SETIT 2005
3rd International Conference: **S**ciences of **E**lectronic, **T**echnologies of **I**nformation and **T**elecommunications
March 27-31, 2005 – TUNISIA

A Proposal For A Key-Dependent AES

A. Fahmy*, M. Shaarawy**, K. El-Hadad***, G. Salama*** and K. Hassanain****

* *Faculty of computers and Information, Cairo University, Egypt*

** *Faculty of computers and Information, Helwan University, Egypt*
shaarawy194@yahoo.com

*** *Egyptian Armed Forces, Egypt*

****Technical Research Department, Egypt*
khass@idsc.net.eg

Abstract: At all times people have wished to have the possibility to communicate in secrecy so as to allow nobody to overhear their messages. A new AES-like cipher, **k**ey-dependent **A**dvanced **E**ncryption **S**tandard algorithm, KAES is proposed, to vanish any suspicion of a trapdoor being built into the cipher and to expand the key-space to strength it against known attacks. The cipher structure resembles the AES approved by **N**ational **I**nstitute for **S**tandardization and **T**echnology, NIST, providing the necessary confusion and diffusion.
Key words AES, Confusion, Diffusion, Key dependent S-Box

1. Introduction

In October 2000, after a four year effort to replace the aging DES, NIST announced the selection of Rijndael as the proposed AES (NIST 2004). Draft of the Federal Information Processing Standard (FIPS) for the AES was published in February 2001, Standardization of AES was approved after public review and comments, and published a final standard FIPS PUB-197 in December 2001. Standardization was effective in May 2002 (NIST 2004).

Rijndael submitted by Joan Daemen and Vincent Rijmen (Daemen 1998), is a symmetric key, iterated block cipher based on the arithmetic in the Galois Field of 2^8 elements – GF(2^8). The block size and the key size can be independently specified to 128, 192 or 256 bits (all nine combinations are possible). A data to be processed by Rijndael is partitioned into a rectangular array of bytes, called a state, The key is similarly pictured as a rectangular array with 4 rows and N_k columns. Where N_k is equal to the key size divided by 32. Rijndaels round function operates on a state Nr times, where N_r is equal to the number of rounds that can be 10, 12 or 14 rounds, depending on N_b and N_k, where N_b is equal to the block size divided by 32. Fig.1. depicts a state, with 4 rows and 4 columns.

Rijndael round is composed of 4 transformations:

1. ByteSub (Sbox substitution) provides non-linearity and confusion.
2. Shiftrow (rotations) provides inter-column diffusion.
3. MixColumn (linear combination) provides inter-Byte diffusion.
4. AddRoundKey (round key bytes XOR into each byte of the state), provides confusion.

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

Fig.1. State 4x4.

Their operations are byte-oriented. Decryption is applying the operations in a reverse order with respect to the order of encryption. For more details refer to (Daemen 1998).

This paper introduces a new, key-dependent Advanced Encryption standard algorithm, **KAES,** to ensure that no trapdoor is present in the cipher and to expand the key-space to slow down attacks.
The paper is organized as follows: Section 2, presents the proposed KAES. Section 3 explains the evaluation

criteria. Section 4 discusses the experimental results. Section 5 summaries and concludes the paper. References are given in Section 6.

2. KAES

KAES is block cipher in which the block length and the key length are specified according to AES specification: three key length alternatives 128, 192, or 256 bits and block length of 128 bits. We assume a key length of 128 bits, which is likely to be the one most commonly implemented. Fig.2 shows the overall structure of KAES.

