I. J. Computer Network and Information Security, 2013, 1, 1-13
Published Online January 2013 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijcnis.2013.01.01

Exponential Brute-Force Complexity of A Permutation Based Stream Cipher

Mohammed Omari[1], Hamdy S. Soliman[2]
[1]LDDI Laboratory, Computer Science Department, University of Adrar, Adrar 01000, Algeria
[2]Computer Science and Engineering Department, New Mexico Tech, Socorro NM 87801, USA
omari@univ-adrar.org, hss@nmt.edu

Abstract —This paper introduces a permutation generation mechanism based on a shared secret key. The generated permutation vectors are used as encryption keys in a stream ciphering cryptosystem. We investigated various types of attacks on the known stream cipher RC4 and patched most of its loopholes, especially biased-byte and state-related attacks. Unique to our approach, we prove mathematically that the complexity of brute-forcing such a system is Ω(2n), where n is the key size in bytes. This paper also presents a complete security model using permutation-based encryption, in order to handle privacy. In addition, our approach achieved higher performance than that of existing peer techniques, while maintaining solid security. Experimental results show that our system is much faster than the existing security mechanisms, such as AES and DES.

Index Terms —Biased byte attack, exponential brute force, network security permutation vector generation, stream cipher

I. Introduction

A permutation describes an arrangement, or ordering, of objects [1]. Many algorithmic problems seek the best way to order a set of objects, including traveling salesman (the least-cost order to visit *n* cities), width (order the vertices of a graph on a line so as to minimize the length of the longest path), and graph isomorphism (order the vertices of one graph so that it is identical to another). Any algorithm for solving such problems must construct a series of permutations along the way.

There are *n*! permutations of *n* items, which grow exponentially to generate all permutations. Numbers like these should calm the urge of anyone interested in exhaustive search and help explain the importance of generating random permutations.

Fundamental to any permutation-generation algorithm is a notion of sequence order, the sequence in which the permutations are constructed, from first to last. The most natural generation order is lexicographic, the order in which permutations would appear if they were sorted numerically. Lexicographic order for $n = 3$ is {1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2}, and finally {3, 2, 1}. Although lexicographic order is aesthetically pleasing, there is often no particular reason to use it. Indeed, nonlexicographic orders lead to faster and simpler permutation generation algorithms [1][2].

The generation of random permutations is important for simplifying security algorithms. One way to do this [3] is the following two-line, linear-time algorithm. We assume that *Random(i,n)* generates a random integer between *i* and *n*.

```
for i ← 1 to n do
    a_i ← i          /* a =(1, 2, ..., n)*/
for i ← 1 to n do
    swap(a_i, a_Random(i, n))
```

It is not obvious that this algorithm generates all permutations uniformly. However, the validity of a security algorithm that is based on such linear generation of permutation vectors is yet to be proven in relation to peer algorithms. Permutations are also used to achieve "diffusion", a critical characteristic of a secure cipher [4], in symmetric-key encryption algorithms such as DES [5], Twofish [6] and Serpent [7]. Some permutations in cryptographic algorithms are not one-way only. For instance, the Expansion Permutation in DES maps some bits in the source data vector to multiple destinations in the result data vector [8].

The rest of this paper is organized as follows. Section 2 presents a description of the RC4 stream cipher with some flaws that made it insecure. In Section 3, we propose a permutation technique to be used in building secure stream ciphers. The base theorem of our crypto system is presented in Section 4 as well as some useful lemmas. Section 5 presents in detail the theorem proof which covers all cases of forming a new permutation vector. Our SDES crypto system is briefly presented in Section 6. Section 7 shows some simulation experiments and comparison of SDES with the state-of-the-art security mechanisms in terms of throughput. The conclusion is given in Section 8.

II. Related Work

Stream cipher algorithms are an important class of encryption techniques. They encrypt individual characters (usually binary digits) of a plaintext message one at a time, using an encryption transformation that varies with time [9]. In contrast, block ciphers tend to simultaneously encrypt groups of characters of a plaintext message using a fixed encryption transformation [10]. Stream ciphers

are generally faster than block ciphers [11] and have less complex hardware circuitry. They are also more appropriate, and in some cases mandatory (e.g., in some telecommunications applications), when buffering is limited or when characters must be individually processed as they are received.

RC4 is one of the dominant stream ciphers used in secure data communications [12]. Ron Rivest of RSA Data Security Inc developed the RC4 cipher in 1987, the details of which were published in 1996. RC4 is a stream cipher encryption system, which uses a shared key to shuffle a permutation vector, S, and randomly selects elements from it to encrypt and decrypt messages transferred during a particular communication session [13].

RC4 consists of two parts, as shown in Figure 1. The first is a key-scheduling algorithm, *KSA*, which turns a random key (whose typical size is 40-256 bits) into an initial permutation vector *S* of {1, …, *n*}; the second is a pseudo-random generation algorithm, *PRGA*, which uses *S* to generate a pseudo-random output sequence.

KSA(K):	*PRGA(K):*
/*Initialization*/	/*Initialization*/
for $i \leftarrow 1$ to n do	$i \leftarrow 1$
$S_i \leftarrow i$	$j \leftarrow 1$
$j \leftarrow 1$	/*Generation loop*/
/*Scrambling*/	$i \leftarrow i + 1$
for $i \leftarrow 1$ to n	$j \leftarrow j + S_i$
$j \leftarrow j + S_i + K_i$	swap(S_i, S_j)
swap(S_i, S_j)	$z \leftarrow S_{Si + Sj}$

Figure 1: Key scheduling and pseudo-random generation algorithms of RC4

There are several methods of attempting a brute force attack on RC4 that are classified into two categories: KSA-based attacks and PRGA-based attacks. Knowing that the initial state is enough to predict all of the keystream bits (regardless of the shared key K), PRGA-based attacks look for contradictions in the chosen keystream (in order to detect incorrect guesses) and discover some of the initial state entries. There has been considerable analysis of the probabilities of any given value being output by RC4. Most of these analyses have approached RC4 by looking at a given output.

Even though RC4 uses a permutation vector as its internal state box, the generated keystream is not necessarily redundancy-free. Fluhrer and McGrew [14] and Mantin and Shamir [15] defined a class of predictive states in which a non-negligible bias appears in the keystream. In their search for a polynomial-space distinguisher, they came up with a startling theorem, claiming that if $S_2 = 0$ and $S_1 \neq 2$, then $z_1 = 0$ with probability of 1.

In a good keystream generator, each bit of the output will depend on the entire key for its value; the relationship between the key and a given bit (or set of bits) should be extremely complicated [9]. However, RC4 uses the shared key only once (in the KSA); the shared key is not involved at all in the keystream generation. Recall that at each step of the PRGA, *S* changes in, at most, two locations; thus we can still expect the prefix of the output stream generated by RC4 from some permutation, *S*, to be highly correlated with the stream generated from the same *S* (or a slightly modified one) by RC4 [12].

III. Proposed Permutation Vector Generation

The generation of permutation vectors can be performed recursively. Given a permutation vector PV^j (a vector that contains all elements from 1 till *n*, in a specific order), the generation of the next permutation vector PV^{j+1} is based on PV^j and some other parameter that provides randomness. Our goal is to generate a large set of *PVs* whose sequence order is difficult to guess. In fact, the shared secret key (*SK*) utilization, in swapping the elements of *PV*, captures the notion of randomness in the abovementioned algorithm. Next is our linear algorithm to generate permutation vectors:

for $i \leftarrow 1$ to n do
  $PV_i \leftarrow i$
for $i \leftarrow 1$ to n do
swap(PV_i , PV_{SK_i}) /* $1 \leq SK_i \leq n$ */

The major advantage of using permutation vectors as encryption keys is the avoidance of biased byte analysis, in contrast with RC4 keystreams. In accordance with good keystream philosophy, an entry in the new generated permutation vector is a function of the entire key and the previous permutation vector, i.e., every bit in the new permutation vector is generated after performing exactly *n* swaps in the previous vector.

Another major contribution of our permutation generation algorithm is the continuous involvement of the shared key in the permutation vector generation. This will render the state-based attacks obsolete, since the attacker is forced to obtain the state and the key together in order to break the system. In order to increase the level of security, the system should update the shared key internally after each record. Therefore, the attacker is compelled to break a system with pseudo-multiple keys, instead of a single static key.

Most of cryptographic schemes are based on the "reducibility from hard problems" technique, which consists of proving that any successful protocol attack leads directly to the ability to solve a well-studied hard problem [11]. This "reference" problem is considered computationally unfeasible, given current knowledge and an adversary with bounded resources, e.g., the "integer factorization" and the "discrete logarithmic" problems. Such analysis yields the so-called provably secure protocols, although the security is conditional on the reference problem's being truly difficult. On the contrary, we will show that a cryptanalyst is cornered to the brute-force option only in order to guess the lexographic order of the generated permutation vectors. Hence, we will prove a theorem that underlimits such brute-force algorithmic complexity to an exponential function $\Omega(2^n)$,

with *n* the byte size of the shared secret key.

