Name of Submission:

**FlexAEAD - A Lightweight Cipher with Integrated Authentication**

Name of Submitters:

**Eduardo Marsola do Nascimento¹, José Antônio Moreira Xexéº²**

---

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 FlexAEAD 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](image1)

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 ($b[0], b[1], ..., b[2^{nb+1} - 1]$) and reordered as ($b[0], b[2^{nb}], b[1], b[2^{nb+1}], ..., b[2^{nb} - 1], b[2^{nb+1} - 1]$).

![Figure 2. The BlockShuffle Layer](image2)

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.

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.

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 $128 \times 2^x$ bits, where $x \geq 0$. 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 ($S_0, S_1, ... S_{n+m}$). This sequence is the same size of the associated data plus the message to be sent. It is generate by applying $PF_{K_3}$ 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 size 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.

![Diagram of subkeys generation](image)

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

To hash the associate data, first the associated data is divided in $n$ blocks ($AD_0, AD_1, ... AD_n$). 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 $PF_{K_2}$ to generate a intermediate state block ($st_x$). The process that each associated data block goes though is ($AD_x \rightarrow$...
To cipher the plain text message, it is broken into \( m \) plaintext blocks \((P_0P_1 \ldots P_m)\). The last block is padded with \((10^{p_b-1})\), where \( p_b \) 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 through 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 \((0101 \ldots 01)\). If there was no padding it is XORed with the bit sequence \((0101 \ldots 01)\). After 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 \((MSB_{Tlen})\).

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 \((10^{p_b+1})\) 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 ...01) then submitted to \((PF_{K_0})\) 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 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 difficulty of an attack based on differential cryptanalysis is \(N_D \approx \frac{1}{p_D}\) (Heys, 2001).

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Rounds (r-1)</th>
<th>Active SBoxes</th>
<th>(p_D)</th>
<th>(N_D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>14</td>
<td>28</td>
<td>(2^{-168})</td>
<td>(2^{168})</td>
</tr>
<tr>
<td>128</td>
<td>17</td>
<td>34</td>
<td>(2^{-204})</td>
<td>(2^{204})</td>
</tr>
<tr>
<td>256</td>
<td>20</td>
<td>40</td>
<td>(2^{-240})</td>
<td>(2^{240})</td>
</tr>
</tbody>
</table>

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 \(\varepsilon = \frac{16}{256} = 2^{-4}\).

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 \( (\epsilon) \) 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): \( \epsilon = p - 0.5 \).

**Table 2. Difficult to perform a linear cryptanalysis attack**

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Rounds ( (r) )</th>
<th>Active SBox</th>
<th>Maximum Bias</th>
<th>( N_L = \frac{1}{\epsilon^2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>15</td>
<td>45</td>
<td>( \epsilon = 2^{-136} )</td>
<td>( N_L = 2^{1272} )</td>
</tr>
<tr>
<td>128</td>
<td>18</td>
<td>54</td>
<td>( \epsilon = 2^{-163} )</td>
<td>( N_L = 2^{326} )</td>
</tr>
<tr>
<td>256</td>
<td>21</td>
<td>63</td>
<td>( \epsilon = 2^{-190} )</td>
<td>( N_L = 2^{380} )</td>
</tr>
</tbody>
</table>

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 $5x$ and another execution of the $PF_x$ 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 deoxys128v141 (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>
<thead>
<tr>
<th>Variant</th>
<th>Parameters sizes (in bits)</th>
<th>Cryptanalysis difficulty</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Key</td>
<td>Block</td>
</tr>
<tr>
<td>FlexAEAD128b064</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>FlexAEAD128b128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>FlexAEAD256b256</td>
<td>256</td>
<td>256</td>
</tr>
</tbody>
</table>

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. Cryptographic competitions. URL: <https://competitions.cr.yp.to>


### APPENDICE A – Direct and Inverse AES SBox

#### Table 3. Direct AES SBox

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x1</td>
<td>0x2</td>
<td>0x3</td>
<td>0x4</td>
<td>0x5</td>
</tr>
<tr>
<td>0x6</td>
<td>0x7</td>
<td>0x8</td>
<td>0x9</td>
<td>0xA</td>
<td>0xB</td>
</tr>
<tr>
<td>0xC</td>
<td>0xD</td>
<td>0xE</td>
<td>0xF</td>
<td>0x0</td>
<td>0x1</td>
</tr>
</tbody>
</table>

