Journal of Computer Science 2 (5): 419-421, 2006
ISSN 1549-3636
© 2006 Science Publications

Key Based Random Permutation (KBRP)

Shakir M. Hussain[1] and Naim M. Ajlouni
Applied Science University[1], Jordan
Amman Arab University for Graduate Studies, Jordan

Abstract: This study introduces a method for generating a particular permutation ***P*** of a given size ***N*** out of ***N!*** permutations from a given key. This method computes a unique permutation for a specific size since it takes the same key; therefore, the same permutation can be computed each time the same key and size are applied. The name of random permutation comes from the fact that the probability of getting this permutation is **1** out of ***N!*** possible permutations. Beside that, the permutation can not be guessed because of its generating method that is depending completely on a given key and size.

Key words: Random permutation, indexing, block cipher, hashaing

INTRODUCTION

A permutation, also called an "arrangement number" or "order," is a rearrangement of the elements of an ordered list ***S*** into a one-to-one correspondence with *S* itself. The number of permutations on a set of *n* elements is given by ***n!*** (***n*** factorial)[1,2].

For example, there are 3***!*** = 3x2x1 = 6 permutations of {1,2,3}. These permutations are: {1,2,3} , {1,3,2}, {2,1,3} , {2,3,1} , {3,1,2}, and {3,2,1}.

A list of length *n* can be tested to see if it is a permutation of 1, ..., *n* using the command PermutationQ[list] in the Mathematica add-on package DiscreteMath`Combinatorica`[3] .

Sedgewick[4] summarizes a number of algorithms for generating permutations, and identifies the minimum change permutation algorithm of Heap[5] to be generally the fastest[6]. Ives[7] gave four new algorithms for permutation enumeration. Johnson[8] gave another method of enumerating permutations.

A random permutation is a permutation containing a fixed number ***n*** of a random selection from a given set of elements. There are two main algorithms for constructing random permutations. The first constructs a vector of random real numbers and uses them as keys to records containing the integers 1 to ***n***. The second starts with an arbitrary permutation and then exchanges the *i*th element with a randomly selected one from the first *i* elements for *i* = 1, ..., ***n***[6,9]. Schemes for generating permutations and for numbering permutation have been developed over the years[10-12]. Campbell[13] gave a simple numbering scheme for calculating the permutation. Hussain[14] gave a method for generating permutation from string of bits.

The proposed method: Key Based Random Permutation (**KBRP**) is a method that can generate one permutation of size n out of ***n!*** permutations. This permutation is generated from certain key (alphanumeric string) by considering all the elements of this given key in the generation process. The permutation is stored in one-dimensional array of size equal to the permutation size (***N***). The process involves three consecutive steps: init(), eliminate(), and fill().
First step, init(), is to initialize array of size n with elements from the given key, by taking the ASCII code of each element in the key and storing them in the array consecutively. To complete all elements of the array, we add elements to the array by adding two consecutive values of the array until all the elements of the array are set to values. Finally, all values are set to the range 1 to ***N*** by applying the mode operation.

The second step, eliminate(), is to get rid of repeated values by replacing them with value of zero and keep only one value out of these repeated values. Last step, fill(), is to replace all zero values with nonzero values in the range 1 to **N** which are not exist in the array. The resulted array now represents the permutation.

Step1: init()

Initialization step can be shown as follows:

```
Let
K: key (string of alphanumeric) of size S
P: array holds permutation with values 1 to N
N: array size
A[i] = K[i]          for i=1 to S
P[i] = P[i] + P[i+1]          for i=1 to S-1
P[S] = A[1]
While (S < N)
    j = S+1
    For( i = 1 to S-1 )
        For( k = i to S-1 && j ≤ N )
                P[i] = P[i] + P[k+1]
                j++
P[i] = P[i] MOD N          for i = 1 to N
```

Corresponding Author: Shakir M. Hussain, Applied Science University, Jordan

Step2: eliminate()