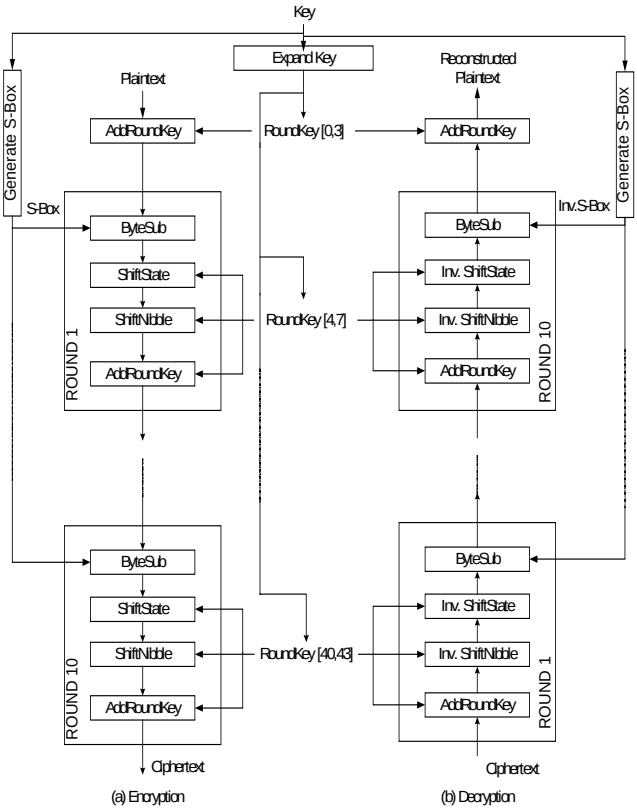

Fig.2. KAES Encryption and Decryption.

The input to the encryption and decryption algorithms is a single 128-bit block, this block is depicted as a square matrix of bytes. This block is copied into the state array as shown in Fig.1, which is modified at each stage of encryption or decryption. After the final stage, state is copied to an output matrix. Similarly, 128-bit key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words: each word is four bytes and the total key schedule is 44 words for the 128-bit key, a round key similar to a state is depicted in Fig.3.

The round function resembles that of AES, it is also composed of 4 stages namely: Byte substitution, Shift State, shift Nibble and Add round key. The encryption and decryption process resembles that of AES with the same number of rounds, data and key size. We describe the forward (encryption) algorithm, the inverse (decryption) algorithm and the rationale for each stage. This is followed by a description of key expansion and generation of shift offset-matrix.

r_0	r_4	r_8	r_{12}
r_1	r_5	r_9	r_{13}
r_2	r_6	r_{10}	r_{14}
r_3	r_7	r_{11}	r_{15}

Fig.3. Round Key.

2.1 Substitute Bytes Transformation

Uses key dependent S-Box to perform a byte-by-byte substitution of the state. The forward substitute byte transformation, called *SubBytes*, is a simple lookup table. KAES defines a 16x16 matrix of byte values, called an S-Box, that contains a permutation of all possible 256 8-bit values (elements of GF(2^8) finite field) with the key used for construction. Each individual byte of the state is mapped into a new byte in the following way: The leftmost 4-bits of the byte (higher nibble) are used as a column value and the rightmost 4-bits (lower nibble) are used as a row value. These row and column values serve as indexes into S-Box to select a unique 8-bit output value.

The S-Box is constructed in the following fashion:

1- Initialize the S-Box with the Byte values in ascending sequence column by column. The first column contains 0x00, 0x01,……,0x0F; the second column contains 0x10, 0x11,… etc; and so on. Thus, the value of the Byte at column x and row y is [xy].

2- For each byte value of the key, k_i (for 0 =< i <= key length), for example, if the key length is 16 Byte, the first byte k_1, then k_2 and so on. Examine the value of k_i, if (k_i mod 2) equals zero, run a pseudorandom generator for the value of k_i; otherwise run another, also for the value of k_i. Two linear congruence pseudorandom generators are used, called rand1 and rand2, that makes use of the linear congruence parameters (Michael 2001) for ISO-C Standard and GNU-C respectively.

3- The last run value of the selected pseudorandom generator, r, is added to the mean of the key, to introduce a loop counter value for the swapping loop

$$loopcounter = [mean(key) + r + 1] \qquad (1)$$

4- Again, use rand1 and rand2 to generate two byte values that serve as an indexes into S-Box to select two bytes to be swapped together. This operation continues until the loop counter ends.

