VMPC One-Way Function

Bartosz Zoltak, bzoltak@vmpcfunction.com; bzoltak@wp.pl

Abstract. The VMPC function is a combination of two basic operations – permutation composition and integer addition. The function resulting from this combination shows to have very high resistance to inverting. Computational effort of about 2^{260} operations is estimated to be required to invert the VMPC function. The value of the function can be computed with 3 elementary computer processor instructions per byte. An open question is whether the function's simplicity raises a realistic chance that the lower bound on the complexity of inverting it might be proved.

1. Introduction

VMPC is an abbreviation of Variably Modified Permutation Composition.

The VMPC function is a combination of triple permutation composition and integer addition. It differs from a simple triple permutation composition with one integer addition operation performed on some of the elements of the permutation. The consequence of this addition operation is corruption of cycle structure of the transformed permutation - the fundamental source of the function's resistance to inverting.

The VMPC function has a simple formal definition and the value of the function can be computed with 3 one-clock-cycle instructions of an Intel 80486 and newer or compatible processor per byte.

Inverting the function by the fastest known inverting algorithm is estimated to require an average computational effort of about 2^{260} operations. This effort can be significantly extended at only linear cost.

Adding one more one-cycle instruction to the implementation of the function increases its resistance to inverting to an average level of about 2^{420} operations. Adding another one-cycle instruction raises it to about 2^{550} operations and adding yet another one-cycle instruction produces a function requiring an average computational effort of about 2^{660} operations to be inverted.

The simplicity of the VMPC function could raise a question whether it might be possible to prove the lower bound on the complexity of inverting it. This currently is an open problem and a possible subject of future research.

The more detailed explanation of why the VMPC function it hard to invert is to be found in section 6.

2. Definition of the VMPC function

Notation:

n, P, Q : P and Q: n-element permutations. For simplicity of further implementations P and Q are one-to-one mappings $A \rightarrow A$, where $A = \{0,1,\ldots,n-1\}$

k : Level of the function; $k<n$

\+ : addition modulo n

Definition:

A k-level VMPC function, referred to as $VMPC_k$, is such transformation of P into Q, where

$$\mathbf{Q[\,x\,] = P\,[\,P_k\,[\,P_{k-1}\,[\ldots[\,P_1\,[\,P\,[\,x\,]\,]\,]\,\ldots]\,]\,],}$$

$x \in \{0,1,\ldots,n-1\}$,

P_i is an n-element permutation such that $P_i[x] = f_i(P[x])$, where f_i is any function such that $P_i[x] \neq P[x] \neq P_j[x]$ for $i \in \{1\ldots k\}$, $j \in \{1\ldots k\} / \{ i \}$.

For simplicity of further implementations f_i is assumed to be $f_i(x) = x + i$

For simplicity of future references notation Q=VMPC(P) is assumed to be equivalent to $Q=VMPC_1(P)$

Example:

$Q=VMPC_1(P)$ is such transformation of P into Q, where:

$Q[\,x\,] = P[P_1[P[x]]]$,

$P_1[x]=P[x]+1$.

(Q[x]=P[P[P[x]]+1], where "+" denotes addition modulo n)

Table 1. Definitions of 1,2,3 and 4-level VMPC function

Function	Definition
$VMPC_1$	Q[x]=P[P[P[x]]+1]
$VMPC_2$	Q[x]=P[P[P[P[x]]+1]+2]
$VMPC_3$	Q[x]=P[P[P[P[P[x]]+1]+2]+3]
$VMPC_4$	Q[x]=P[P[P[P[P[P[x]]+1]+2]+3]+4]

3. The 3-instruction implementation of the VMPC function

Implementation of a 1-level VMPC function, where Q[x] = P[P[P[x]]+1], for 256-element permutations P and Q in assembly language is described.

Assume that :

- Pa is a 257-byte array indexed by numbers from 0 to 256, the P permutation is stored in the array at indexes from 0 to 255 (Pa[0...255]=P) and that Pa[256]=Pa[0].