In this step, array **P** contains **N** values. Repetition for some values maybe exists; therefore, the repeated values are examined and replaced with zero. Only one value out of the repeated values is kept in **P**. now **P** has only distinct values in the range 1 to **N** and some zero values are appeared in **P**. Missing values in the range 1 to **N** that are not exist in **P** will be substituted by the zero elements. This process is shown in the following algorithm:

```
Let
L: left of array P
R: right of array P
For all values where  L < R
    P[i] = 0    if P[L] = P[i]     for i = L+1 to R
    P[j] = 0    if P[R] = P[j]     for j = R-1 to L+1
    Increment L by 1
    Decrement R by 1
```

Step3: fill()

The final step, fill(), is to replace any zero value in P by a value in the range 1 to ***N*** which is not exist in **P**. All zero values will be replaced through a sequence of one value from the left side of P and one value from the right side of P and repeating this sequence until all zero values are gone. The resulted array now contains all distinct values in the range 1 to ***N*** which represents the permutation stored in ***P***. This process is shown in the following algorithm:

```
Let
A: array contains missing values in P
m: number of missing values in A
i = 0
while ( i < m )
    j = N
    while( P[i] != 0 && j > 0 )
        decrement j
    if( j > 0 )
        P[j] = A[i]
        increment i
    k = 1
    while( P[k] != 0 && k ≤ N  )
        increment k
    if( k <= N )
        P[k] = A[i]
        increment i
```

An illustrative example: To illustrate the three steps of the process of **KBRP**, let us take the following example:
KEY: computer
Permutation size: 12
step1, fill() works as follows
array P holds first the ASCII code of the input

99	111	109	112	117	116	101	114				

Then values will be changed to the following

210	220	221	229	233	217	215	99	430	431	439	443

Finally, P will hold the following values

6	4	5	1	5	1	11	3	10	11	7	11

In step2, the repeated values are examined and replaced with zero and this is done by the function eliminate(). Now P looks like

6	4	5	1	0	0	0	3	10	0	7	11

Finally, the function fill() replaces all zero values with the missing values in P in order to get the final permutation P.

6	4	5	1	8	2	9	3	10	12	7	11

Application: Permutation is used for encryption. Shiho and Serge[15] stated that block cipher can be considered as an instance of a random permutation over a message block space. In block cipher, permutation is used to rearrange a message block. This permutation needs to be random and secret. I have built the method KBRP that provides randomness and secrecy. Randomness is available since producing the permutation is completely depending on the secret key and each key generate one permutation for a given permutation size. Secrecy of permutation is comprised in the generating way. Permutation is used in block cipher as a mapping function that maps the elements of a message block in its original position into a new position.. For example, a permutation, ***P***, of size 4 has four elements **P[1]**, **P[2]**, **P[3]**, and **P[4]** whose vales are 3,4,1, and 2 respectively. Any message block M of size 4 can be rearranged according to the permutation P. this mapping is shown in Fig. 1.

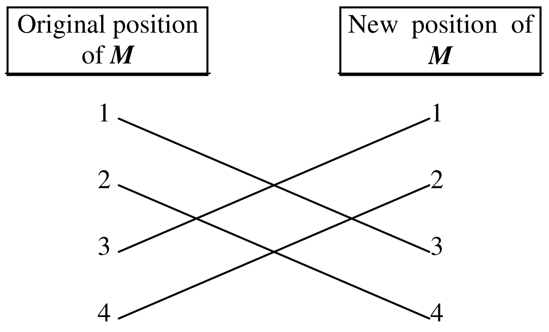


Fig. 1: Mapping message ***M*** with permutation ***P***

Mapping reflects the relation between original position and destination position of the message block. This relation should be weak for the reason of the difficulty of knowing or guessing the mapping (permutation). For this reason, I propose a test for the generated permutation and the sequence set one to block size (original positions of message block). This test is the correlation coefficient test (ρ), whose accepted values are $|\rho| < 0.5$; otherwise, the permutation can be easily changed by making a one right shift and then compute new correlation. Right shift continues until we get the proper correlation value. Fig. 2 shows the mapping for the permutation generated in our illustrative example, P, and then we make one right shift to get a new permutation, ***P1***. These permutations are