5- Repeat steps 2, 3, and 5 until all the key byte values k_i (for 0 =< i <= key length) has been taken.

The inverse substitute byte transformation, called *InvSubBytes*, makes use of the inverse S-Box The inverse S-Box is constructed by determining a substitution pair and replacing it with it's inverse.

The S-Box is designed that has a low correlation between input bits and output bits The S-Box has a property that there is no "**fixed points**"

$$[\text{S-Box}(a) = a] \quad (2)$$

and no "**opposite fixed points**"

$$[\text{S-Box}(a) = \overline{a}\,] \quad (3)$$

where $\overline{a}$ is bit-wise complement of a. It is invertible, that is

$$\text{Inv-S-Box}\,[\text{S-Box}(a)] = a \quad (4)$$

However the S-Box is not self inverse, in the sense that it is not true that

$$\text{S-Box}(a) = \text{Inv-S-Box}(a) \quad (5)$$

2.2. Shift State Transformation

Permute the state either column-wise or row-wise interchangeable dependent on round key values. The forward shift state transformation, called *ShiftState*, is composed of two transformations:

ShiftRow transformation, that performs circular left shift on individual rows of the state, according to offset_1.

ShiftColumn transformation, that performs circular up shift on individual columns of the state, according to offset_2.

offset_1 and offset_2 are selected from an offset_matrix; a linear array that contains combinations of different offset values. The selection depends on the round key byte values r_1 and r_2 respectively, in the following way: the round key r_i (for i =1,2) is looked up the S-Box, the column index for r_i serve as an index into the offset_matrix and the corresponding value is copied into offset_i. The order in which shift row or shift column are executed depends on the round key byte value r_0, that is if (r_0 mod 2) equals to zero the order is shift row followed by shift column, otherwise shift column followed by shift row.

The inverse shift state transformation, called *InvShiftState*, performs the *shift row* and *shift column* in opposite direction right and down respectively. Also their order are reversed, i.e. if (r_0 mod 2) equals to zero the order becomes shift column followed by shift row, instead of shift row followed shift column as in *ShiftState*. Shift row (a part of shift state) moves an individual Byte from one column to another, which is a linear distance of a multiple of 4 Bytes, and shift column (the other part of shift state) moves an individual Byte across a column (from one row to another), which is a linear displacement within a row

2.3. Shift Nibble Transformation

Shifts the higher nibbles or lower nibbles of the state columns dependent on round key values. The forward nibble shift transformation, called *ShiftNibble*, operates on each column individually. Each Byte is composed of two nibbles, higher nibble (leftmost 4-bits of a Byte) and lower nibble (right most 4-bits of a Byte). Each byte of a column is mapped into a new value that is a function of the next Byte in the column. The transformation can be defined by the following operation on a state. For each column, C_j (where, 0 =< j <= number of columns). Either higher nibbles or lower nibbles are circular shifted up one position dependent on the value of the Round key. i.e. if ($r_{3,j}$ mod 2), i.e. elements of the 3rd row of round key, equals to zero then lower nibbles of column, C_j are shifted up one position, otherwise higher nibbles of column, C_j are shifted up one position. The inverse shift nibble transformation, called *InvShiftNibble*, performs circular down shift by one position, i.e. in opposite direction of the forward transformation. The *ShiftNibble* transformation combined with the *ShiftState* transformation ensures that after a few rounds, all output bits depend on all input bits.

2.4. Add Round Key Transformation

A simple bit-wise XOR operation on a state with a portion of the expanded key, round key. i.e. 128-bits of a state are bit-wise XORed with 128-bits of a round key. The add round key transformation, is called *AddRoundKey*,

The operation is viewed as a column wise operation between the 4-Bytes of a state column and one word of the round key; it can also be viewed as a Byte-level operation. The inverse AddRoundKey transformation is identical to the forward AddRoundKey transformation, because the XOR operation is it's own inverse.

$$state \oplus Roundkey = state' \quad (6)$$

$$state' \oplus Roundkey = state \quad (7)$$

The Add Round Key transformation is a simple as possible and effects every bit of a state. The complexity of the round key expansion, plus the complexity of the other stages of KAES, ensure security.

