FlexAEAD v1.1 -A Lightweight AEAD Cipher with Integrated Authentication

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

***Abstract*— This paper describes a symmetrical block cipher family – FlexAEAD v1.1. This is an updated version of the work presented as round 1 candidate on the contest for NIST lightweight crypto standardization process. 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. The FlexAEAD also supports the authentication of the Associated Data (AD). The version 1.1 makes the algorithm resistant to iterated differential attacks. It also resolves a padding attack on the AD that allowed messages to have the same tag if the last AD block was filled with zeros.**

***Index Terms*— authenticated encryption, lightweight, NIST LWC**

I. INTRODUCTION

ON August 2018, the National Institute of Standards and Technology (NIST) published call for algorithm (NIST, 2018) describing the contest and requirements for a new lightweight authenticated encryption with associated data (AEAD) algorithm and an optional hash algorithm.

The FleaxAEAD algorithm family was inscribed in the contest and analyzed by several researchers. The cipher family is an evolution of the FlexAE algorithm presented at IEEE ICC2017 (Paris – France) and SBSEG2018 (Natal – Brazil). The first difference is the capacity to allow the validation of an associated data together with the encrypted data. The new family also resolved a reorder block attack.

During NIST contest first round, independent researchers found a weakness related to the associated data padding and an iterated differential attack. The weakness were solved and resulted on the cipher version 1.1.

This specification and security claims for the cipher variations were revised and they are presented on this paper. The cipher source code is available on the URL https://github.com/edunasci/FlexAEAD.

[1] Eduardo Marsola do Nascimento, Petróleo Brasileiro S.A. – Petrobras, Rio de Janeiro, RJ (email: edunasci@yahoo.com).

2 José Antônio Moreira Xexéo, Instituto Militar de Engenharia, Rio de Janeiro, RJ (email: xexeo@ime.eb.br).

II. 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 ([1]).

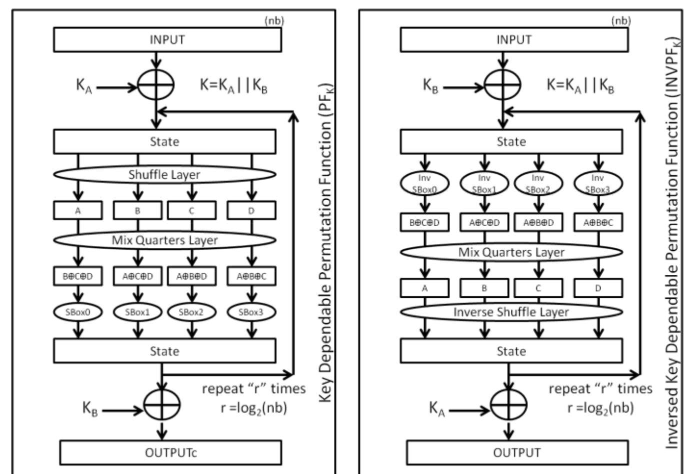

Fig. 1. The permutation function PF_K and its inverse $INVPF_K$.

On the (PF_K), after the XOR with K_A, the block is transformed by a shuffle layer, where a *nb* bytes input is reordered as $(\mathrm{b}[0], \mathrm{b}\left[\frac{\mathrm{nb}}{2}\right], \mathrm{b}[1], \mathrm{b}\left[\frac{\mathrm{nb}}{2}+1\right], \ldots, \mathrm{b}\left[\frac{\mathrm{nb}}{2}-1\right]$, $\mathrm{b}[\mathrm{nb}-1]$.

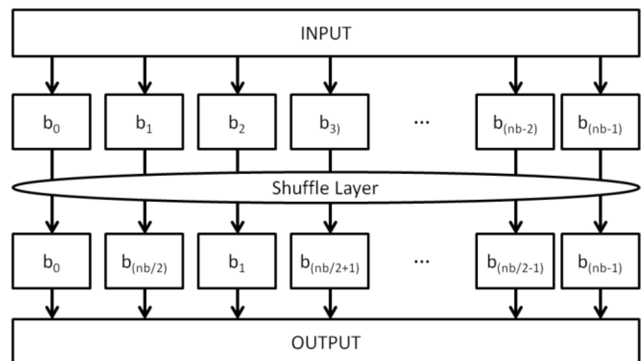

Fig. 2. The Shuffle Layer.

[1] This function were rewritten to avoid an efficient iterated truncated differential attacks proposed by Mostafizar Rahman, Dhiman Saha and Goutam Paul during the contest discussions.

After the shuffle Layer, the input is divided in quarters and the mix quarters layers combine them together. Considering the quarters (A, B, C, D) as input, the output will be $(B \oplus C \oplus D, A \oplus C \oplus D, A \oplus B \oplus D, A \oplus B \oplus C)$. The function is its own inverse, if the output is submitted again to the function, it will generate the original input. A difference on one byte will generate differences in 3 bytes, in different quarters.