A. Permute function

A permutation function *Permute(X, K)* is a function that takes a permutation vector *X* (of size *m*) and a key *K* (of size *m*), and returns another permutation vector, as follows:

Permute(X, K):
  Temp ← *X*
    for *i* ← 1 to *m* do
    swap($Temp_i$, $Temp_{K_i}$)
  return *Temp*

B. Reverse permutation function

Similarly, we can define a reverse function of *Permute()* as follows:

ReversePermute(Y, K):
  Temp ← *Y*
  for *i* ← *m* downto 1 do
    swap($Temp_i$, $Temp_{K_i}$)
  return *Temp*

C. Reduction function

A reduction function *Reduction(V)* of a permutation vector *V* of size *m*+1, is a function that reduces *V* to a permutation vector of size *m* as follows (illustrated in Figure 2):

Reduction(V):
  j ← 1
    for *i* ← 1 to *m*+1 do
    if $V_i \neq m+1$ then /* skipping the '*m*+1' value*/
      $Temp_j \leftarrow V_i$
    j ← j + 1
  return *Temp* /* Temp is a vector of size *m**/

Thus, if *R* = *Reduction(V)*, and $V_p = m+1$, then:

$$R_i = \begin{cases} V_i, if 1 \leq i \leq p \\ V_i, if 1 \leq i \leq p \end{cases}$$

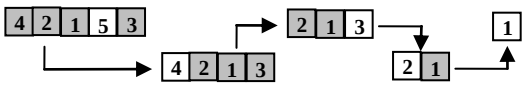


Figure 2: Multiple reduction of a permutation vector

D. Index map function

The index map $IMAP_{X,V}$ corresponding to two permutation vectors *X* (of size *m*) and *V* (of size *m* or higher) is defined as follows (illustrated in Figure 3):

$IMAP_{X,V}(i) = j$ if and only if $X_i = V_j$, $(i, j \in \{1, 2, \ldots, m\})$.

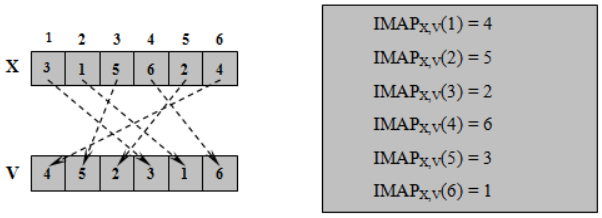


Figure 3: Index map function for two permutation vectors

E. Reverse function

A reverse function *Reverse(X)* of a vector *X* of size *m*, is a function that reverses backwards the coordinates of *X* (illustrated in Figure 4):

Reverse(X)
  Temp ← *X*
    for *i* ← 1 to *m* do
  $Temp_i = X_{m-i} + 1$
  return *Temp*

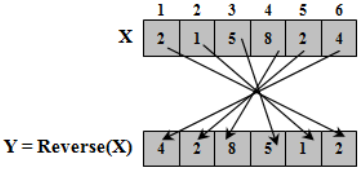


Figure 4: The corresponding reverse vector of X = (2, 1, 5, 8, 2, 4)

IV. Theorem Statement

In order to strengthen our security mechanism, we present throughout this paper a detailed proof the following theorem: "*Given two permutation vectors V and W of size m+1, there are 2^m different keys (K) (of size m+1) that satisfy W = Permute(V, K).*"

This theorem provides theoretical strength to cryptosystems in a way that a cryptanalyst who managed to obtain two consecutive encryption keys (permutation vectors), which is not an obvious task, will find it very hard to break the system and guess the secret key (2^m possibilities) that is indispensible to calculate the next encryption keys.

A. Lemma 1

Given an index map function $IMAP_{X,V}$ of two permutation vectors *X* and *V*, $IMAP_{X,V}$ remains unchanged when swapping any two elements in *X* and *V*; swap(X_{i_1}, X_{i_2}) on *X* and swap(V_{j_1}, V_{j_2}) on *V*, where i_1, i_2, j_1, $j_2 \in \{1, \ldots, m\}$) only if $X_{i_1} = V_{j_1}$ and $X_{i_2} = V_{j_2}$ (illustration is shown in Figure 5).

Proof: Suppose that $X_{i_1} = V_{j_1} = a$, and $X_{i_2} = V_{j_2} = b$, then $IMAP_{X,V}(i_1) = j_1$ and $IMAP_{X,V}(i_2) = j_2$. After performing swap(X_{i_1}, X_{i_2}), $X_{i_1} = b$ and $X_{i_2} = a$. Also, after performing swap(V_{j_1}, V_{j_2}), $V_{j_1} = b$ and $V_{j_2} = a$. Thus, $X_{i_1} = V_{j_1}$ and $X_{i_2} = V_{j_2}$; then, $IMAP_{X,V}(i_1)$ is still equal to j_1 and $IMAP_{X,V}(i_2)$ is still equal to j_2. Since i_1 and i_2 are the only indices involved in swap(X_{i_1}, X_{i_2}), then $IMAP_{X,V}(i)$ still maintains the same value, for any other index $i \notin \{i_1, i_2\}$.

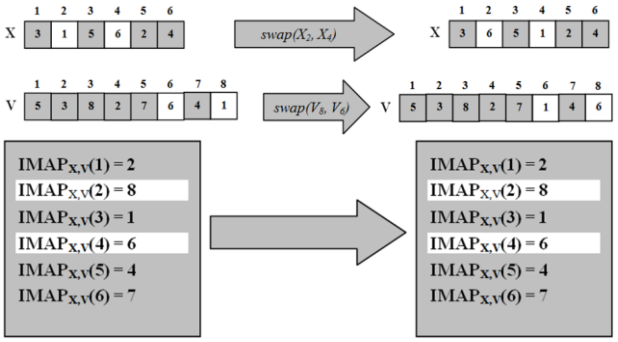


Figure 5: IMAP stability.

V. Theorem Proof

The proof of the above mentioned theorem is obtained simply by induction, which is based on proving the base case and the induction step.

A. Base Case: m = 2

Table I. All possible keys that permute certain permutation vector V to certain permutation vector W of size 2

V	W	All Possible values for K
[1, 2]	[1, 2]	[1, 2], [2, 1]
[1, 2]	[2, 1]	[1, 1], [2, 2]
[2, 1]	[2, 1]	[1, 1], [2, 2]
[2, 1]	[2, 1]	[1, 2], [2, 1]

Table I shows that there are $2^1 = 2$ different keys K that permute V into W, for any permutation vectors V and W of size 2.

B. Induction Step

Given two permutation vectors V and W of size m+1, we will prove that there are 2^m keys that permute V into W, assuming that for any two permutation vectors X and Y of size m, there are 2^{m-1} keys that permute X into Y (inductive hypothesis).

Given two permutation vectors V and W of size m+1 (where $V_{p_1} = W_{p_2}$ = m+1), we extract three permutation vectors X, Y (Figure 6) and Z (Figure 7), of size m, as follows. X and Y are the corresponding reductions of V and W, respectively, i.e., X = *Reduction(V)* and Y = *Reduction(W)*.

Z is obtained directly from Y as follows:

$$Z_i = \begin{cases} Y_i, & if\ i < p_2 \\ Y_{p_1-1}, & if\ i = p_2 \\ Y_{i-1}, & if\ p_2 < i < p_1 \\ Y_i, & if\ i \geq p_1 \end{cases} \quad (1)$$

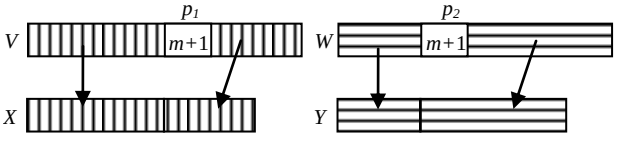


Figure 6: V and W reduction to X and Y respectively

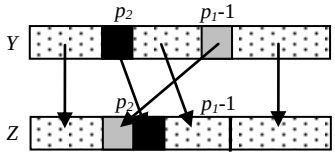


Figure 7: Z vector extraction from Y

a. Case of $p_1 > p_2$

In this section, we will investigate a methodology to build keys that permute V into W, where the element m+1 moves backward. First, we will construct 2^{m-1} keys based on the inductive hypothesis. Then we will deduce a second set of 2^{m-1} keys that basically permute a different vector Vr to a different vector Wr, where m+1 still moves backward from Vr to Wr, and prove that these keys also permute V into W.

First set of 2^{m-1} keys

Based on the inductive hypothesis, there are 2^{m-1} keys that permute X into Y. Also, there are 2^{m-1} keys that permute X into Z. Since X is the reduction of V and Y is the reduction of W, we will try to construct 2^{m-1} keys of size m+1 from the existing 2^{m-1} keys that permute X to Y. Unfortunately, this process will fail, as we will show later.

Note that V has all elements of X plus an extra element, m+1, and W has all elements of Y plus an extra element, m+1. We will consider a key $K^{X\to Y}$ from the 2^{m-1} keys that permute X into Y. We will try to expand $K^{X\to Y}$ into a larger key of size m+1 with the property of permuting V into W. Therefore, the permutation algorithm based on the expanded key (here, $K^{V\to W}$) has basically two tasks:

(i) permuting the X elements inside V into the Y elements in W, and

(ii) moving the value m+1 from position p_1 to position p_2.