2.5. KAES Key Expansion

The KAES key expansion algorithm, takes as an input a four word (16 Bytes) key, produces a linear array of forty four words (176 Bytes) keys. This is sufficient to provide a four word round key for the initial *AddRoundKey* stage and each of the 10 rounds of cipher. The following pseudo code describes the expansion.

The key is utilized to construct a Key_Box, similar to S_Box. Four bytes from the Key_Box is copied into a word, w to fill the first word of the expanded key. for a word whose position in the w array a multiple of 4, a new Key_Box is constructed

using previous 4 words of the array w to construct a key that is exploited in the construction of the Key_Box. The remainder of the expanded key is filled in a similar way.

```
KeyExpansion (Byte key[16] , Word w[44])
{ Byte Key_Box = [16,16] , k[4];

 for (r = 0; r < 44; r++)
 {
   if (r mod 4 = 0)   Key_Box = gen_S_Box(key);
   for(i=0;i<4;i++)
     for(j=0;j<4;j++)
        k[j] = Key_Box[(r mod 16,4*i+j)];
     w[4*r+i] = (k[0], k[1], k[2], k[3]);
     if (r mod 4 = 0)
       key = (w[4*r],w[4*r+1],w[4*r+2],w[4*r+3]);
 }
}
```

2.6. Shift Offset Matrix Generation

The shift offset matrix algorithm, takes as an input a 16-Byte key and produces a linear array of 16 words called offset_matrix. Which is sufficient to provide offsets for both Shift Row and Shift Column of the *ShiftState* transformation. The length of the offset_matrix, 16, symbolizes the number of columns of S_Box, since column index serve as an index into the offset_matrix.
The offset_matrix is constructed in the following fashion:

1. The mean of the key is employed to reset the pseudorandom generator to it's J^{th} state, i.e. start generation from position J that is equal to the value of the mean of the key. The pseudorandom generator provides numbers that are uniformly distributed in the interval [0,1]. To match the offset shifts, the generated numbers are multiplied by 4 and rounded to the nearest integer to form the word offset [0 1 2 3], where 0 implies no shift, 1 implies a shift by one position and so on. The direction of the shift depends on whether it's *ShiftState* or *InvShiftState*.

2. The word offset [0 1 2 3] is pseudorandom permuted and copied in offset_matrix. The operation continues until 16 words have been generated and filled into the offset_matrix.

3. The offset_matrix is circularly down shifted by the number of columns of the state.

2.7. KAES vs. AES

The main differences between AES and KAES can be summarized in Table.1. Where the key and round key are applied in each stage, transformation as depicted in Fig.2.

Table.1. Differences between KAES and AES

	AES	KAES
Block Length	128-bit	same
Key Length	128-bit,192-bit and 256-bit	Same
Number of Rounds	For key length 128-bit. 10 Rounds	Same
Round Function	# Composed of 4 transformations, namely: -ByteSub -Shift Row -Mix Column -AddRoundKey. # For last round Mix Column is eliminated	# Composed of 4 ransformations, namely: -ByteSub -Shift State -Shift Nibble - AddRoundKey. # Same for all rounds
S-BOX	fixed	key dependent
Key Expansion	Utilize the fixed S-Box	Generate a new Key-Box
Round Transformation	Independent on the key, except for Add Round Key.	All transformations utilize the key.
Shift Offset	Fixed [0 1 2 3]	Reliant on the key

3. Evaluation Criteria

Various tests can be applied to sequences of bytes for evaluating pseudorandom number generators for encryption and compression algorithms (ENT 2004). To facilitate interpretation of the experimental results, a brief description is given, to make the analysis of the tests' output understandable

(a) Entropy
The information density expressed as a number of bits per byte. Extremely dense information indicate that information is essentially random. Hence optimal compression is unlikely to reduce it's size.

(b) Optimal "Best" Compression
Reflects compressibility and is computed based on entropy encoding.