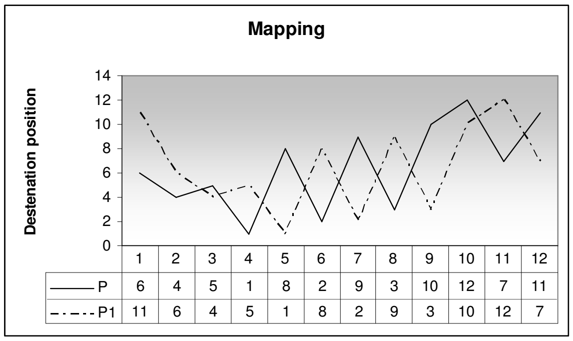
P

6	4	5	1	8	2	9	3	10	12	7	11

P1

11	6	4	5	1	8	2	9	3	10	12	7

Correlations for ***P*** and ***P1*** are 0.587413 and 0.20979 respectively.



	1	2	3	4	5	6	7	8	9	10	11	12
—— P	6	4	5	1	8	2	9	3	10	12	7	11
-·-·- P1	11	6	4	5	1	8	2	9	3	10	12	7

Fig. 2: Mapping two permutations into original position

CONCLUSION

This study introduced a new approach for generating permutation. This approach depends on using a specific key and size in order to cover the randomness and secrecy properties for permutation. This approach is intended to use permutation in block cipher; therefore, it is suggested that a statistical test can be used to consider the permutation for the block cipher. The weak correlation coefficient reflects the weak relation between the original position of an element in a message block and the destination position. In this paper the new "***KBPR***" method is applied with the key "computer" and size = 4.

The correlation test showed a value greater than 0.5; therefore, the permutation was changed by making a one right shift to obtain a new one. The modified permutation was tested and the correlation becomes less than 0.5 which is considered as a weak relation.

REFERENCES

1. Uspensky, J.V., 1937. Introduction to Mathematical Probability. New York, McGraw-Hill, pp: 18.
2. Gallian, J.A., 2002. Contemporary Abstract Algebra. 5/e, Houghton Mifflin, Boston JCSC, 19: 3 (Jan. 2004).
3. <http://mathworld.wolfram.com>
4. Sedgewick, R., 1977. Permutation Generation Methods. Comput. Surveys, 9: 137-164.
5. Heap, B.R., 1963. Permutations by Interchanges. Computer J., 6: 293-294.
6. Skiena, S., 1990. Permutations.§1.1 in Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp: 3-16.
7. Ives, F.M., 1976. Permutation enumeration: Four new permutation algorithms. Communications of the ACM**,** 19: 68-72.
8. Johnson, S.M., 1963. Generation of Permutations by Adjacent Transpositions. Math. Comput., 17: 282-285.
9. Richard, D., 1964. Algorithm 235: Random permutation. CACM, 7: 420.
10. Ord-Smith, R.J., 1971. Generation of permutation sequences. Comput, J., 14: 136-139.
11. Ord-Smith, R.J., 1970. Generation of permutation sequences. Comput, J., 13: 152-155.
12. Shadan, P., 1961. Permutation ordering and identification. Math. Mag., 34: 353-358.
13. William, C., 2004. Indexing permutation. Comput, J., 19: 296-300.
14. Hussain, S., 1998. Dynamic generation of permutation from a string of bits. Proc. SCCA'98, Applied Science University, Jordan, 21-22: 158-160.
15. Shiho, M. and S. Vaudenay. Comparison of Randomness Provided by Several Schemes for Block Ciphers. http://csrc.nist.gov/CryptoToolkit/aes/round2/conf3/papers/34-smoriai.pdf