PASSED
sts_serial	4	100000	100	0.43152701	PASSED
sts_serial	5	100000	100	0.70013670	PASSED
sts_serial	5	100000	100	0.92175886	PASSED
sts_serial	6	100000	100	0.63596468	PASSED
sts_serial	6	100000	100	0.63897130	PASSED
sts_serial	7	100000	100	0.36519471	PASSED
sts_serial	7	100000	100	0.87776520	PASSED
sts_serial	8	100000	100	0.78504105	PASSED
sts_serial	8	100000	100	0.68670977	PASSED
sts_serial	9	100000	100	0.53458473	PASSED
sts_serial	9	100000	100	0.96686776	PASSED
sts_serial	10	100000	100	0.93208301	PASSED
sts_serial	10	100000	100	0.41830759	PASSED
sts_serial	11	100000	100	0.44154753	PASSED
sts_serial	11	100000	100	0.04949517	PASSED
sts_serial	12	100000	100	0.50092968	PASSED
sts_serial	12	100000	100	0.19714967	PASSED
sts_serial	13	100000	100	0.69263841	PASSED
sts_serial	13	100000	100	0.84095563	PASSED
sts_serial	14	100000	100	0.24424891	PASSED
sts_serial	14	100000	100	0.88271258	PASSED
sts_serial	15	100000	100	0.38119541	PASSED
sts_serial	15	100000	100	0.66073910	PASSED
sts_serial	16	100000	100	0.68054873	PASSED
sts_serial	16	100000	100	0.75566807	PASSED
rgb_bitdist	1	100000	100	0.06100868	PASSED
rgb_bitdist	2	100000	100	0.33521314	PASSED
rgb_bitdist	3	100000	100	0.96149073	PASSED
rgb_bitdist	4	100000	100	0.52070848	PASSED
rgb_bitdist	5	100000	100	0.98851270	PASSED
rgb_bitdist	6	100000	100	0.13418091	PASSED
rgb_bitdist	7	100000	100	0.13906705	PASSED
rgb_bitdist	8	100000	100	0.51265948	PASSED
rgb_bitdist	9	100000	100	0.73103752	PASSED
rgb_bitdist	10	100000	100	0.57102500	PASSED
rgb_bitdist	11	100000	100	0.56515679	PASSED
rgb_bitdist	**12**	**100000**	**100**	**0.99917966**	**WEAK**
rgb_minimum_distance	2	10000	1000	0.53587905	PASSED
rgb_minimum_distance	3	10000	1000	0.34210762	PASSED
rgb_minimum_distance	4	10000	1000	0.58613763	PASSED
rgb_minimum_distance	5	10000	1000	0.19434753	PASSED
rgb_permutations	2	100000	100	0.68699976	PASSED
rgb_permutations	3	100000	100	0.17402171	PASSED
rgb_permutations	4	100000	100	0.38105709	PASSED
rgb_permutations	5	100000	100	0.93408952	PASSED
rgb_lagged_sum	0	1000000	100	0.71633791	PASSED
rgb_lagged_sum	1	1000000	100	0.82789524	PASSED
rgb_lagged_sum	2	1000000	100	0.82437890	PASSED
rgb_lagged_sum	3	1000000	100	0.80529476	PASSED
rgb_lagged_sum	4	1000000	100	0.21479258	PASSED
rgb_lagged_sum	5	1000000	100	0.02661369	PASSED
rgb_lagged_sum	6	1000000	100	0.63510522	PASSED
rgb_lagged_sum	7	1000000	100	0.51597148	PASSED
rgb_lagged_sum	8	1000000	100	0.67268338	PASSED
rgb_lagged_sum	9	1000000	100	0.29814160	PASSED
rgb_lagged_sum	10	1000000	100	0.73545520	PASSED
rgb_lagged_sum	11	1000000	100	0.94261731	PASSED
rgb_lagged_sum	12	1000000	100	0.56493673	PASSED
rgb_lagged_sum	13	1000000	100	0.32623547	PASSED
rgb_lagged_sum	14	1000000	100	0.86849070	PASSED
rgb_lagged_sum	15	1000000	100	0.20498726	PASSED
rgb_lagged_sum	16	1000000	100	0.71300651	PASSED
rgb_lagged_sum	17	1000000	100	0.10728202	PASSED
rgb_lagged_sum	18	1000000	100	0.66967662	PASSED
rgb_lagged_sum	19	1000000	100	0.87808186	PASSED
rgb_lagged_sum	20	1000000	100	0.01152262	PASSED
rgb_lagged_sum	21	1000000	100	0.53744897	PASSED
rgb_lagged_sum	22	1000000	100	0.41257966	PASSED

```
       rgb_lagged_sum|  23|   1000000|     100|0.57216229|  PASSED
       rgb_lagged_sum|  24|   1000000|     100|0.88346704|  PASSED
       rgb_lagged_sum|  25|   1000000|     100|0.41339647|  PASSED
       rgb_lagged_sum|  26|   1000000|     100|0.71925925|  PASSED
       rgb_lagged_sum|  27|   1000000|     100|0.75322746|  PASSED
       rgb_lagged_sum|  28|   1000000|     100|0.63884993|  PASSED
       rgb_lagged_sum|  29|   1000000|     100|0.98819306|  PASSED
       rgb_lagged_sum|  30|   1000000|     100|0.33043748|  PASSED
       rgb_lagged_sum|  31|   1000000|     100|0.10463550|  PASSED
       rgb_lagged_sum|  32|   1000000|     100|0.46124090|  PASSED
      rgb_kstest_test|   0|     10000|    1000|0.18623770|  PASSED
      dab_bytedistrib|   0|  51200000|       1|0.71777194|  PASSED
              dab_dct| 256|     50000|       1|0.01985939|  PASSED
Preparing to run test 207.  ntuple = 0
         dab_filltree|  32|  15000000|       1|0.17292794|  PASSED
         dab_filltree|  32|  15000000|       1|0.35405515|  PASSED
Preparing to run test 208.  ntuple = 0
        dab_filltree2|   0|   5000000|       1|0.68458837|  PASSED
        dab_filltree2|   1|   5000000|       1|0.04958262|  PASSED
Preparing to run test 209.  ntuple = 0
         dab_monobit2|  12|  65000000|       1|0.34004526|  PASSED
```

dieharder rerun sts_monobit test

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  4.13e+05  |3345856669|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
         rgb_bitdist|  12|    100000|     100|0.85373615|  PASSED
```

dieharder rerun rgb_bitdist test

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  4.15e+05  |3664988861|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
         sts_monobit|   1|    100000|     100|0.35268451|  PASSED
```