We will work on task *(i)* separately, and ignore the fact that m+1 is migrating from p_1 to p_2. Practically, we will set the entry of index p_1 in the key $K^{V\to W}$ to be equal to p_1 itself (Figure 8). Hence, the permutation of V based on $K^{V\to W}$ will skip the "moving" of m+1. Then, we fill the rest of the $K^{V\to W}$ entries from $K^{X\to V}$ sequentially, as follows:

$$K_i^{V\to W} = \begin{cases} K_i^{X\to Y} & if\ i < p_1 \\ p_1 & if\ i = p_1 \\ K_{i-1}^{X\to Y} & if\ i > p_1 \end{cases}$$

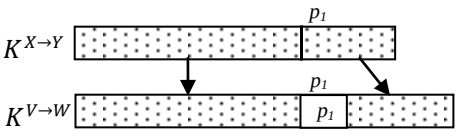


Figure 8: Key expansion to permute V into W

Due to the insertion of the element m+1 into X, $V_{i+1} = X_i$ when $i \geq p_1$; hence the next modification is necessary, in order to cope with the shifting of X elements, indexed from p_1 through m, filling V elements, indexed from p_1+1 through m+1:

for $i \leftarrow 1$ to m+1 do
  if $i \neq p_1$ then
    if $K_i^{V\rightarrow W} \geq p_1$ then increment $K_i^{V\rightarrow W}$

Now, we will work on task *(ii)*; i.e., moving m+1 in V from p_1 to p_2 (Figure 9). This can be done easily by setting $K_{p_1}^{V\rightarrow W}$ to p_2.

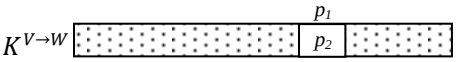


Figure 9: Modification of the p_1 entry to let m+1 migrate from p_1 to p_2

However, this single change in $K^{V\rightarrow W}$ leads to two side effects. First, when swap(V_{p_1}, $V_{K_{p_1}^{V\rightarrow W}} = p_2$) is executed, X_{p_2} and V_{p_2} are no longer identical, violating the strict equality of $V_i = X_i$, when $i < p_1$ (Figure 10). In fact, X_{p_2} will equal V_{p_1}.

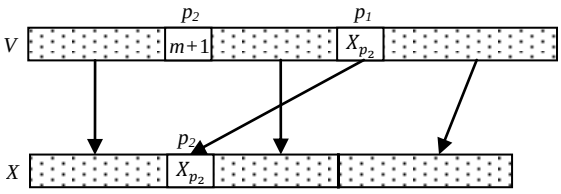


Figure 10: Violation of the reduction property when a unilateral swap occurs on V only

Any further swapping involving X_{p_2} (using $K^{X\rightarrow Y}$) should be translated to an access to V_{p_1} by $K^{V\rightarrow W}$:

for $i \leftarrow p_1$+1 to m+1 do
  if $K_i^{V\rightarrow W} = p_2$, then $K_i^{V\rightarrow W} \leftarrow p_1$

The second side effect is that, at the end of the permutation algorithm, the elements of the resulting vector between position p_2+1 and p_1 will be rotated to the left, compared to the expected W.

Hence, we failed to obtain W by permuting V using $K^{V\rightarrow W}$, which was constructed based on $K^{X\rightarrow Y}$. Had we selected a mysterious permutation vector identical to Y, except with the element range between p_2 and p_1-1 rotated right by one step, we would have obtained our target W (Figure 11). This mysterious vector is exactly Z. Therefore, we can generate $K^{V\rightarrow W}$ using $K^{X\rightarrow Z}$ instead of $K^{X\rightarrow Y}$.

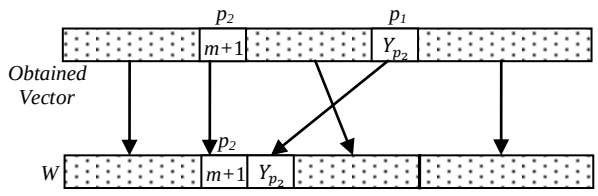


Figure 11: The violation of the reduction property misleads the swapping into a vector other than W

Here is a three–step algorithmic depiction of the above discussion:

Step 1

$$K_i^{V\rightarrow W} = \begin{cases} K_i^{X\rightarrow Z} & \text{if } i < p_1 \\ p_1 & \text{if } i = p_1 \\ K_{i-1}^{X\rightarrow Z} & \text{if } i > p_1 \end{cases} \quad (2)$$

Step 2 (3)

  for $i \leftarrow 1$ to m+1 do
    if $i \neq p_1$ then
      if $K_i^{V\rightarrow W} \geq p_1$ then increment $K_i^{V\rightarrow W}$

Step 3 (4)

  for $i \leftarrow p_1$+1 to m+1 do
    if $K_i^{V\rightarrow W} = p_2$, then $K_i^{V\rightarrow W} \leftarrow p_1$

In order to prove that the newly constructed key $K^{V\rightarrow W}$ permutes V into W, we will perform *Permute(V, $K^{V\rightarrow W}$)* in parallel with *Permute(X, $K^{X\rightarrow Z}$)*, as shown in Table II. Then, we will show that the result of *Permute(V, $K^{V\rightarrow W}$)* is identical to W.

Phase I: At the end of this phase, $IMAP_{xTemp,vTemp}$ will be set as follows (based on *IMAP* Definition):

$$IMAP_{xTemp,vTemp}(i) = \begin{cases} i & if\ 1 \leq i < p_1 \\ i+1 & if\ p_1 \leq i \leq m \end{cases}$$

Phase II: In order to maintain the same $IMAP_{xTemp,vTemp}$ after executing *Phase II*, we need to show, based on *Lemma* 1, that $xTemp_i = vTemp_i$, and $xTemp_{K_i^{X\rightarrow Z}} = vTemp_{K_i^{V\rightarrow W}}$, for $1 \leq i < p_1$.

i. $xTemp_i = vTemp_i$ (based on $IMAP_{xTemp,vTemp}$).

ii. If $K_i^{V\rightarrow W} < p_1$, then $K_i^{V\rightarrow W}$ $K_i^{X\rightarrow Z}$ =, based on (2). Since $xTemp_j = vTemp_j$, for $j < p_1$ (based on $IMAP_{xTemp,vTemp}$), then $xTemp_{K_i^{X\rightarrow Z}} = vTemp_{K_i^{V\rightarrow W}}$ (substituting j with $K_i^{X\rightarrow Z}$ and $K_i^{V\rightarrow W}$).

iii. If $K_i^{V\rightarrow W} > p_1$, then $K_i^{V\rightarrow W} = K_i^{X\rightarrow Z} + 1$, based on (2) and (3). Since $xTemp_j = vTemp_j$+1, for $j > p_1$ (based on $IMAP_{xTemp,vTemp}$), then $xTemp_{K_i^{X\rightarrow Z}} = vTemp_{K_i^{V\rightarrow W}}$ (substituting j and j+1 with $K_i^{X\rightarrow Z}$ and $K_i^{V\rightarrow W}$, respectively).

Thus, the same $IMAP_{xTemp,vTemp}$ is still maintained at the end of *Phase II*. We skipped the case of $K_i^{V\rightarrow W} = p_1$, because it is inconsistent with (3).

Table II. Running Permute(X, $K^{X\rightarrow Z}$) and Permute(V, $K^{V\rightarrow W}$) in parallel

Pha-ses	*Permute(X, $K^{X\rightarrow Z}$)*	*Permute(V, $K^{V\rightarrow W}$)*
I	$xTemp \leftarrow X$	$vTemp \leftarrow V$
II	for $i \leftarrow 1$ to p_1-1 do swap($xTemp_i$, $xTemp_{K_i^{X\rightarrow z}}$)	for $i \leftarrow 1$ to p_1-1 do swap($vTemp_i$, $vTemp_{K_i^{V\rightarrow w}}$)
III		swap($vTemp_{p_1}$, $vTemp_{p_2}$)
IV	for $i \leftarrow p_1$+1 to m+1 do swap($xTemp_{i-1}$, $xTemp_{K_{i-1}^{X\rightarrow z}}$)	for i $\leftarrow$ p1+1 to m+1 do swap($vTemp_i$, $vTemp_{K_i^{V\rightarrow w}}$)
V	return *xTemp*	return *vTemp*

Phase III: Since swapping is performed on *vTemp* only, a new $IMAP_{xTemp,vTemp}$ is obtained:

$$IMAP_{xTemp,vTemp}(i) = \begin{cases} i & if\ 1 \le i < p_2 \\ p_1 & if\ i = p_2 \\ i & if\ p_2 < i < p_1 \\ i+1 & if\ p_1 \le i \le m \end{cases}$$

Also, $xTemp_{p_2} = m+1$.