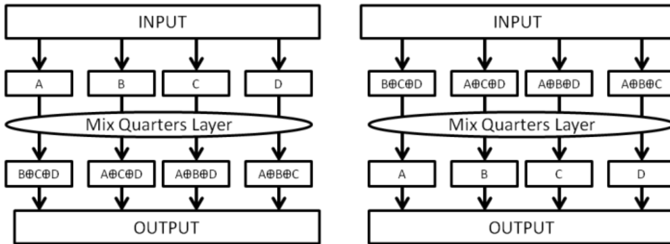

Fig. 3. Mix Quarters Layer.

The next is the SBox layer, where each quarter suffers a non-linear transformation using a different SBox. The first SBox is the AES SBox, the other SBoxes are generate using the process as the first (multiplicative inverse on the $GF2^8$) using different irreducible polynomial (IP), multiplicative constant (MC) and additive constants (A).

TABLE I
Parameters used to create FlexAEAD SBoxes

SBox	IP	MC	AC
SBox0	$x^8 + x^4 + x^3 + x^1 + 1$ (0b100011011)	0x1F	0x63
SBox1	$x^8 + x^4 + x^3 + x^2 + 1$ (0b100011101)	0x3D	0x95
SBox2	$x^8 + x^5 + x^3 + x^1 + 1$ (0b100101011)	0x3B	0xA6
SBox3	$x^8 + x^5 + x^3 + x^2 + 1$ (0b100101101)	0x37	0xD9

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

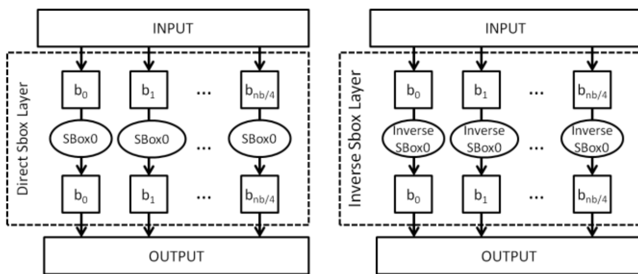

Fig. 4. The SBox Layer.

The number of rounds (r) on this construction is $r = \log_2 nb$, 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 5.

```
dirPF(INPUT[nb], K[nk])

K_A = K[1..(nk/2)]
K_B = K[((nk/2)+1)..nk]
state = INPUT ⊕ K_A
for ( i =1 to log(nb)]
   state = ShuffleLayer(state)
   state = MixQuartersLayer(state)
   state = dirSBoxLayer(state)
end for
OUTPUT = state ⊕ K_B
```

```
invPF(INPUT[nb], K[nk])

K_A = K[1..(nk/2)]
K_B = K[((nk/2)+1)..nk]
state = INPUT ⊕ K_B
for ( i =1 to log(nb)]
   state = MixQuartersLayer(state)
   state = invShuffleLayer(state)
   state = invSBoxLayer(state)
end for
OUTPUT = state ⊕ K_B
```

Fig. 5. 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 $2 \times$ 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 $(S_0 S_1 \dots S_{n+m})$. 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 added with the constant 0x11111111 for every block of the sequence by the function INC32([2]). 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 (x12,x13,x14,x15,x10,x14,x13,x14).

The sequence will be unique for every NONCE. The chance of occurring overlapping sequences for two different NONCE

[2] On the original cipher the constant added was 0x1 but it allowed a differential attack proposed by Maria Eichlseder, Daniel Kales and Markus Schofnegger.

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 S0 can be generate adding 0x11111111 to the base counter and in a parallel thread the S0 can be generate adding 0xBBBBBBBB (or $0xB \times$ 0x11111111) to the base counter. This allows the cipher to use multiples processors or core if available. The sequence can be generated during the process of hashing the associate data or encrypting a data block, avoiding unnecessary memory allocation.

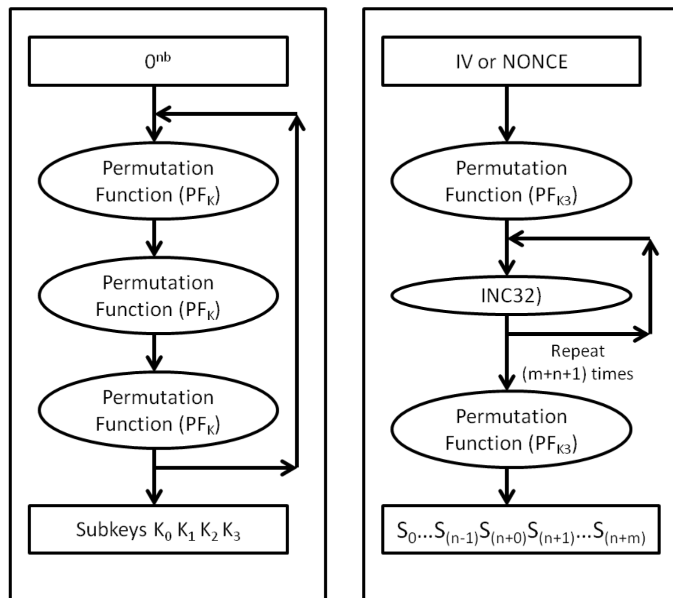

Fig. 6. The K_0,K_1,K_2 and $S_0S_1\dots S_{m+n}$ generation processes.