(c) Chi-square Distribution
The chi-square (χ^2) distribution is calculated for a stream of bytes, expressed as a percentage. The percentage can be interpreted as the degree to which a sequence under investigation is suspect of being "NON-RANDOM". If the percentage is greater than 99% or less than 1%, the sequence is almost certainly "NOT-RANDOM". If the percentage is between 99% and 95% or between 1% and 5%, the sequence is "SUSPECT". Percentages

between 90% and 95% or between 5% and 10% indicate that the sequence is "ALMOST SUSPECT" . Otherwise, the sequence is random.

(d) Arithmetic Mean Value

Calculate the mean of a sequence. If a sequence, close to random, it's mean should be about 127.5. If the mean departs from this value, the values are consistently high or low.

(e) Monte Carlo value for π

Each successive sequence of six bytes is used as 24-bit x and y coordinates within a square. If the distance of a randomly generated point is less than the radius of a circle inscribed within the square, the six byte sequence is considered a "hit". The percentage of hits can be used to calculate the value of π. If the computed value approaches the correct value of π, the sequence is close to random.

(f) Serial Correlation Coefficient

The quantity measures the extent to which each byte in a sequence depends upon the previous byte. If the value (which can be positive or negative) close to zero, the sequence is random (totally uncorrelated). Otherwise serial correlation coefficient will be greater than or equal 0.5.

4. Experimental Results

To simulate KAES, a MATLAB (Mathworks 2004) script was implemented for both AES and KAES. The key was fixed for both algorithms. A number of files were encrypted and decrypted, including images and audio. For evaluation a program, named "**ENT"** (ENT 2004) for testing pseudo number sequences was used in comparison between KAES and AES as shown in Tables.2-3. To measure the performance of KAES, NIST statistical test suite for testing randomness (NIST 2001), completeness principle, Avalanche effect (Webster 1985) was implemented in MATLAB (Mathworks 2004).

Table 2 shows a comparison between AES and KAES for encrypting a text file of size 3228 Bytes. It can be observed that, the Arithmetic mean value of KAES reaches 127.5, thus the data is close to random. In addition to the information is more dense indicating that the information is essentially random.

Table.3. illustrates the comparison of encrypting an audio file of size 11,532 Bytes. χ^2 distribution shows that both sequences are random. The main feature is that KAES processing time is much more faster than AES.

For an image of size 56296 Bytes, that contains a wildly predictable repeated data. Table.4. shows that the calculated χ^2 distribution for both algorithms is almost certainly not random, and the optimum compression indicates that compressibility may be achieved. Again the main feature is processing time.

Table.2.Comparison Between AES & KAES for a text file

Measure	Plaintext	AES	KAES
Entropy (bits/byte)	4.7854	7.9362	7.9443
Optimum compression	40%	0%	0%
χ^2 distribution	0.01%	25%	50%
Arithmetic mean value	87.996	123.78	128.87
Monte Carlo value for Pi (3.1417)	4.0	3.2416	3.0631
Serial correlation coefficient	0.1808	0.0024	-0.0023
Processing time		20.75s	20.29s

Table3.Comparison Between AES & KAES for an audio file

Measure	Plaintext	AES	KAES
Entropy (Bits /byte)	4.4218	7.9840	7.9820
Optimum compression	44%	0%	0%
χ^2 distribution	0.01%	50%	11%
Arithmetic mean value	126.200	128.69	126.07
Monte Carlo value for Pi	3.99	3.1175	3.2008
Serial correlation coefficient	-0.0820	0.0038	0.0043
Processing time		74.438s	23.547s

Table4.Comparison Between AES & KAES for an image

	Plaintext	AES	KAES
Entropy (Bits / byte)	3.67435	7.2802	7.1697
Optimum compression	54%	8%	10%
χ^2 distribution	0.01%	0.01%	0.01%
Arithmetic mean value	182.397	126.66	127.28
Monte Carlo value for Pi (3.1416)	0.5687	3.1346	3.3158
Serial correlation coefficient	0.6128	-0.075	0.0217
Processing time		362.73s	39.48s

Several tests have been conducted to observe the performance of KAES. Below experimental results

achieved for a plaintext of 16 bytes all zeros, and only one bit complemented at a time, a sample of the plaintext and its corresponding ciphertext is shown in Table.5.