Phase IV: In order to maintain the same $IMAP_{xTemp,vTemp}$ after executing this set of swaps, we need to show, based on *Lemma 1*, that $V_i = X_{i-1}$, and $xTemp_{K_{i-1}^{X\to Z}} = vTemp_{K_i^{V\to W}}$, for $i > p_1$.

i. $X_{i-1} = V_i$ (based on $IMAP_{xTemp,vTemp}$).

ii. If $K_i^{V\to W} < p_1$, then $K_i^{V\to W} = K_{i-1}^{X\to Z}$, based on (2). Also, $K_i^{V\to W} \neq p_2$, based on (4). Since $xTemp_j = vTemp_j$, for $j < p_1$ and $j \neq p_2$ (based on $IMAP_{xTemp,vTemp}$), then $xTemp_{K_i^{X\to Z}} = vTemp_{K_i^{V\to W}}$ (substituting j with $K_{i-1}^{X\to Z}$ and $K_i^{V\to W}$).

iii. If $K_i^{V\to W} = p_1$, then $K_{i-1}^{X\to Z} = p_2$ based on (4). But $xTemp_{p_2} = vTemp_{p_1}$, based on $IMAP_{xTemp,vTemp}$. Therefore, $= vTemp_{K_i^{V\to W}}$.

iv. If $K_i^{V\to W} > p_1$, then $K_i^{V\to W} = K_{i-1}^{X\to Z} + 1$, based on (2) and (3). Since $xTemp_j = vTemp_j + 1$, for $j > p_1$ (based on $IMAP_{xTemp,vTemp}$), then $xTemp_{K_{i-1}^{X\to Z}} = vTemp_{K_i^{V\to W}}$ (substituting j and $j+1$ with $K_{i-1}^{X\to Z}$ and $K_i^{V\to W}$, respectively).

So, based on *Lemma 1*, the last version of $IMAP_{xTemp,vTemp}$ is still maintained at the end of *Phase IV*.

Since $K_i^{V\to W} \neq p_2$, based on (4), $vTemp_{p_2}$ is not swapped in this phase.

Thus,

$$vTemp_{p_2} = m+1 \quad (5)$$

Phase IV: In this phase, the result of permuting *X* and *V* is returned. Based on the inductive hypothesis, *Permute(X, $K^{X\to Z}$)* results in *Z*, i.e., *xTemp = Z*. However, we still need to show that *vTemp* is identical to *W*.

Based on the last version of $IMAP_{xTemp,vTemp}$, we have:

$$xTemp_i = \begin{cases} vTemp_i, & if\ 1 \le i \le p_2 \\ vTemp_{p_1}, & if\ i = p_2 \\ vTemp_i, & if\ p_2 < i < p_1 \\ vTemp_{i+1}, & if\ p_1 \le i \le m \end{cases}$$

Therefore (substituting *xTemp* with *Z*):

$$Z_i = \begin{cases} vTemp_i, & if\ 1 \le i \le p_2 \\ vTemp_{p_1}, & if\ i = p_2 \\ vTemp_i, & if\ p_2 < i < p_1 \\ vTemp_{i+1}, & if\ p_1 \le i \le m \end{cases}$$

Then, based on (1), we have:

$$\begin{cases} Y_i = vTemp_i, & if\ 1 \le i \le p_2 \\ Y_{p_1-1} = vTemp_{p_1}, & if\ i = p_2 \\ Y_{i-1} = vTemp_i, & if\ p_2 < i < p_1 \\ Y_i = vTemp_{i+1}, & if\ p_1 \le i \le m \end{cases}$$

which is equivalent to:

$$\begin{cases} Y_i = vTemp_i, & if\ 1 \le i \le p_2 \\ Y_i = vTemp_{i+1} & if\ p_2 < i < p_1 - 1 \\ Y_{p_1-1} = vTemp_{p_1} & \\ Y_i = vTemp_{i+1}, & if\ p_1 \le i \le m \end{cases}$$

After simplification:

$$Y_i = \begin{cases} vTemp_i, & if\ 1 \le i < p_2 \\ vTemp_{i+1}, & if\ p_2 \le i \le m \end{cases}$$

But *Y = Reduction(W)* . Then, based on the reduction definition we have:

$$\begin{cases} W_i = vTemp_i, & if\ 1 \le i < p_2 \\ W_{i+1} = vTemp_{i+1}, & if\ p_2 \le i \le m \end{cases}$$

$$\begin{cases} W_i = vTemp_i, & if\ 1 \le i < p_2 \\ W_{i+1} = vTemp_{i+1}, & if\ p_2 \le i \le m \end{cases}$$

Also, $vTemp_{p_2} = m+1$, based on (5), and $W_{p_2} = m+1$ based on the induction step. Therefore:

$$\begin{cases} W_i = vTemp_i, & if\ 1 \le i < p_2 \\ W_{p_2} = vTemp_{p_2}, & \\ W_{i+1} = vTemp_{i+1}, & if\ p_2 \le i \le m \end{cases}$$

Hence: *vTemp = W*.

At this stage, we have proved that the constructed key $K^{V\to W}$ does permute *V* into *W*. Therefore, given that there are 2^{m-1} keys that permute *X* into *Z* (based on the induction hypothesis), we can expand them to generate 2^{m-1} keys that permute *V* into *W*, following (2), (3) and (4) (see example in Figure 12).

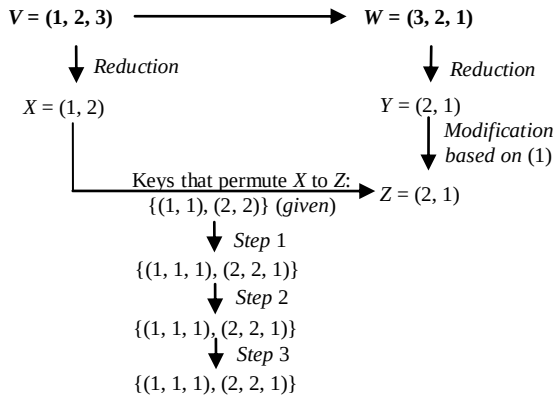


Figure 12: First set of two keys that permute (1, 2, 3) to (3, 2, 1)

Second set of 2^{m-1} keys

In order to complete the proof, we need to construct

another set of 2^{m-1} keys that permutes *V* into *W*. For this purpose, we will investigate the relationship between keys that permute *V* into *W*, and keys that shuffle the reverse vector of *W* (*Wr = Reverse(W)*) into the reverse vector of *V* (*Vr = Reverse(V)*).

Suppose that there is a key $\hat{K}^{V\to W}$ (different from any of the 2^{m-1} keys generated in the first set) that permutes *V* into *W*, i.e., *W* = Permute(*V*, $\hat{K}^{V\to W}$). Then, *V* = *ReversePermute*(*W*, $\hat{K}^{V\to W}$), based on the reverse permutation definition.

Table III. Modifying the ReversePermute() code

Original execution of ReversePermute(W, $\hat{K}^{V\to W}$)	Modified version of ReversePermute(W, $\hat{K}^{V\to W}$)
wTemp ← *W* for *i* ← *m*+1 downto 1 do swap($wTemp_i$, $wTemp_{\hat{K}_i^{V\to w}}$) return *wTemp*	*wTemp* ← *W* for *i* ← 1 to *m*+1 do swap($wTemp_{m+2-i}$, $wTemp_{\hat{K}_{m+2-i}^{V\to W}}$) return *wTemp*

As shown in Table III, we modified the original code of *ReversePermute*(*W*, $\hat{K}^{V\to W}$) in order to resemble a regular *Permute()* function, i.e., the inner loop becomes *to..do* instead of *downto..do*. Consequently, *wTemp* is scanned backward starting by $wTemp_{m+1}$, $wTemp_m$, $wTemp_{m-1}$, …, $wTemp_2$, and $wTemp_1$. In order to restore the original form of scanning elements in an increasing order, we may assign *Wr* to *wTemp* instead of *W*, as shown next:

wTemp ← *Wr*
for *i* ← 1 to *m*+1 do
  swap($wTemp_i$, $wTemp_{m+2-\hat{K}_{m+2-i}^{V\to W}}$)
return *wTemp*

Since *wTemp* was reversed at the beginning of the above function, the returned vector will be also reversed at the end of execution, i.e., *wTemp* will be identical to *Vr* instead of *V*.

The above function looks like a typical *Permute()* function, except for the second parameter of swap(). For this matter, we will construct a new key $K^{Wr\to Vr}$ as follows:

$$\mathrm{K_i^{Wr\to Vr}} = \mathrm{m+2-K_{m+2-i}^{V\to W}}, \text{for } 1 \le \mathrm{i} \le \mathrm{m+1} \qquad (6)$$

And similarly,

$$\mathrm{\hat{K}_i^{V\to W}} = \mathrm{m+2-K_{m+2-i}^{Wr\to Vr}}, \text{for } 1 \le \mathrm{i} \le \mathrm{m+1} \qquad (7)$$

Therefore, the above algorithm will look like:

wTemp ← *Wr*
for *i* ← 1 to *m*+1 do
  swap($wTemp_i$, $wTemp_{\hat{K}_i^{Wr\to Vr}}$)
return *wTemp*

which is identical to *Permute*(*Wr*, $K^{Wr\to Vr}$), which results in *Vr*.