To hash the associate data, first the associated data is divided in n blocks $(AD_0AD_1 \dots AD_{n-1})$. The final block is padded with 10…0 bits ([3]). 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 $(AD_x \rightarrow XOR(s_x) \rightarrow PF_{K2} \rightarrow st_x)$. If the last block has been padded, the function PF_{K2} is applied twice: $(AD_{x(padded)} \rightarrow XOR(s_x) \rightarrow PF_{K2} \rightarrow PF_{K2} \rightarrow st_x)$.

The (S_n) block is submitted to PF_{K2} twice to generate a intermediate state block $(S_n \rightarrow PF_{K2} \rightarrow PF_{K2} \rightarrow (st_n)$. This operation was included[4] to avoid having the same tag, for different NONCEs, when both AD and M are empty. Another reason is to avoid having the same tag for $(N, A_{0\dots n-1}||P_0, P_{1\dots m-1})$ and $(N, A_{0\dots n-1}, P_{0\dots m-1})$.

To cipher the plain text message, it is broken into m plaintext blocks $(P_0P_1 \dots P_{m-1})$. 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$(1010 \dots 10)$. If there was no padding it is XORed with the bit sequence $(0101 \dots 01)$. 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}).

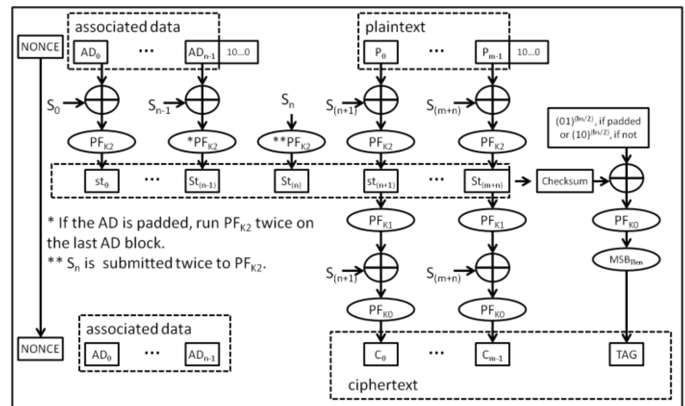

Fig. 7. The FlexAEAD encryption diagram.

For decryption, first the Associated Data is submitted to the same process as in encryption $(AD_x \rightarrow XOR(s_x) \rightarrow PF_{K2} \rightarrow st_x)$ or $(AD_{x(padded)} \rightarrow XOR(s_x) \rightarrow PF_{K2} \rightarrow PF_{K2} \rightarrow st_x)$. The (S_n) block is submitted to PF_{K2} twice $(S_n \rightarrow PF_{K2} \rightarrow PF_{K2} \rightarrow (st_n)$. 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 decryption process all (st_x) are XORed together. This checksum is XORed with bit sequence $(1010 \dots 10)$ 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

[3] The original cipher permitted the forgery extended length attack. The actual version solved the problem by using a resistant padding as suggested by by Alexandre Mège.

[4] Both problems where pointed by Maria Eichlseder on NIST LWC discussion forum.

equal to the received TAG, the message is invalid and it is discarded.

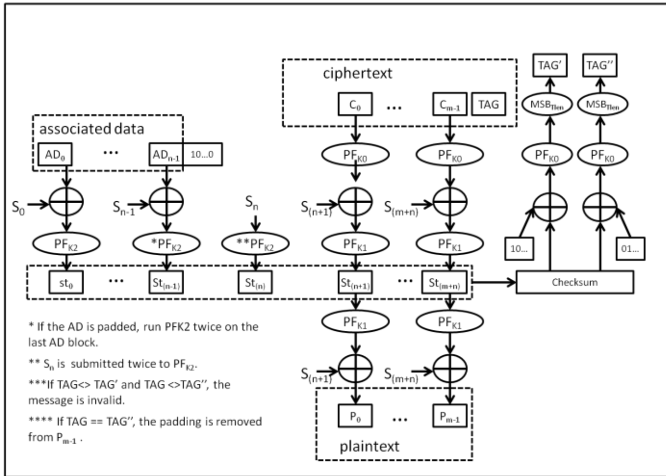

Fig. 8. The FlexAEAD decryption diagram.

III. 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 the NIST 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

FlexAEAD256b128 – 256 bits key, 128bits 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.

IV. 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 of the FlexAEAD are similar to the analysis performed on the algorithm FlexAE (NASCIMENTO and XEXEO, 2018). The differences are the number of rounds and the inclusion of the function mix adjacent bytes.