Table.5. Plaintext & Ciphertext sample

Plaintext: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Ciphertext: 50 94 43 9C F4 BE 6E F4 9C 67 1E E4 54 33 4B 95
Plaintext: 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Ciphertext: F9 BA D3 C6 E6 C4 A8 1A 8E 4A AA 7D A0 B5 F0 9E
Plaintext: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Ciphertext: 10 18 0C 6E A3 4A EE 28 BA 7B 25 1C 2F A6 68 0C

Fig.4. shows the Pearson's correlation coefficient of plaintext and its corresponding ciphertext, that provides a measure of how the two affect one another, i.e. how much one of them depends on the other.

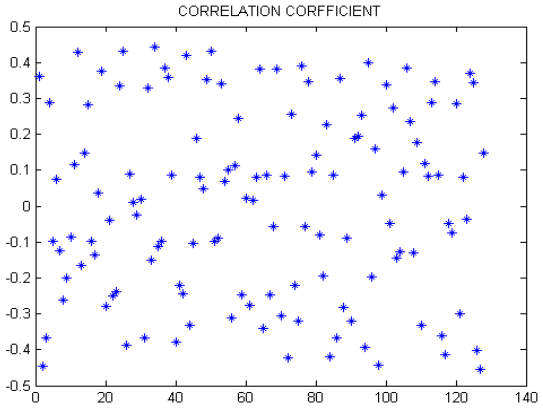

Fig.4.Correlation Coefficient.

It could be noticed that, the correlation coefficient values indicate a degree of linear dependence between the plaintext bits and the ciphertext bits as shown in Fig.4.

To measure confusion and diffusion, we applied avalanche effect and completeness principle on the same plaintext and it's ciphertext stated above. It can be observed that complementing one bit of the plaintext results in an average change of the ciphertext bits as depicted in Fig.5. Also the number of avalanche vectors that had more than one half of its bits changed was 73 and is more than one half 128-bit pairs.

Fig.6. plots the P-value of the frequency test of both the plaintext and cipher, almost the P-value of the ciphertext is more than 0.01, we conclude that the sequence is random

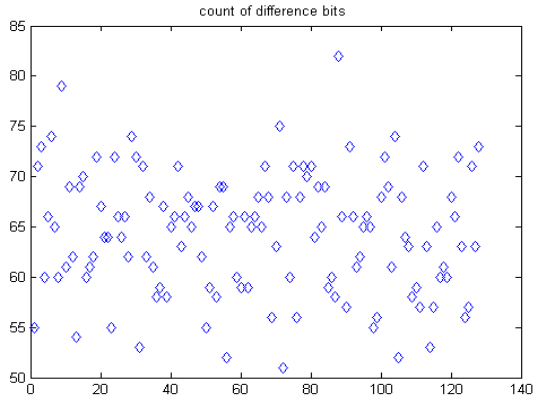

Fig.5.Number of ciphertext bits changed when complementing one bit of plaintext

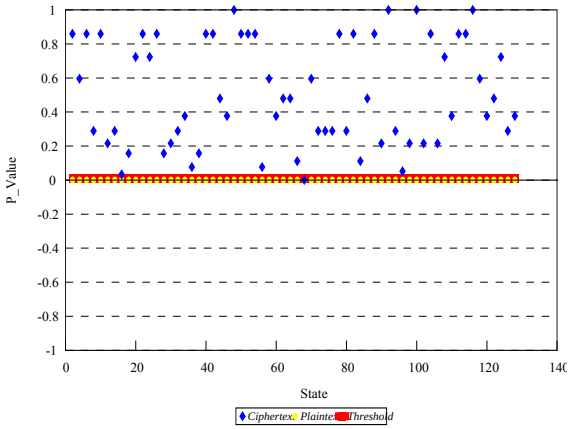

Fig.6.Frequency Test

The P-value for frequency test within a block of size 12-bits indicates that the sequence of the ciphertext is random as illustrated in Fig.7.