Thus, we found a relationship between keys that permute *V* into *W*, and keys that permute *Wr* into *Vr*, referring to (6) and (7). Therefore, if we could generate keys that permute *V* into *W*, we could easily infer the same number of keys that permute *Wr* into *Vr*, and vice versa. Since $V_{p_1} = W_{p_2} = m+1$, then, $Vr_{m+2-p_1} = Wr_{m+2-p_2} = m+1$. Then, we can set two new positions $\hat{p}_1$ and $\hat{p}_2$ to be $m+2-p_2$ and $m+2-p_1$, respectively. Since p_1 is greater than p_2, then $\hat{p}_1$ is also greater than $\hat{p}_2$. Based on the previous section, there are 2^{m-1} keys that permute *Wr* into *Vr*, since $\hat{p}_1 > \hat{p}_2$. We will consider a key $K^{Wr\to Vr}$ from this set. Then, based on (7), we can infer a new key $\hat{K}^{V\to W}$ that permutes *V* into *W*. Consequently, we can infer another set of 2^{m-1} keys in this manner.

Now, we will show that $\hat{K}^{V\to W}$ does not belong to the first set of 2^{m-1} keys that we generated previously. For this purpose, we will choose a key, $K^{V\to W}$, from the previously generated keys (first set).

We know that $K_i^{Wr\to Vr} \neq \hat{p}_1$, when $i < \hat{p}_1$, based on (2) and (3).

Therefore, $m+2-\hat{K}_{m+2-i}^{V\to W} \neq \hat{p}_1$, for $i < \hat{p}_1$.

Then, $m+2-\hat{K}_{m+2-i}^{V\to W} \neq m+2-p_2$, for $i < m+2-p_2$.

Therefore $\hat{K}_{m+2-i}^{V\to W} \neq p_2$,
for $i < m+2-p_2$.

So, if $j = m+2-i$, then $\hat{K}_j^{V\to W} \neq p_2$,
for $m+2-j < m+2-p_2$.

Therefore, $\hat{K}_j^{V\to W} \neq p_2$, for $j > p_2$.

Hence, when $j = p_1$, $\hat{K}_{p_1}^{V\to W} \neq p_2$.

But, for any key $K^{V\to W}$ from the first set, we have $K_{p_1}^{V\to W} = p_2$, based on (2).

Therefore, $\hat{K}^{V\to W}$ does not belong to the 2^{m-1} set of generated keys.

To conclude this subsection: given two permutation vectors *V* and *W* of size *m*+1, where $V_{p_1} = W_{p_2} = m+1$, and $p_1 > p_2$, we constructed 2^{m-1} keys that permutes *V* into *W*, and inferred a second, different set of 2^{m-1} keys that also permutes *V* into *W*, for a total of 2^m keys (see example in Figure 13).

Hence, we finished the proof by induction for the case of ($p_1 > p_2$).

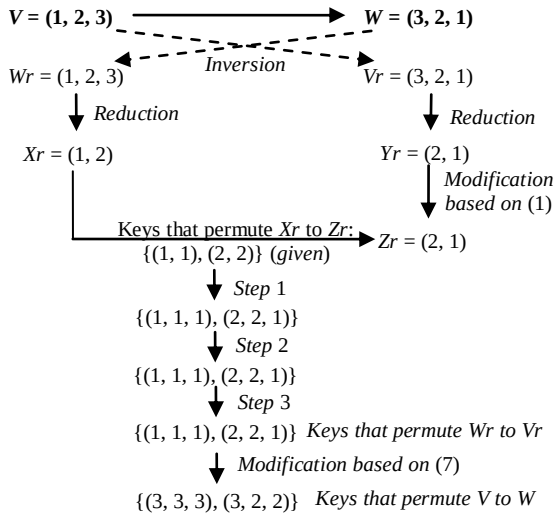


Figure 13: Second set of two keys that permute (1, 2, 3) to (3, 2, 1)

(b) Case of $p_1 < p_2$

In this case, the element m+1 moves forward from position p_1 in V to position p_2 in W. We will try to find a specific pair of permutation vectors, $\hat{V}$ and $\widehat{W}$, where the element m+1 moves backward ($\hat{V}_{\hat{p}_1}=\widehat{W}_{\hat{p}_2}= m$+1, and $\hat{p}_1 > \hat{p}_2$). Then, we have to show that any key that permutes $\hat{V}$ into $\widehat{W}$ also permutes V into W. Since there are 2^m keys that permute $\hat{V}$ into $\widehat{W}$ (based on *case (a)*), we can infer that there are also 2^m keys that permute V into W.

Index Map Set

An index map set, $IMAPS_{X,V}(A)$, corresponding to a set of indices, A, is the set of indices that are produced by the function $IMAP_{X,Y}$ over the set A, i.e., $IMAPS_{X,V}(A) = \{IMAP_{X,V}(i) \mid i \in A\}$.

Lemma 2: For any given set of indices, A, $|A| = |IMAPS_{X,V}(A)|$, where X and V are two permutation vectors.

Proof:

1. If $|A| < |IMAPS_{X,V}(A)|$, then $IMAP_{X,V}$ is not a function.
2. If $|A| > |IMAPS_{X,V}(A)|$, then there exist two different indices i, $j \in A$ where $IMAP_{X,V}(i) = IMAP_{X,V}(j)$. Therefore, if $r = IMAP_{X,V}(i) = IMAP_{X,V}(j)$, then $X_i = V_r$ and $X_j = V_r$. Thus $X_i = X_j$, which is a contradiction, since X is a permutation vector.

Lemma 3: (*Equilibrium Principle*) Given two permutation vectors, V and W, where $V_{p_1} = W_{p_2} = m$+1, and $p_1 < p_2$, there exists an index $j \neq p_1$ such that $j > IMAP_{V,W}(j)$. In other words, since the element m+1 moves forward from p_1 in V into p_2 in W, there should be another element, v_1, that moves backward from position j in V into position $IMAP_{V,W}(j)$ in W (Figure 14).

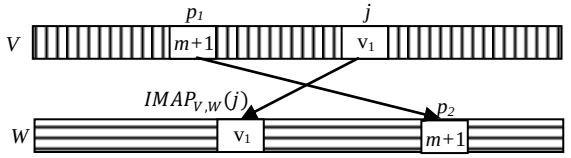


Figure 14: The equilibrium principle

Proof: By contradiction. Suppose that $j \leq IMAP_{V,W}(j)$, for any $j \neq p_1$. Based on that, if $j > p_1$, then $IMAP_{V,W}(j) > p_1$. Given that $IMAP_{V,W}(p_1) = p_2 \in \{p_1+1, \ldots, m+1\}$ since $p_1 < p_2$; therefore $IMAPS_{V,W}\{p_1, \ldots, m+1\} \subseteq \{p_1+1, \ldots, m+1\}$, which is a contradiction based on *Lemma 2*, since $|\{p_1, \ldots, m+1\}| > |\{p_1+1, \ldots, m+1\}|$.

So, based on *Lemma 3*, there exists $j_1 \neq p_1$ where $j_1 > IMAP_{V,W}(j_1) = j_2$, and $V_{j_1} = W_{j_2} = v_1$.

Then, we will construct a permutation vector, $\hat{V}$, of size m+1 as follows:

$$\hat{V}_i = \begin{cases} V_i, & \text{if } i \neq p_1, j_1 \\ V_{j_1} = v_1, & \text{if } i = p_1 \\ V_{p_1} = m+1, & \text{if } i = j_1 \end{cases} \quad (8)$$

And another permutation vector $\widehat{W}$ of size m+1 as follows:

$$\widehat{W}_i = \begin{cases} W_i, & \text{if } i \neq p_2, j_2 \\ W_{j_2} = v_1, & \text{if } i = p_2 \\ W_{p_2} = m+1, & \text{if } i = j_2 \end{cases} \quad (9)$$

In other words, we perform swap(V_{p_1}, V_{j_1}) in V to get $\hat{V}$, and swap(W_{p_2}, W_{j_2}) in W to get $\widehat{W}$ (Figure 15).

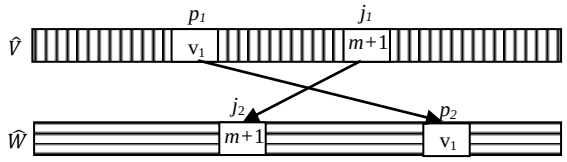


Figure 15: Utilization of the equilibrium principle to generate different vectors V and W

Therefore, $\hat{V}$ and $\widehat{W}$ are also two permutation vectors of size m+1. Since $j_1 > j_2$, we have 2^m keys that permute $\hat{V}$ into $\widehat{W}$ based *case (a)*. Then, all we need is to show that these keys also permute V into W.

Lemma 4 (*Value-independent property of the 'Permute()' function*): Given two permutation vectors A and B of size m+1, a key K of size m+1, a pair $(a, b) \in (\{1, \ldots, m+1\})^2$, a position p where $A_p = a$ and $B_p = b$, two permutation vectors $C = Permute(A, K)$ and $D = Permute(B, K)$; therefore, if $C_q = a$ for certain position q, then $D_q = b$.

Proof: by loop invariant. Suppose that we run *Permute(A, K)* and *Permute(B, K)* in parallel as shown in Table IV. We will set the loop invariant as "if $aTemp_t = a$ for some position t, then $bTemp_t = b$."