- the EAX 32-bit register stores value zero. ("AL" denotes 8 least significant bits of EAX)

- the EDX 32-bit register stores an address, where the Pa array is stored in memory

- the ECX 32-bit register specifies which element of the Q permutation to compute

Execute:

Table 2. Implementation of 1-level VMPC function

Instruction	Description
MOV AL, [EDX] + ECX	Store ECX-th element of P in AL
MOV AL, [EDX] + EAX	Store AL-th element of P in AL
MOV AL, [EDX] + EAX + 1	Store (AL+1)-th element of P in AL

The 3 MOV instructions in Table 2 store the ECX-th element of permutation Q, where $Q=VMPC_1(P)$, in the AL (and EAX) register.

4. Example values of the VMPC function

Values of the 1,2,3 and 4-level VMPC function of an example 10-element permutation P are shown in Table 3.

Table 3. Example values of the VMPC function for 10-element permutations

Index	0	1	2	3	4	5	6	7	8	9
P	2	0	4	3	6	9	7	8	5	1
$Q_1=VMPC_1(P)$	9	3	8	6	5	4	1	7	2	0
$Q_2=VMPC_2(P)$	0	9	2	5	8	7	3	1	6	4
$Q_3=VMPC_3(P)$	3	4	9	5	0	2	7	6	1	8
$Q_4=VMPC_4(P)$	8	5	3	1	6	7	0	2	9	4

5. Complexities of computing / inverting the VMPC function

Efforts required to compute and to invert the 1,2,3 and 4-level VMPC function for 256-element permutations by the fastest known inverting algorithm are shown in Table 4:

Table 4. Complexities of computing / inverting the VMPC function for 256-element permutations

Function	$VMPC_1$	$VMPC_2$	$VMPC_3$	$VMPC_4$
Number of MOV instructions required to compute one byte of the value of the function	3	4	5	6
Estimated average number of computational operations required to invert the function	2^{260}	2^{420}	2^{550}	2^{660}

6. Difficulty of inverting the VMPC function

n-element permutation P has to be recovered given information from n-element permutation, Q, where $Q=VMPC_k(P)$ (e.g. n=256, k=1: Q[x]=P[P[P[x]]+1]).

By definition each element of Q is formed by k+2 (e.g. 3), usually different, elements of P. One element of Q (e.g. Q[33]=25) can be formed by many possible configurations of P elements (e.g. P[33]=10, P[10]=20, P[21]=25 or P[33]=1, P[1]=4, P[5]=25, etc.).

It cannot be said which of the configurations is more probable. One of the configurations has to be picked (usually k+1 (e.g. 2) elements of P have to be guessed) and the choice must be verified using all those other Q elements, which use at least one of the P elements from the picked configuration.

Each element of P is usually used to form k+2 (e.g. 3) different elements of Q. As a result, usually (k+2)*(k+1) (e.g. 6) new elements of Q need to be inverted (all k+2 elements of P used to form each of those Q elements need to be revealed) to verify the P elements from the picked configuration.

This would not be difficult for a simple (e.g. triple) permutation composition, where the cycle structure of P is retained by Q (some cycles are only shortened).

In Variably Modified Permutation Composition however the cycle structure of P is corrupted by the addition operation(s) and cannot be easily recovered from Q.

Due to that it is usually impossible to find two different elements of Q, which use at least k+1 (e.g. 2) exactly the same elements of P. (This can be done easily for a simple permutation composition)

In fact only such element of Q can usually be found, name it Q[r], which uses only one of the k+2 (e.g. 3) elements of P, used to form another Q element. This forces the k remaining (e.g. 1) elements of P, used to form Q[r], to be guessed to make the verification of the initial pick possible.

However at each new guessed element of P, there usually occur k+1 (e.g. 2) new elements of Q which use this element of P and which need to be inverted to verify the guess.