To analyze the differences of a specific SBox construction, a difference distribution table (DDT) is created. To create this table the input differences $(\Delta X = X' \oplus X'')$ and the output differences $(\Delta Y = Y' \oplus Y'')$ are calculated for every possible input pair (X', X''). The table columns are ΔY values and the lines are ΔX. Each cells contains the number of times that ΔX generates ΔY. Exemplifying, considering the AES SBox,

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 ($r = \log_2 nb$). The total of rounds for block sizes of 64, 128 and 256 bits are respectively 9, 12 and 15.

For a 64 bits block size: if the 1st round has 1 active SBox[5], the 2nd round will have 3 active SBoxes; the 3rd round will have 7 SBoxes; and from 4th round on, there will have 8 active SBoxes per round. On (r-1) or 8 rounds, there is 51 active Sboxes: $nActiveSboxes = 1 + 3 + 7 + (5 \times 8) = 51$.

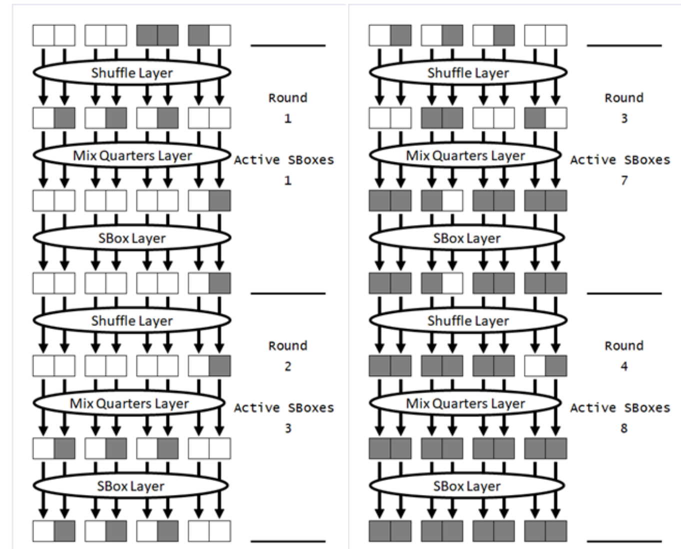

Fig. 9. Active sboxes after 4 rounds for 64 bits bock size.

For a 128 bits block size: if the 1st round has 1 active SBox, the 2nd round has a minimum of 3 active SBoxes; the 3rd round will have 7 active SBoxes; the 4th round – 15 active SBoxes; and from 5th round on, there will be 16 active SBoxes per round. On (r-1) or 11 rounds, there is 138 active Sboxes: $nActiveSboxes = 1 + 3 + 7 + 15 + (7 \times 16) = 138$.

[5] On the first round it is possible to control the input difference to force only one 1 active SBox.

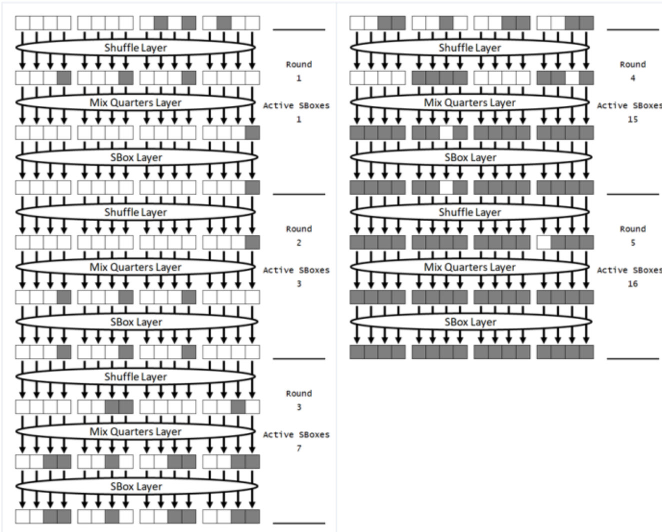

Fig. 10. Active sboxes after 5 rounds for 128 bits bock size.

For a 256 bits block size: there is 26 active SBoxes from round 1 to 4; the 5th round - 31 active SBoxes; from the 6th round on, there is 32 active SBoxes. On (r-1) or 14 rounds, there is 336 active Sboxes: $nActiveSboxes = 26 + 31 + (9 \times 32) = 336$.

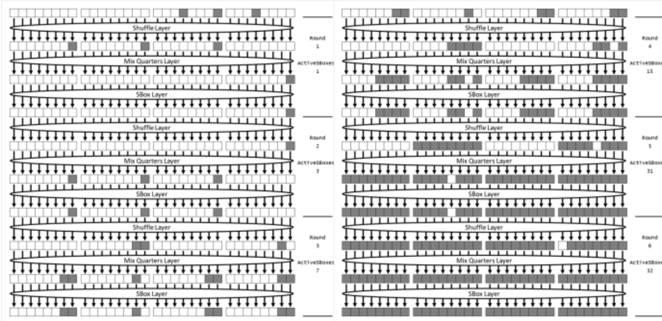

Fig. 11. Active sboxes after 6 rounds for 256 bits bock size.