Table IV. Running Permute(A, K) and Permute(B, K) in parallel

Permute(A, K)	Permute(B, K)
$aTemp \leftarrow A$ for i $\leftarrow$ 1 to m+1 do swap($aTemp_i$, $aTemp_{K_i}$) return $aTemp$	$bTemp \leftarrow B$ for i $\leftarrow$ 1 to m+1 do swap($bTemp_i$, $bTemp_{K_i}$) return $bTemp$

Initialization: Before executing the two loops, $aTemp = A$ and $bTemp = B$. Therefore, $aTemp_p = a$ and $bTemp_p = b$ for position p defined in *Lemma 4*.

Maintenance: Suppose that before executing the i^{th} loop, there exists a position t such that $aTemp_t = a$ and $bTemp_t = b$. Also, we will set $j = K_i$.

i. If $i = t$ and $j = t$, then a and b will remain in their position t inside *aTemp* and *bTemp*, respectively, after executing the i^{th} loop.

ii. If $i = t$ and $j \neq t$, then a and b will swap to position j inside *aTemp* and *bTemp*, respectively, after executing the i^{th} loop.

iii. If $i \neq t$ and $j = t$, then a and b will swap to position i inside *aTemp* and *bTemp*, respectively, after executing the i^{th} loop.

iv. If $i \neq t$ and $j \neq t$, then a and b are not involved at all in the i^{th} swap operation. Therefore, a and b will remain in their position t inside *aTemp* and *bTemp*,

respectively, after executing the i^{th} loop.

Therefore, *a* and *b* will share the same position (*t*, *i*, or *j*) in *aTemp* and *bTemp*, respectively, after executing the i^{th} loop, based on the above cases. Hence, the loop invariant is still valid at the end of the i^{th} loop.

Termination: When $i = m+2$, *Permute(A, K)* and *Permute(B, K)* will return *aTemp* and *bTemp*, and store them in *C* and *D*, respectively. Therefore, if $C_q = a$ for some position *q*, then $D_q = b$.

Next, we will choose a key $K^{\hat{V}\to\hat{W}}$ from the 2^m keys that permute $\hat{V}$ into $\hat{W}$ and observe the result of Permute(*V*, $K^{\hat{V}\to\hat{W}}$). For this purpose, we will run *Permute*(*V*, $K^{\hat{V}\to\hat{W}}$) and *Permute*($\hat{V}K^{\hat{V}\to\hat{W}}$,) in parallel, as shown in Table V.

Table V. Running Permute(V, $K^{\hat{V}\to\hat{W}}$) and Permute($\hat{V}$,$K^{\hat{V}\to\hat{W}}$) in parallel

Phases	Permute(*V*, $K^{\hat{V}\to\hat{W}}$)	Permute($\hat{V}$,$K^{\hat{V}\to\hat{W}}$)
I	$vTemp \leftarrow V$	$\hat{v}Temp \leftarrow \hat{V}$
II	for $i \leftarrow 1$ to m+1 do swap($vTemp_i$, $vTemp_{K_i^{\hat{V}\to\hat{W}}}$)	for $i \leftarrow 1$ to m+1 do swap($\hat{v}Temp_i$, $\hat{v}Temp_{K_i^{\hat{V}\to\hat{W}}}$)
III	return *vTemp*	return $\hat{v}Temp$

We know that *Permute*($\hat{V}$, $K^{\hat{V}\to\hat{W}}$) results in $\hat{W}$. We will use *Lemma 4* to show that *Permute*(*V*, $K^{\hat{V}\to\hat{W}}$) also results in *W*, i.e., at the end of *Permute*(*V*, $K^{\hat{V}\to\hat{W}}$), *vTemp* will be identical to *W*. We will consider all elements of *vTemp* and $\hat{v}Temp$ at Phase I ($vTemp_j$ and $\hat{v}Temp_j$, for $1 \le j \le m$+1).

i. If $vTemp_j = v_1$ at Phase I, then $\hat{v}Temp_j = m$+1, based on (8). At Phase III, $\hat{v}Temp_{p_2} = \hat{W}_{j_2} = m$+1, based on (9). Therefore, $vTemp_{p_2} = m$+1, based on *Lemma 4*.

$$\text{Hence, } \text{vTemp}_{p_2} = \text{W}_{p_2} \quad (10)$$

ii. If $vTemp_j = m$+1 at Phase I, then $\hat{v}Temp_j = v_1$, based on (8). At Phase III, $\hat{v}Temp_{p_1} = \hat{W}_{j_1} = v_1$, based on (9). Therefore, $vTemp_{p_1} = v_1$, based on *Lemma 4*.

$$\text{Hence, } \text{vTemp}_{p_1} = \text{W}_{p_1} \quad (11)$$

iii. If $vTemp_j \neq v_1$ or m+1 at Phase I, then$vTemp_j = \hat{v}Temp_j$, based on (8). Thus, at Phase III, $vTemp_k = \hat{v}Temp_k$, based on *Lemma 4*, when $\hat{v}Temp_k \neq v_1$ or m+1 ($1 \le k \le$ m+1). Therefore, $vTemp_k = \hat{v}Temp_k$, when k ≠ *v*1 or m+1, based on (9).

$$\text{Hence, } vTemp_k = W_k, \text{ when } k \neq v_1 \text{ or } m+1 \quad (12)$$

Based on (10), (11) and (12), *vTemp* is identical to *W* at Phase III.

Hence, $K^{\hat{V}\to\hat{W}}$ permutes *V* into *W*.

To conclude this sub-section: given two permutation vectors *V* and *W* of size m+1, where $V_{p_1} = W_{p_2} = m$+1 and $p_1 < p_2$, we constructed two new vectors $\hat{V}$ and $\hat{W}$ of size m+1, where $\hat{V}_{j_1} = \hat{W}_{j_2} = m$+1 and $j_1 > j_2$. Based on (*case (a)*), there are 2^m keys that permute $\hat{V}$ into $\hat{W}$. Ultimately, we proved that these keys also permute *V* into *W* (see example in Figure 16).

Then, we finished the proof by induction for the case of ($p_1 < p_2$).

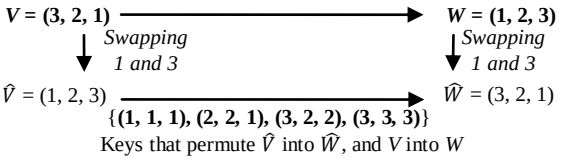


Figure 16: Inferring a set of four keys that permute (3, 2, 1) to (1, 2, 3)

(c) Case of $p_1 = p_2$

Lemma 5: (*Equilibrium Principle #2*) Given two permutation vectors *V* and *W*, where $V_{p_1} = W_{p_1} = m$+1, there exists an index $j \neq p_1$ such that $j \ge IMAP_{V,W}(j)$. In other words, since the element m+1 stays in position p_1 when permuting *V* into *W*, there should be another element that either stays still or moves backward from position *j* in *V* into position $IMAP_{V,W}(j)$ in *W* (Figure 17).

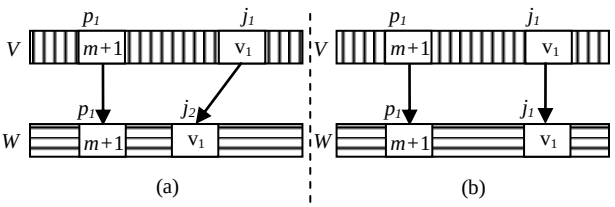


Figure 17: The second equilibrium principle: (a) there exists an element that moves backward (b) there exists an element that stays still

Proof: By contradiction. Suppose that $\forall j \neq p_1$ $j < IMAP_{V,W}(j)$. So, if $j > p_1$, then $IMAP_{V,W}(j) > p_1$+1. Therefore $IMAP_{V,W}\{p_1+1, \ldots, m+1\} \subseteq \{p_1+2, \ldots, m+1\}$, which is a contradiction based on *Lemma* 2, since $|\{p_1+1, \ldots, m+1\}| > |\{p_1+2, \ldots, m+1\}|$.

Notice that *Lemma* 5 is another version of *Lemma* 3, but it is not necessary to find an element moving backwards. Therefore, two cases are to be discussed:

1. There exists $j_1 \neq p_1$ where $j_1 > IMAP_{V,W}(j_1) = j_2$. Then, following the same steps in *case (b)* ($p_1 < p_2$), we can infer 2^m keys that permute *V* into *W* (see example in Figure 18).

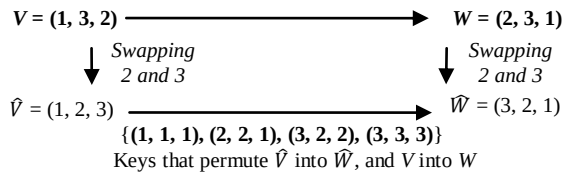


Figure 18: Inferring a set of four keys that permute (1, 3, 2) to (2, 3, 1)

2. For any index j, $j = IMAP_{V,W}(j)$. This means that V and W are identical. Following the same steps of constructing keys in *case (a)*, we will end up with 2^{m-1} keys that permute V into W (or V). We will choose a key $K^{V\to W}$ for comparison purposes.