The algorithm falls into a loop, where at every step usually k (e.g. 1) new elements of P need to be guessed to verify the previously guessed elements. It quickly occurs that the k+2 (e.g. 3) elements of P picked at the beginning of the process indirectly depend on all n (e.g. 256) elements of Q.

The described scenario is the case usually and it is sometimes possible to benefit from coincidences (where for example it is possible to find two elements of Q, which use more than one (e.g. 2) exactly the same P elements (e.g. Q[2]=3: P[2]=4, P[4]=8, P[9]=3 and Q[5]=8: P[5]=9, P[9]=3, P[4]=8)).

The actual algorithm of inverting VMPC was optimized to benefit from the possible coincidences. The average number of P elements which need to be guessed - for n=256 - has been reduced to only about 34 for 1-level VMPC function, to about 57 for 2-level VMPC, to about 77 for 3-level VMPC and to about 92 for 4-level VMPC function.

Searching through half of the possible states of these P elements takes on average about 2^{260} steps for 1-level VMPC function, about 2^{420} for 2-level VMPC, about 2^{550} for 3-level VMPC and about 2^{660} steps for 4-level VMPC function.

7. Algorithm of inverting the VMPC function

The fastest method of inverting the VMPC function found derives n-element permutation P which produces a given $Q=VMPC_k(P)$ permutation, according to the following algorithm:

Notation:

P_t : n-element table, the searched permutation will be stored in

Argument, Value; Base, Parameter of an element of P_t :

For an element $P_t[x]=y$: x is termed the argument and y the value.

The base is either the argument or the value; the parameter is the corresponding – the value or the argument.

Example: for an element $P_t[3]=5$: If value 5 is the base, argument 3 is the parameter.

1.1) Reveal one element of P_t by assuming $P_t[x]=y$; where x and y are any values within range $x \in \{0,1,\ldots,n-1\}$, $y \in \{0,1,\ldots,n-1\}$

1.2) Choose at random whether x is the base and y the parameter or y the base and x the parameter of the element $P_t[x]=y$. Denote $P_t[x]=y$ as the current element of P_t.

2) Reveal all possible elements of P_t by running the deducing process (see section 7.1)

3) If n elements of P_t have been revealed with no contradiction in step 2:
Terminate the algorithm and output P_t

4) If fewer than n elements of P_t have been revealed with no contradiction in step 2:

4.1) Reveal a new element of P_t by running the selecting process (see section 7.2). Denote the revealed element as the current element of P_t.

4.2) Save the parameter of the current element of P_t

4.3) Go to step 2

5) If a contradiction occurred in step 2:

5.1) Remove all elements of P_t revealed in step 2 when the current element of P_t had been revealed

5.2) Increment modulo n the parameter of the current element of P_t

5.3) If the parameter of the current element of P_t has returned to the value saved in step 4.2:

5.3.1) Remove the current element of P_t

5.3.2) Denote the element, which had been the current element of P_t directly before the element removed in step 5.3.1 became the current one, as the current element of P_t

5.3.3) Go to step 5.1

6) Go to step 2

7.1. The deducing process

The deducing process reveals all possible elements of P_t, given Q and given the already revealed elements of P_t according to the following algorithm:

Notation: as in section 7, with:

C,A : temporary variables
Word x of Statement y:
Statement y: A set of all elements of P_t used to calculate Q[y]
Word x: x-th consecutive element of P_t used to calculate Q[y]:
Example for $VMPC_2$: Q[x] = P[P[P[P[x]]+1]+2] :
Assume $P_t[2]=3$, $P_t[3]=5$, $P_t[6]=2$, $P_t[4]=7$, which produces Q[2]=7.
The elements $P_t[2]=3$, $P_t[3]=5$, $P_t[6]=2$, $P_t[4]=7$ form statement 2.
The element $P_t[2]=3$ is word 1 of statement 2; $P_t[3]=5$ is
word 2 of statement 2, etc.