The maximum probability can be calculated by $p_D = \prod_{i=1}^{(nActiveSboxes)} 2^{-6}$ and the difficult of an attack based on differential cryptanalysis is $N_D \cong \frac{1}{p_D}$ (Heys, 2001).

TABLE II
Difficult to perform a differential cryptanalysis attack

Block Size	Rounds (r-1)	Active SBoxes	p_D	N_D
64	8	51	2^{-306}	2^{306}
128	11	138	2^{-828}	2^{828}
256	14	336	2^{-2016}	2^{2016}

An attack based on a differential cryptanalysis is more difficult than a brute force attack in all cases.

V. 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.

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 $p = \frac{1}{2} + 2^{n-1} \prod_{i=1}^{n} \left(p_i - \frac{1}{2}\right)$ (MATSUI,1993): $\epsilon = p - 0.5$.

The number of active SBoxes on the linear cryptanalysis can be considered the same as the differential cryptanalysis per round due to the cipher its internal structure and the effect of the mix adjacent bytes function.

TABLE III
Difficult to perform a linear cryptanalysis attack

Block Size	Rounds (r)	Active SBox (r rounds)	Maximum Bias	$N_L = \frac{1}{\epsilon^2}$
64	9	59	$\epsilon = 2^{-178}$	$N_L = 2^{356}$
128	12	154	$\epsilon = 2^{-463}$	$N_L = 2^{926}$
256	15	368	$\epsilon = 2^{-1105}$	$N_L = 2^{2210}$

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

VI. 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, FlexAEAD256b128 and FlexAEAD256b256). The code used to generate the sequence for the dieharder tool is on the appendices.

VII. 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).

For the FlexAEAD, the eBAEAD - ECRYPT Benchmarking of Authenticated Ciphers from supercop framework (Bernstein, 2019) was used to compare the implementations with NIST LWC round2 candidates. A virtual machine with 2 dedicated processors (AMD EPYC 7501@2GHz) running Linux Ubuntu 19.10 was used to evaluate the performance. In total, 92 implementations were compared. The measure was done twice and the median was used for comparison.

The median time for encrypt 2048 bytes message with 2048 bytes associate data for the variants FlexAEAD128b064, FlexAEAD128b128, FlexAEAD256b128 and FlexAEAD256b256 are respectively 340206, 313413, 314486 and 223220 cpu cycles. It position against the other ciphers were 41st, 37th, 36th and 33rd. The FlexAEAD256b256 implementation is 8.7 times slower than the fastest implementation (ascon128av12 – 25643) but 95.2 faster than the slowest implementation compared (elephant160v1 - 21242710). The complete table with the comparison is available on the appendixes.

VIII. Conclusion

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 4. Their capacity to generate a pseudorandom sequence was also confirmed using the dieharder tool.

TABLE IV
Difficult to perform a differential cryptanalysis attack

Variant	Parameters sizes (in bits)				Cryptanalysis difficulty	
	Key	Block	Nonce	Tag	Linear	Differential
FlexAEAD128b064	128	64	64	64	2^{306}	2^{356}
FlexAEAD128b128	128	128	128	128	2^{828}	2^{926}
FlexAEAD256b128	256	128	128	128	2^{828}	2^{926}
FlexAEAD256b256	256	256	256	256	2^{2016}	2^{2210}

The cipher performance, was evaluated comparing its 4 variant against the 32 ciphers selected for round 2 of the NIST LWC contest. The tests show the FlexAEAD variants are faster than half of the implementations. 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.

For future works, the cipher implementation should be optimized to increase the performance. The cipher should also be implemented in hardware and compared to the other ciphers.

Appendix A – Direct and Inverse SBoxes

Direct SBox0 (AES SBox)

*	-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	-	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	-	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	-	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	-	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	-	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	-	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	-	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	-	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	-	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	-	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	-	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	-	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	-	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	-	E	F	9	1	6	D	8	9	9	1	8	E	C	5	2	D

		1	8	8	1	9	9	E	4	B	E	7	9	E	5	8	F
F	-	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Inverse SBox0 (AES SBox)

*	-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	-	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	-	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	-	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	-	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	-	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	-	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	-	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	-	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	-	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	-	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	-	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	-	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	-	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	-	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	-	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Direct SBox1