We need to find another set of 2^{m-1} keys that permute V into W. Notice that we cannot use the same methodology of *case (a)*, because the argument presented to prove that the new set of keys is different than the one constructed at *case (a)* is not valid in case $p_1 = p_2$.

Instead, we will try to come up with a second pair of identical permutation vectors $\hat{V}$ and $\hat{W}$, and we will show that the keys that permute $\hat{V}$ into $\hat{W}$ also permute V into W.

2.1 if $p_1 \neq 1$:

We will construct a permutation vector $\hat{V}$ (also identical to $\hat{W}$) of size m+1 as follows:

$$\hat{V}_i = \begin{cases} V_i, & if\ i \neq p_1, p_1 - 1 \\ V_{p_1}, & if\ i = p_1 - 1 \\ V_{p_1-1}, & if\ i = p_1 \end{cases}$$

In other words, we perform $swap(V_{p_1}, V_{p_1-1})$ in V to get $\hat{V}$ (also $swap(W_{p_1}, W_{p_1-1})$ in W to get $\hat{W}$). Therefore, $\hat{V}$ and $\hat{W}$ are also two identical permutation vectors of size m+1. Following the same steps for constructing the keys in *case (a)*, we obtain 2^{m-1} keys that permute $\hat{V}$ into $\hat{W}$. Since $Reduction(V) = Reduction(W) = Reduction(\hat{V}) = Reduction(\hat{W}) = X = Y$, then the 2^{m-1} set of keys that permute V into W, and the 2^{m-1} set of keys that permute $\hat{V}$ into $\hat{W}$ are constructed based on the same 2^{m-1} set of keys that permute X into Z (Z is the modification of Y based on (1)). For comparison purposes, we will choose a key, $K^{\hat{V}\to\hat{W}}$, from the 2^{m-1} set of keys that permute $\hat{V}$ into $\hat{W}$, such that $K^{\hat{V}\to\hat{W}}$ and $K^{V\to W}$ are constructed using the same key, $K^{X\to Z}$, that permutes X into Z (based on (2), (3), and (4)). Following the same methodology in *case (b)*, we can prove that $K^{\hat{V}\to\hat{W}}$ also permutes V into W.

Unfortunately, $K^{V\to W}$ and $K^{\hat{V}\to\hat{W}}$ are not always different, which prevents the union of 2^{m-1} sets of keys that permute V into W, and 2^{m-1} sets of keys that permute $\hat{V}$ into $\hat{W}$, to achieve 2^m different keys.

i. If $K^{V\to W}_{p_1-1} \neq p_1$-1, then $K^{V\to W}$ and $K^{\hat{V}\to\hat{W}}$ are different, since $K^{\hat{V}\to\hat{W}}_{p_1-1} = p_1$-1.

ii. If $K^{V\to W}_{p_1-1} = p_1$-1, then $K^{V\to W}$ and $K^{\hat{V}\to\hat{W}}$ may be identical, since $K^{\hat{V}\to\hat{W}}_{p_1-1} = p_1$-1. In this case, $Permute(V, K^{V\to W})$ will be idle at step p_1-1; $swap(V_{p_1-1}, V_{p_1-1})$. Also, since $K^{V\to W}_{p_1} = p_1$, $Permute(V, K^{V\to W})$ will be idle at step p_1; $swap(V_{p_1}, V_{p_1})$. Since these two swaps are adjacent, then we can replace them by two new swaps: $swap(V_{p_1-1}, V_{p_1})$ and $swap(V_{p_1}, V_{p_1-1})$. In other words, instead of two idle swaps, we will swap the elements twice at position p_1 (which is m+1) and position p_1-1, which will cancel the swap effect. Based on the latter, we can modify the key $K^{\hat{V}\to\hat{W}}$ to be:

$$K_i^{\hat{V}\to\hat{W}} = \begin{cases} K_i^{V\to W}, & if\ i \neq p_1, p_1 - 1 \\ p_1, & if\ i = p_1 - 1 \\ p_1 - 1, & if\ i = p_1 \end{cases} \quad (13)$$

Now, $K^{V\to W}$ has a different corresponding key $K^{\hat{V}\to\hat{W}}$ that also permutes V into W. Hence, we infer a second, different set of 2^{m-1} keys that permute V into W, when V and W are identical (see example in Figure 19).

2.2 if p_1=1:

Following the same methodology of Section 2.1, we can proceed in the proof by setting $\hat{V}$ as follows (using p_1+1 = 2 instead of p_1-1):

$$\hat{V}_i = \begin{cases} V_i, & if\ i \neq 1, or\ 2 \\ V_1 = m + 1, & if\ i = 2 \\ V_2, & if\ i = 1 \end{cases}$$

Hence, we finish the proof by induction for the case of ($p_1 = p_2$).

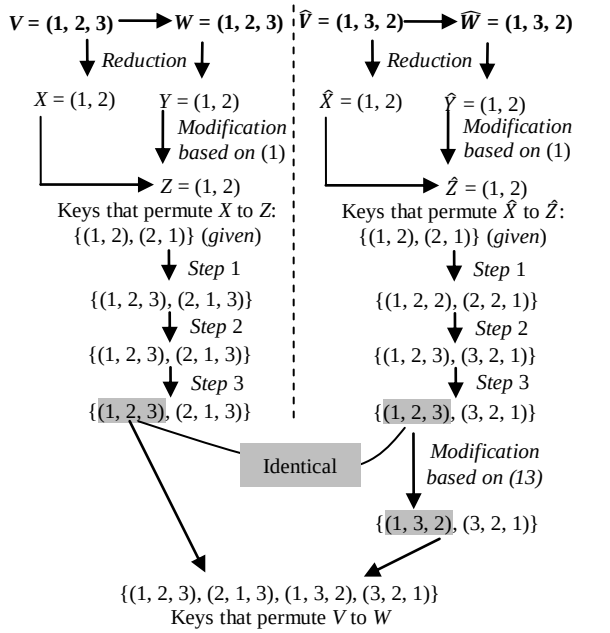


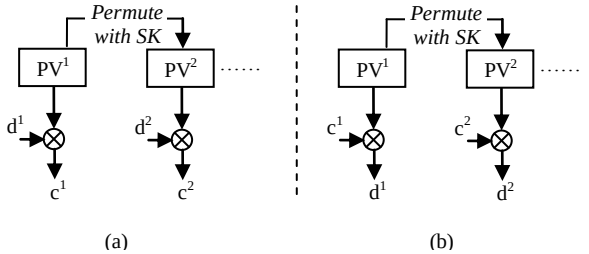
Figure 19: Constructing a set of four keys that permute (1, 2, 3) to (1, 2, 3)

To conclude this subsection: given two permutation vectors V and W of size m+1, where $V_{p_1} = W_{p_1} = m$+1, we constructed 2^m-1 keys that permute V into W based on *case (a)*, and inferred a second different set of 2^{m-1} keys

that also permutes *V* into *W*, based on *case (b)*.

VI. Security Model: Synchronous Dynamic Encryption System (SDES)

SDES is a stream cipher cryptosystem based on permutation vector generation [16][17][18]. The encryption function is simplified in order to minimize the overhead cost. Thus, a simple XOR is performed between the data record d^j and one of the generated permutation vectors PV^j, resulting in a cipher c^j to be transmitted (Figure 20 (a)). The decryption function is performed in the same manner as the encryption function. The cipher record c^j is XORed with the same permutation vector PV^j (generated at the recipient side), producing the original data record d^j (Figure 20 (b)). Then, a new permutation vector is generated ($PV^{j+1} = Permute(PV^j, SK)$) to be used in the next encryption/decryption operations. Notice that the first permutation vector to be used in encryption PV^1 is the result of Permute(PV^0, *SK*), where PV^0 is the initial



permutation vector (1, 2, …, *n*), and *n* is the key size.

Figure 20: (a) Encryption at the sender side (b) Decryption at the recipient side.

A. Static shared key

This option provides a low security profile; compromising two consecutive *PV*s will break the entire system, since the corresponding *IMAP* between PV^j and PV^{j+1} is identical to the *IMAP* of PV^{j+1} and PV^{j+2}. Therefore, this option is used only with the assumption that a cryptanalysis is unfeasible (e.g., transmitted data are proportionally limited).

B. Stream of shared keys

In order to alleviate the previous security loophole, a second option is also provided to modify the shared key after each data record encryption, for more *IMAP* dynamics. Practically, we perform $SK^{j+1} \leftarrow SK^j + PV^j$ before generating the next permutation vector PV^{j+1}. Then, the shared key generation is not vulnerable to "biased byte" analysis since the involved permutation vector is a good source of byte diversity. However, in the event that more than one encryption session is opened (in parallel or at different times), the same stream of shared keys is generated, lacking security independence between sessions; i.e., breaking one session breaks all.

C. Session-based stream of shared keys

For ultimate security, the communicated data is involved in the shared key generation, as a third option. Basically, we perform $SK^{j+1} \leftarrow SK^j + PV^j + d^j$ before generating the next permutation vector, PV^{j+1}. Hence, a different sequence of shared keys is generated in every session (assuming that sessions are of different communication data).