1.1) Set C to 0
1.2) Set A to 0

2) If the element P_t [A] is revealed:

2.1) If the element P_t [A] and k other revealed elements of P_t fit a general pattern of k+1 words of any statement : Deduce the remaining word of that statement (see example 7.1.1)

2.2) If the deduced word is not a revealed element of P_t :

2.2.1) Reveal the deduced word as an element of P_t

2.2.2) Set C to 1

2.3) If the deduced word contradicts any of the already revealed elements of P_t :
Output a contradiction and terminate the deducing algorithm (see example 7.1.2)

3.1) Increment A
3.2) If A is lower than n: Go to step 2
3.3) If C is equal 1: Go to step 1.1

Example 7.1.1)

For $VMPC_2$: Q[x] = P[P[P[P[x]]+1]+2]:

Assume that Q[0]=9 and that the following elements of P_t are revealed:
$P_t[0]=1$, $P_t[1]=3$, $P_t[8]=9$

Word 3 of statement 0 can be deduced as $P_t'[4]=6$ ($P_t'[3+1]=8-2$)

Example 7.1.2)

For $VMPC_2$: Q[x] = P[P[P[P[x]]+1]+2]:

Assume that Q[7]=2 and that the following elements of P_t are revealed:
$P_t[1]=8$, $P_t[9]=3$, $P_t[5]=2$ and $P_t[6]=1$

Word 1 of statement 7, deduced as $P_t'[7]=1$, contradicts the already revealed element $P_t[6]=1$.

7.2. The selecting process

The selecting process selects such new element of P_t to be revealed which maximizes the number of elements of P_t possible to deduce in further steps of the inverting algorithm.

The selecting process outputs a selected base and a randomly chosen parameter of a new element of P_t.

Notation: as in section 7.1, with:

G,R,X,Y : temporary variables

Ta,Tv : temporary tables

Weight : table of numbers:

Weight[1; 2; 3; 4] = (2; 5; 9; 14); *Example*: Weight[3]=9

1.1) Set Ta and Tv to 0
1.2) Set G to 0
1.3) Set R to 1

2) Count the number of revealed elements of P_t which fit the general pattern of words of a statement in which an unrevealed element of P_t with argument G would be word R. Increment Ta[G] by Weight of this number (see example 7.2.1)

3) Count the number of revealed elements of P_t which fit the general pattern of words of a statement in which an unrevealed element of P_t with value G would be word R. Increment Tv[G] by Weight of this number

4.1) Increment R
4.2) If R is lower than k+3: Go to step 2

4.3) Increment G
4.4) If G is lower than n: Go to step 1.3

5.1) Pick any index of Ta or Tv for which the number stored in any of the tables Ta or Tv is maximal (see example 7.2.2)

5.2) If the index picked in step 5.1 is an index of Ta:

5.2.1) Store this index in variable X

5.2.2) Generate a random number Y within range $Y \in \{0,1,\ldots,n-1\}$, such that an element of P_t with value Y is not revealed

5.2.3) Output $P_t'[X]=Y$, where X is the base and Y is the parameter

5.3) If the index picked in step 5.1 is an index of Tv:

5.3.1) Store this index in variable Y

5.3.2) Generate a random number X within range $X \in \{0,1,\ldots,n-1\}$, such that an element of P_t with argument X is not revealed

5.3.3) Outputs $P_t'[X]=Y$, where Y is the base and X is the parameter

Example 7.2.1)

For $VMPC_2$: Q[x] = P[P[P[P[x]]+1]+2]:
Assume that G=8; R=2; Q[6]=1 and that the following elements of P_t are revealed:
$P_t[6]=8$, $P_t[5]=1$

There are two revealed elements of P_t which fit the general pattern of words of a statement in which $P_t[8]$ would be word 2 : $P_t[6]=8$, $P_t[5]=1$:

word 1	word 2	word 3	word 4
$P_t[6]=8$,	$P_t[8]=?$,	$P_t[?]=?$,	$P_t[5]=1$

Ta[8] = Ta[8] + Weight[2] = Ta[8] + 5

Example 7.2.2)

Assume Ta = (0, 5, 2, 7, 5, 7) and Tv = (2, 7, 0, 0, 5, 2)
The maximal number stored in any of the tables is 7: Ta[3]=Ta[5]=Tv[1]=7
Pick any of: index 3 of Ta, index 5 of Ta or index 1 of Tv

8. Example complexities of inverting the VMPC function

Complexity of inverting the VMPC function has been approximated as an average number of times the deducing process in step 2 of the inverting algorithm described in section 7 has to be run until permutation P_t satisfies $Q=VMPC_k(P_t)$.
Average numbers of elements of P_t which need to be assumed are given in brackets in Table 5.