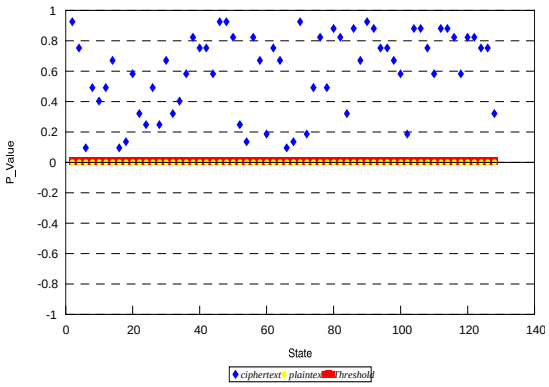

Fig.7.Frequency Test within a Block

Runs test focus on the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits, i.e. determine the oscillation between zeros and ones is too fast or too slow, as shown in Fig.8. Where P-value of ciphertext exceeds the threshold value 0.01, and thus the sequence is random.

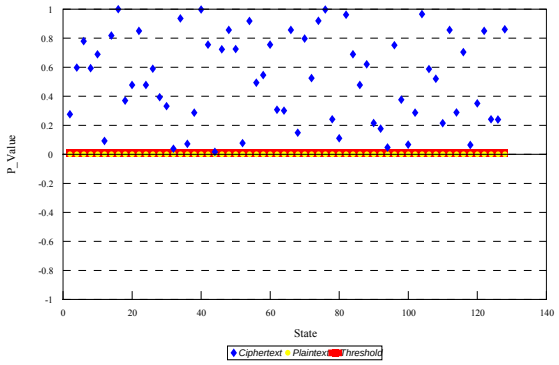

Fig.8.Runs Test

The longest run of ones in a block of size 16 bit determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence. Fig.9. depicts the P-value which is greater than 0.01, thus we can conclude that the sequence is random.

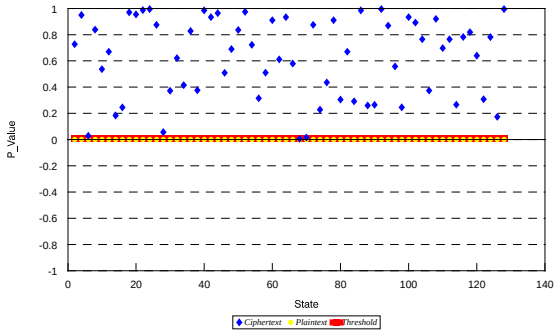

Fig.9.Longest Run of Ones within a Block

Conclusion

KAES doesn't contradict the security of the AES algorithm. We tried to keep all the mathematical criteria for AES without change. We improved the security of AES by employing the key to be the main parameter of the algorithm.

References

(Daemen 1998) Joan Daemen, Vincent Rijmen, "AES Proposal: Rijndael", Banksys/Katholieke Universiteit Leuven, Belgium, AES submission, June 1998.

http://www.esat.kuleuven.ac.be/~rijmen/rijndael/

(ENT 2004) A Pseudorandom Number Sequence Test Program. http://www.fourmilab.ch/random.

(Mathworks 2004) MATLAB computational environment, www.mathworks.com

(Michael 2001) Michael Welschenbach, "Cryptography in C and C++", Apress, ISBN: 1-893115-66-6, 2001

(NIST 2001) A Statistical Test Suite for Random and Pseudorandom Generators for Cryptographic Applications", NIST Special Publication 800-22, 2001.

(NIST 2004) NIST, Advanced Encryption Standard. http://www.nist.gov/aes/

(Webster 1985) A. F. Webster and S. E. Tavares, "On the Design of S-Boxes" in Advances Cryptology – CRYPTO'85, Volume 219, Lecture Notes in Computer Science, pages 523-534, Springer-Verlag, Berlin, 1986.