VII. Experimental Results

A prototype simulation of our technique proved to run two to three times faster than the-state-of-the-art AES (Advanced Encryption Standard), DES (Data Encryption Standard), and Triple DES (Figure 21). Using the NS2 simulator (version 2.26), we designed a topology of five nodes connected to a router that routes packets to a sink node through a duplex connection of 1 Mbits/s maximum capacity. Each of the four nodes tries to send an exponential generated traffic data to the sink passing through the bottleneck 1Mbits/s connection. The first node sends non-encrypted packets. The second, the third, the fourth and the fifth nodes send packets securely, encrypted with AES, DES, 3DES, and SDES, respectively.

Figure 21 shows that SDES (using the dynamic stream of shared keys option) achieves a maximum throughput of 896 Kbits/s. This result proves the higher efficiency of our SDES algorithm compared to other peer techniques (AES: 409 Kbps, DES: 528 Kbps, and Triple DES: 112 Kbps). The reason for achieving such better performance is the simplicity of our encryption/ decryption function, since the function complexity is shifted to the dynamics of the key management (i.e., the permutation vectors generation).

VIII. Conclusion

In this paper, a simple mechanism to generate permutation vectors (based on a random secret key) is introduced for data encryption. Unique to our technique, we proved that 2^{m-1} different secret keys have the same effect on the generation of the next encryption key. Hence, even if an intruder (hypothetically) compromises two consecutive encryption keys, he is cornered to brute-force a massively huge key space (for $m = 256$). Moreover, the involvement of the entire permutation vector in the encryption process results in a much more diverse stream of keys than those of RC4, avoiding state-related attacks.

We also presented a cryptosystem implementation that utilizes permutation vectors in the process of encryption as well as in key management. Experimental results (using the NS2 simulator) showed that our security system outperformed peer security mechanisms, e.g., AES and Triple DES, due to the simplicity of both encryption and key management functions.

Future work is related to the diversity of *IMAP*s between consecutive encryption keys. On average, there is a chance of $1/n!$ for two subsequent secret keys to yield the same *IMAP*. Hence, it would be useful to investigate mechanisms to enhance secret key management in order to assert such diversity.

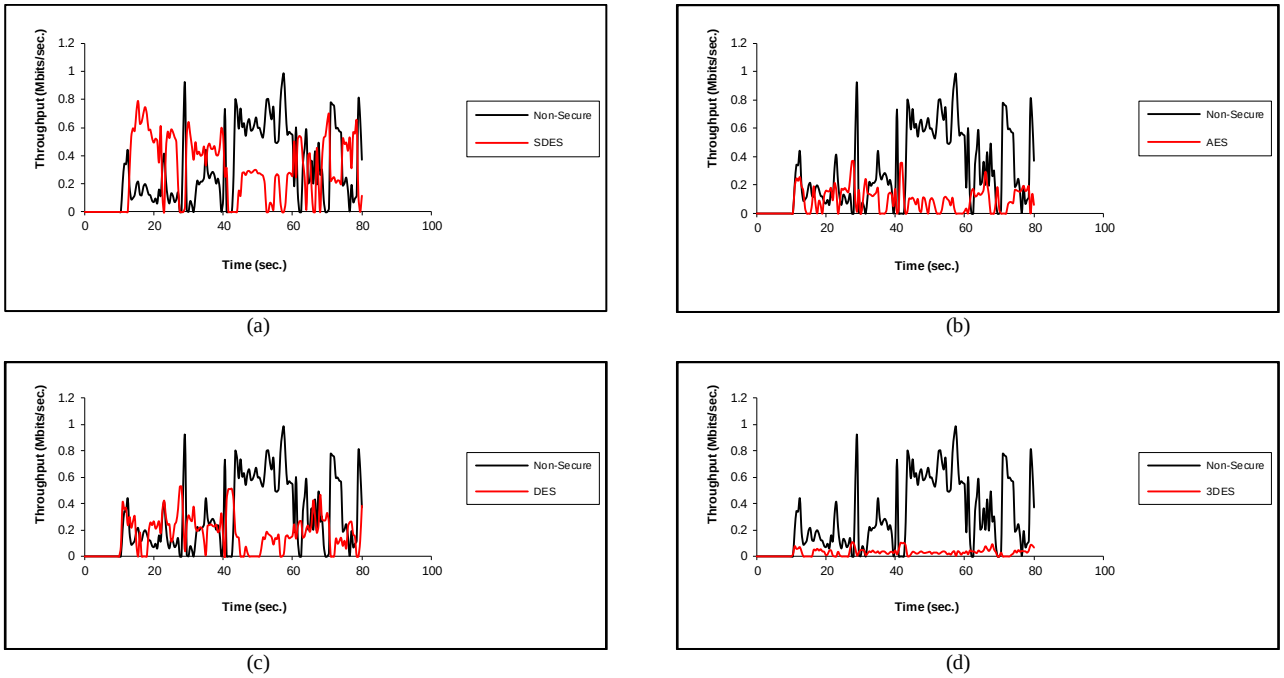


Figure 21: NS2 experimental results showing network throughput using secure transmission modes: (a) SDES (b) AES (c) DES (d) 3DES

REFERENCES

[1] D. Knuth, "The Art of Computer Programming, Volume 4: Generating All Tuples and Permutations," Addison-Wesley, 2005.

[2] A. Nijenhuis and H. Wilf, "Combinatorial Algorithms for Computers and Calculators," Academic Press, Orlando FL, second edition, 1978.

[3] S. Skiena, "Implementing Discrete Mathematics," Addison-Wesley, Redwood City, CA, 1990.

[4] D. Stinson. "Cryptography: Theory and Practice." Boca Raton, CRC Press, LLC, 1995.

[5] National Bureau of Standards. "Data Encryption Standard." NBS FIPS Publication 46, Jan. 1977.

[6] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "The Twofish Encryption Algorithm: A 128-Bit Block Cipher," John Wiley and Sons, 1999.

[7] E. Biham, R. Anderson and L. Knudsen, "Serpent: A New Block Cipher Proposal," Proc. 5th International Workshop on Fast Software Encryption, Paris, France, Mar 1998, pp. 222-238.

[8] J. P. McGregor and R. B. Lee. "Architectural Enhancements for Fast Subword Permutations with Repetitions in Cryptographic Applications," The International Conference on Computer Design: VLSI in Computers and Processors, 2001.

[9] R. Wash, "Lecture Notes on Stream Ciphers and RC4," working paper, Sep. 2001.

[10] B. Schneier. "Applied Cryptography," John Wiley and Sons, Second Edition, 1996, pp 397-400.

[11] A. Menezes, P. Van Oorschot and S. Vanstone, "Handbook of Applied Cryptography," CRC Press, Inc., 1997.

[12] S Fluhrer, I. Mantin, and A. Shamir. "Weaknesses in the Key Scheduling Algorithm of RC4," Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, Aug. 2001.

[13] G. Gong, K. C. Gupta, M. Hell, and Y. Nawaz, "Towards a General RC4-Like Keystream Generator," Proc. 1st SKLOIS conference on Information Security and Cryptology, Beijing, China, Dec. 2005, pp 162-174.

[14] S. Fluhrer and D. McGrew, "Statistical Analysis of the Alleged RC4 Key Stream Generator," Proc. 7th International Workshop on Fast Software Encryption, New York, NY, USA, Apr. 2000, pp 19-30.

[15] I. Mantin and A. Shamir, "A Practical Attack on Broadcast RC4," Proc. 8th International Workshop on Fast Software Encryption, Yokohama, Japan, Apr. 2001, pp 152-164.

[16] H. S. Soliman and M. Omari, "New Design Strategy of Dynamic Security Implementation," IEEE Globecom 2004 Workshop on Adaptive Wireless Networks, Dallas, TX, Dec. 2004.

[17] H. S. Soliman and M. Omari, "An Efficient Application of a Dynamic Crypto System in Mobile Wireless Security," IEEE Wireless Communications and Networking Conference, Atlanta, Georgia, Mar. 2004.

[18] H. S. Soliman and M. Omari, "Application of Synchronous Dynamic Encryption System (SDES) in Wireless Sensor Networks," International Journal of Network Security, vol 3, no. 2, 2006, pp 160-171.

Dr. Mohammed Omari received his B.E degree from the University of Es-Senia Oran (Algeria) in 1995 and the M.S. degree from New Mexico Tech (New Mexico, USA) in 2002, as well as the Ph.D. degree in 2005. His major is in computer science. He is currently an associate professor at the computer science department, and a unit chair at the LDDI laboratory, University of Adrar, Algeria. His research interests include network security, cryptography, sensors and ad hoc networks protocols, image processing, neural networks, and genetic algorithms.

Prof. Hamdy S. Soliman Received his B.S. in 1978, from Alexandria University (Egypt), and his M.S. in 1983 from Florida Institute of Technology, and his Ph.D. 1989 from New Mexico State University (USA), all degrees are in Computer Science. He is currently a full Professor in the Computer Science & Engineering Department at New Mexico Tech (NMT). His research interests include Wireless Mobile sensor networks routing and security protocols, neural networks application in data/image processing, all-fiber optics networks admission control, computer and networks security, wireless security and protocols.