Complexities of inverting the VMPC function of the following levels have been approximated:

$VMPC_1$: Q[x] = P[P[P[x]]+1]
$VMPC_2$: Q[x] = P[P[P[P[x]]+1]+2]
$VMPC_3$: Q[x] = P[P[P[P[P[x]]+1]+2]+3]
$VMPC_4$: Q[x] = P[P[P[P[P[P[x]]+1]+2]+3]+4]

Table 5. Example complexities of inverting the VMPC function

Function / n	$\mathbf{VMPC_1}$	$\mathbf{VMPC_2}$	$\mathbf{VMPC_3}$	$\mathbf{VMPC_4}$
6	$\mathbf{2^{4,1}}$ (2,3)	$\mathbf{2^{5,5}}$ (3,1)	$\mathbf{2^{6,1}}$ (3,3)	$\mathbf{2^{6,9}}$ (3,8)
8	$\mathbf{2^{5,5}}$ (2,7)	$\mathbf{2^{7,5}}$ (3,4)	$\mathbf{2^{8,8}}$ (4,0)	$\mathbf{2^{9,8}}$ (4,4)
10	$\mathbf{2^{7,1}}$ (3,0)	$\mathbf{2^{9,7}}$ (4,0)	$\mathbf{2^{11,5}}$ (4,7)	$\mathbf{2^{13,0}}$ (5,2)
16	$\mathbf{2^{11,5}}$ (3,8)	$\mathbf{2^{16,6}}$ (5,4)	$\mathbf{2^{20,4}}$ (6,6)	$\mathbf{2^{23,3}}$ (7,5)
32	$\mathbf{2^{24}}$ (6,0)	$\mathbf{2^{37}}$ (9,1)	$\mathbf{2^{47}}$ (11,5)	$\mathbf{2^{54}}$ (13,4)
64	$\mathbf{2^{53}}$ (10,2)	$\mathbf{2^{84}}$ (16,2)	$\mathbf{2^{108}}$ (21,0)	$\mathbf{2^{127}}$ (24,9)
128	$\mathbf{2^{117}}$ (18,5)	$\mathbf{2^{190}}$ (30,0)	$\mathbf{2^{245}}$ (40,0)	$\mathbf{2^{292}}$ (47,0)
256	$\mathbf{2^{260}}$ (34,0)	$\mathbf{2^{420}}$ (57,0)	$\mathbf{2^{550}}$ (77,0)	$\mathbf{2^{660}}$ (92,0)

Example: For 1-level VMPC function applied on 256-element permutations about 34 elements of P_t need to be assumed to recover all elements of P_t. Searching through half of the possible states of the 34 assumed elements takes about 2^{260} steps.

9. Conclusions

An idea of a simple one-way function has been presented. The VMPC function's resistance to inverting is strictly related to the addition operation(s) performed at some step(s) of the composition. Their role can be clearly illustrated by comparing the process of inverting a simple permutation composition with inverting the Variably Modified Permutation Composition.

It is an open problem whether the simplicity of the VMPC function helps make a hypothetical attempt to prove the lower bound on the complexity of inverting the function worth undertaking.

A proposed practical application of the VMPC one-way function in a stream cipher is described in "VMPC Stream Cipher" by Bartosz Zoltak (possible to download from http://www.VMPCfunction.com or from http://eprint.iacr.org).

Current developments in the analysis of the VMPC function are to be found at http://www.VMPCfunction.com.