*	-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	95	A8	6C	C4	69	1F	3D	EC	8C	F8	B7	31	C1	3F	29	56
1	-	7E	D4	44	E0	E3	86	C7	F3	D8	F0	C0	0B	AC	4C	74	A1
2	-	60	C3	35	34	7D	87	2F	98	AE	97	1C	49	BC	A5	A6	1A
3	-	33	DF	27	55	58	03	DA	6E	09	48	1E	78	02	88	8F	DE
4	-	6F	53	D9	5E	A2	BD	22	61	E1	E2	9C	21	C8	CE	13	9F
5	-	08	75	94	16	36	D5	FB	40	01	79	EA	3A	6B	F2	52	E7
6	-	C6	BA	D7	A7	AB	B0	F5	FA	73	2B	B9	38	32	FE	68	9B
7	-	DB	AA	7B	43	37	9E	04	7A	39	1D	1B	D1	FF	64	57	2D
8	-	E8	FD	91	66	B3	59	17	7F	0E	DC	81	12	4E	A9	EF	F9
9	-	AF	CD	2E	80	76	62	CF	14	3B	8A	5F	2C	B1	41	F7	D6
A	-	5B	71	82	CA	15	3E	54	5C	23	4F	B5	FC	C5	7C	18	CC
B	-	B8	2A	84	D3	4D	4A	25	F6	8D	89	26	00	11	4B	CB	F1
C	-	3C	DD	65	28	B4	96	EB	BF	ED	83	07	9A	C2	8E	45	72
D	-	E6	93	AD	BE	E4	9D	24	19	46	E9	20	47	0C	06	92	E5
E	-	B2	BB	6D	30	85	42	99	0D	A3	5A	77	8B	5D	0F	05	EE
F	-	A4	50	B6	70	D2	51	D0	90	A0	63	0A	67	F4	6A	C9	10

Inverse SBox1

*	-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	BB	58	3C	35	76	EE	DD	CA	50	38	FA	1B	DC	E7	88	ED
1	-	FF	BC	8B	4E	97	A4	53	86	AE	D7	2F	7A	2A	79	3A	05
2	-	DA	4B	46	A8	D6	B6	BA	32	C3	0E	B1	69	9B	7F	92	26
3	-	E3	0B	6C	30	23	22	54	74	6B	78	5B	98	C0	06	A5	0D
4	-	57	9D	E5	73	12	CE	D8	DB	39	2B	B5	BD	1D	B4	8C	A9
5	-	F1	F5	5E	41	A6	33	0F	7E	34	85	E9	A0	A7	EC	43	9A
6	-	20	47	95	F9	7D	C2	83	FB	6E	04	FD	5C	02	E2	37	40
7	-	F3	A1	CF	68	1E	51	94	EA	3B	59	77	72	AD	24	10	87
8	-	93	8A	A2	C9	B2	E4	15	25	3D	B9	99	EB	08	B8	CD	3E
9	-	F7	82	DE	D1	52	00	C5	29	27	E6	CB	6F	4A	D5	75	4F
A	-	F8	1F	44	E8	F0	2D	2E	63	01	8D	71	64	1C	D2	28	90
B	-	65	9C	E0	84	C4	AA	F2	0A	B0	6A	61	E1	2C	45	D3	C7
C	-	1A	0C	CC	21	03	AC	60	16	4C	FE	A3	BE	AF	91	4D	96
D	-	F6	7B	F4	B3	11	55	9F	62	18	42	36	70	89	C1	3F	31
E	-	13	48	49	14	D4	DF	D0	5F	80	D9	5A	C6	07	C8	EF	8E
F	-	19	BF	5D	17	FC	66	B7	9E	09	8F	67	56	AB	81	6D	7C

Direct SBox2

*	-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

0	-	A6	9D	5F	08	3E	7B	F1	B0	8E	EC	2C	0C	69	B6	AD	ED
1	-	B2	60	E7	F8	E3	39	97	11	41	DB	AE	27	23	3F	67	51
2	-	C8	B3	A1	4B	62	A9	89	2E	04	20	0D	72	5A	26	19	7C
3	-	55	36	18	1B	C6	D4	66	0A	00	34	0E	74	22	B9	5D	D3
4	-	F5	CD	48	84	25	73	50	14	C4	43	45	6F	31	E8	86	E9
5	-	F7	7A	E5	D6	17	32	CC	E0	D8	C2	E6	35	79	29	AF	77
6	-	3B	90	EE	12	F9	02	1C	BA	96	DE	FB	A4	A2	CB	94	A3
7	-	91	57	8B	3C	F2	2F	CF	61	80	E4	4D	9C	5B	15	78	B1
8	-	0F	AB	13	A7	B5	44	B7	70	03	83	4C	98	DD	4F	FF	8A
9	-	F3	FA	30	4E	33	D0	42	D5	6D	5C	81	95	D2	2B	01	99
A	-	6A	56	AC	B4	07	CA	9E	EF	1A	EA	88	C1	93	8D	E1	7D
B	-	FD	A5	F0	3A	E2	B8	0B	C5	49	6E	05	71	46	1F	2A	8F
C	-	68	F6	D9	38	82	47	FC	7E	09	37	F4	1D	9F	A0	A8	52
D	-	DA	24	FE	75	6C	BC	C3	63	C0	9B	10	BD	BF	1E	40	4A
E	-	59	16	5E	BB	54	C7	EB	64	8C	9A	06	3D	76	28	21	BE
F	-	D1	85	87	AA	53	CE	DF	65	58	DC	7F	D7	C9	6B	2D	92