#### Table 4. Reverse AES SBox

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x1</td>
<td>0x2</td>
<td>0x3</td>
<td>0x4</td>
<td>0x5</td>
</tr>
<tr>
<td>0x6</td>
<td>0x7</td>
<td>0x8</td>
<td>0x9</td>
<td>0xA</td>
<td>0xB</td>
</tr>
<tr>
<td>0xC</td>
<td>0xD</td>
<td>0xE</td>
<td>0xF</td>
<td>0x0</td>
<td>0x1</td>
</tr>
</tbody>
</table>
APPENDICE B – encrypt-dieharder.c code to generate pseudorandom sequence

```c
#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   |n    | tsamples |psamples|  p-value |Assessment
#=============================================================================#
diehard_birthdays|   0|       100|     100|0.53243263|  PASSED
diehard_operm5| 1000000|     100|0.92541253|  PASSED
diehard_rank_6x8|  400000|     100|0.15594265|  PASSED
diehard_bitstream|  2097152|     100|0.34139275|  PASSED
diehard_opso|  2097152|     100|0.32834173|  PASSED
diehard_opso|  2097152|     100|0.91056284|  PASSED
diehard_dna|  2097152|     100|0.38464814|  PASSED
diehard_count_1s_str|  256000|     100|0.34100720|  PASSED
diehard_count_1s_byt|  256000|     100|0.96884054|  PASSED
diehard_parking_lot|  120000|     100|0.96913730|  PASSED
diehard_2dsphere|   2|       8000|     100|0.20717814|  PASSED
diehard_3dsphere|   3|       4000|     100|0.09572503|  PASSED
```
<table>
<thead>
<tr>
<th>Test</th>
<th>Value</th>
<th>P-value</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diehard Squeeze</td>
<td>0</td>
<td>10000</td>
<td>0.49830589</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>0</td>
<td>100</td>
<td>0.42558220</td>
</tr>
<tr>
<td>Diehard Runs</td>
<td>0</td>
<td>100000</td>
<td>0.03886906</td>
</tr>
<tr>
<td>Diehard Runs</td>
<td>0</td>
<td>20000</td>
<td>0.11980794</td>
</tr>
<tr>
<td>Diehard Craps</td>
<td>0</td>
<td>20000</td>
<td>0.71676496</td>
</tr>
<tr>
<td>Marsaglia Tsang-GCD</td>
<td>0</td>
<td>10000000</td>
<td>0.54813906</td>
</tr>
<tr>
<td>Marsaglia Tsang-GCD</td>
<td>0</td>
<td>10000000</td>
<td>0.96626464</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>0</td>
<td>10000</td>
<td>0.4298167</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>1</td>
<td>10000</td>
<td>0.77127272</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>2</td>
<td>10000</td>
<td>0.98176496</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>3</td>
<td>10000</td>
<td>0.69443393</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>4</td>
<td>10000</td>
<td>0.68843008</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>5</td>
<td>10000</td>
<td>0.43152701</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>6</td>
<td>10000</td>
<td>0.63596468</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>7</td>
<td>10000</td>
<td>0.24298167</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>8</td>
<td>10000</td>
<td>0.25485876</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>9</td>
<td>10000</td>
<td>0.38119541</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>10</td>
<td>10000</td>
<td>0.41830759</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>11</td>
<td>10000</td>
<td>0.44154753</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>12</td>
<td>10000</td>
<td>0.04949517</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>13</td>
<td>10000</td>
<td>0.87776520</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>14</td>
<td>10000</td>
<td>0.88271258</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>15</td>
<td>10000</td>
<td>0.93208301</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>16</td>
<td>10000</td>
<td>0.19714967</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>17</td>
<td>10000</td>
<td>0.36558756</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>18</td>
<td>10000</td>
<td>0.84095563</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>19</td>
<td>10000</td>
<td>0.24424891</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>20</td>
<td>10000</td>
<td>0.88271258</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>21</td>
<td>10000</td>
<td>0.93208301</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>22</td>
<td>10000</td>
<td>0.19714967</td>
</tr>
<tr>
<td>Diehard Sums</td>
<td>23</td>
<td>10000</td>
<td>0.36558756</td>
</tr>
</tbody>
</table>
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|   100000|  1000|0.18623770|  PASSED
dab_bytedistrib|  0|  51200000|       1|0.71777771|  PASSED
dab_filltree|  32|  15000000|       1|0.17292794|  PASSED
dab_filltree2|  0|   5000000|       1|0.68458837|  PASSED
Preparing to run test 207.  ntuple = 0
dab_filltree|  32|  15000000|       1|0.35405515|  PASSED
dab_filltree2|  1|   5000000|       1|0.04958262|  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          #
#======================================================
#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          #
#======================================================
#test_name   |ntup| tsamples |psamples|  p-value |Assessment#
#sts_monobit|   1|    100000|     100|0.35268451|  PASSED