Inverse SBox2

*	-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	38	9E	65	88	28	BA	EA	A4	03	C8	37	B6	0B	2A	3A	80
1	-	DA	17	63	82	47	7D	E1	54	32	2E	A8	33	66	CB	DD	BD
2	-	29	EE	3C	1C	D1	44	2D	1B	ED	5D	BE	9D	0A	FE	27	75
3	-	92	4C	55	94	39	5B	31	C9	C3	15	B3	60	73	EB	04	1D
4	-	DE	18	96	49	85	4A	BC	C5	42	B8	DF	23	8A	7A	93	8D
5	-	46	1F	CF	F4	E4	30	A1	71	F8	E0	2C	7C	99	3E	E2	02
6	-	11	77	24	D7	E7	F7	36	1E	C0	0C	A0	FD	D4	98	B9	4B
7	-	87	BB	2B	45	3B	D3	EC	5F	7E	5C	51	05	2F	AF	C7	FA
8	-	78	9A	C4	89	43	F1	4E	F2	AA	26	8F	72	E8	AD	08	BF
9	-	61	70	FF	AC	6E	9B	68	16	8B	9F	E9	D9	7B	01	A6	CC
A	-	CD	22	6C	6F	6B	B1	00	83	CE	25	F3	81	A2	0E	1A	5E

B	-	07	7F	10	21	A3	84	0D	86	B5	3D	67	E3	D5	DB	EF	DC
C	-	D8	AB	59	D6	48	B7	34	E5	20	FC	A5	6D	56	41	F5	76
D	-	95	F0	9C	3F	35	97	53	FB	58	C2	D0	19	F9	8C	69	F6
E	-	57	AE	B4	14	79	52	5A	12	4D	4F	A9	E6	09	0F	62	A7
F	-	B2	06	74	90	CA	40	C1	50	13	64	91	6A	C6	B0	D2	8E

Direct SBox3

*	-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	D9	EE	83	B5	F4	02	EF	64	8E	4D	34	48	C2	29	C6	90
1	-	B3	9F	52	22	2F	E7	D0	76	95	8D	A1	2B	56	D7	7D	1C
2	-	2D	9A	3B	12	DD	00	24	A2	63	11	07	94	5D	F6	0E	7F
3	-	FF	5E	F3	65	E5	F1	A0	93	1E	BC	DE	A9	8B	F5	FA	B2
4	-	62	7E	B9	57	69	4C	FD	43	1A	08	35	05	E6	88	A5	44
5	-	45	01	BD	5B	B6	CC	BE	D3	9B	9E	8F	40	32	C3	8A	3E
6	-	0B	58	DB	99	0D	E1	87	B8	06	0F	0C	66	A4	FE	3D	10
7	-	FB	BB	6B	53	5A	C1	20	42	31	7C	CF	E0	89	E2	6C	09
8	-	04	17	CB	C0	E9	AC	5F	4E	81	8C	13	BA	0A	CE	55	23
9	-	38	4B	F0	79	6E	21	B7	82	46	D1	71	BF	26	86	D6	2E
A	-	97	C9	74	A6	2A	98	59	DA	AF	78	92	28	6A	6D	1D	4F
B	-	F8	61	7A	60	F2	6F	15	C4	ED	16	D4	EA	70	CD	EB	DC
C	-	B0	77	19	3A	D8	5C	F9	27	72	50	C5	3C	37	E3	A8	AA
D	-	F7	2C	73	1F	33	75	C7	68	67	36	4A	96	AB	EC	FC	1B
E	-	C8	7B	E8	A3	80	B4	9C	AE	18	41	D5	E4	25	51	14	49
F	-	AD	3F	CA	91	D2	A7	84	9D	30	DF	85	47	03	39	B1	54

Inverse SBox3

*	-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	-	25	51	05	FC	80	4B	68	2A	49	7F	8C	60	6A	64	2E	69
1	-	6F	29	23	8A	EE	B6	B9	81	E8	C2	48	DF	1F	AE	38	D3
2	-	76	95	13	8F	26	EC	9C	C7	AB	0D	A4	1B	D1	20	9F	14
3	-	F8	78	5C	D4	0A	4A	D9	CC	90	FD	C3	22	CB	6E	5F	F1

4	-	5B	E9	77	47	4F	50	98	FB	0B	EF	DA	91	45	09	87	AF
5	-	C9	ED	12	73	FF	8E	1C	43	61	A6	74	53	C5	2C	31	86
6	-	B3	B1	40	28	07	33	6B	D8	D7	44	AC	72	7E	AD	94	B5
7	-	BC	9A	C8	D2	A2	D5	17	C1	A9	93	B2	E1	79	1E	41	2F
8	-	E4	88	97	02	F6	FA	9D	66	4D	7C	5E	3C	89	19	08	5A
9	-	0F	F3	AA	37	2B	18	DB	A0	A5	63	21	58	E6	F7	59	11
A	-	36	1A	27	E3	6C	4E	A3	F5	CE	3B	CF	DC	85	F0	E7	A8
B	-	C0	FE	3F	10	E5	03	54	96	67	42	8B	71	39	52	56	9B
C	-	83	75	0C	5D	B7	CA	0E	D6	E0	A1	F2	82	55	BD	8D	7A
D	-	16	99	F4	57	BA	EA	9E	1D	C4	00	A7	62	BF	24	3A	F9
E	-	7B	65	7D	CD	EB	34	4C	15	E2	84	BB	BE	DD	B8	01	06
F	-	92	35	B4	32	04	3D	2D	D0	B0	C6	3E	70	DE	46	6D	30

APPENDIX B – ENCRYPT-DIEHARDER.C CODE TO GENERATE PSEUDORANDOM SEQUENCE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "encrypt.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, 0x11111111 );
      encryptBlock( &flexaeadv1, state);
      fwrite(state, 1, flexaeadv1.nBytes, stdout);
   }
   free(state);
}

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

REFERENCES

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

[2] BERNSTEIN, D. J. Cryptographic competitions. URL: < https://competitions.cr.yp.to> Access Date: Feb 28th 2019.

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

[4] 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.

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

[6] 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.

[7] EICHLSEDER, M. Posting on the NIST LWC mailing list, 2019. URL:< https://groups.google.com/a/list.nist.gov/forum/#!topic/lwc-forum/SgmvFLzFQNI>. Access Date: Jul 21st 2019.

[8] EICHLSEDER, M.; KALES, D.; SCHOFNEGGER, M. Forgery Attacks on FlexAE and FlexAEAD. IACR Cryptology ePrint Archive, Report 2019/679, 2019. URL:<https://eprint.iacr.org/2019/679>. Access Date: Jul 21st 2019.

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

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

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

[12] MÈGE, A.: OFFICIAL COMMENT: FlexAEAD. Posting on the NIST LWC mailing list, 2019. URL:<https://groups.google.com/a/list.nist.gov/forum/#!topic/lwc-forum/DPQVEJ5oBeU> . Access Date: Jul 21st 2019.

[13] NASCIMENTO, E.M.; XEXÉO, J.A.M. FlexAEAD - A Lightweight Cipher with Integrated Authentication. Round 1 submission to NIST lightweight cryptography Standardization process, 2019. URL: <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/FlexAEAD-spec.pdf>. Access Date: Jul 21st 2019.

[14] 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.

[15] 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.

[16] 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.

[17] NIST - NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process, 2018. URL:< https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf>. Access Date: Oct 21st 2019.

[18] RAHMAN, M.; Saha, D.; Paul, G. Posting on the NIST LWC mailing list, 2019. URL:< https://groups.google.com/a/list.nist.gov/forum/#!topic/lwc-forum/VLWtGnJStew> . Access Date: Jul 21st 2019.

[19] RAHMAN, M.; Saha, D.; Paul, G. Iterated Truncated Differential for Internal Keyed Permutation of FlexAEAD. IACR Cryptology ePrint Archive, Report 2019/539, 2019. URL:<https://eprint.iacr.org/2019/539> . Access Date: Jul 21st 2019.

Eduardo Marsola do Nascimento received his MSc. in System and Computing from Instituto Militar de Engenharia - IME (2017), MBA in Business Management from Fundação Getúlio Vargas - FGV (2011), Undergraduate degree in Computer Engineering from Universidade São Judas Tadeu - USJT (2001). He has worked at the private sector to several multinational companies, mostly in it infrastructure and it security teams. Actually he is a telecommunication engineer at Petrobras - Petróleo Brasileiro S/A working at cybersecurity department.

José Antonio Moreira Xexéo possui graduação em Engenharia da Comunicações pelo Instituto Militar de Engenharia (1972), mestrado em Sistemas e Computação pelo Instituto Militar de Engenharia (1983) e doutorado em Engenharia de Sistemas e Computação pela Universidade Federal do Rio de Janeiro (2001). É avaliador institucional e de curso de graduação do INEP. Atualmente é professor nos cursos de graduação em Administração e Engenharia de Produção da Universidade Veiga de Almeida e dos cursos de graduação e mestrado em Engenharia de Computação do Instituto Militar de Engenharia (IME), onde realiza pesquisa em criptologia. Liderou a equipe que projetou e implantou no IME, em 1985, o primeiro curso de Engenharia de Computação do Brasil, do qual foi o seu primeiro coordenador. Atuou por mais de 10 anos na área de desenvolvimento tecnológico na área industrial de informática. Tem experiência acadêmica em engenharia e ciência da computação, principalmente criptologia. Atua no Ensino Superior como professor e coordenador de cursos há quase 40 anos**.**