The IDEA Block Cipher

Elena Canali

Partners: Giuseppe Vitto
Rosanna Racanelli
Regina Krisztina Biro
Bertalan Borsos
Jan Helge Wolf
Ogwara Vincent

1 Introduction

The block cipher IDEA (International Data Encryption Algorithm) was designed by X. Lai and J.L. Massey. It was proposed as candidate to replace DES procedure developed in the seventies. The main innovation of this block cipher is the use of operations over different algebraic groups and the absence of S-boxes.
In this report I first briefly describe the algorithm and then I discuss about implementation choices of my *MAGMA* version of IDEA.

2 Description of the Algorithm

IDEA works with plaintexts of 64 bits and keys of 128 bits and output 64-bit ciphertexts. Plaintext and key vectors are partitioned in sub-blocks of 16 bits as operations in rounds operates with 16-bit numbers; the ciphertext vector is the union of four 16-bit sub-blocks.

2.1 Encryption and Decryption algorithm

Encryption and decryption algorithm are exactly the same with the exception that different key sub-blocks are used. The process consists of eight identical encryption steps followed by an output transformation, the ninth round. The structure of the rounds is shown in the following figure.

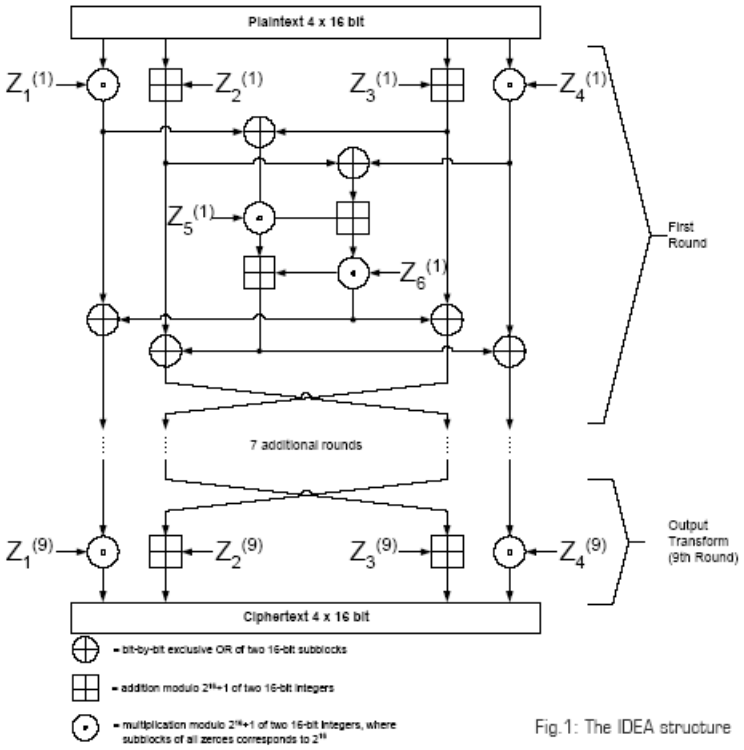

Fig.1: The IDEA structure

As already mentioned, IDEA's algorithm uses operations over three different algebraic groups: bitwise XOR , sum modulo 2^{16} and product modulo $2^{16}+1$.

2.2 Key Schedule

Encryption and decryption phase require 52 16-bit key sub-blocks. For encryption process, the first 8 sub-blocks are obtained by simply dividing the 128-bit vector in input; the others are obtained by shifting the 128-bit vector to the left by 25 positions. The shifts are repeated until the required number of sub-blocks is obtained.
For decryption phase, the 52 sub-blocks are obtained by changing the sub-blocks of the encryption (the elements of K) as shown in the following table

Round 1	$K_1^{(9)-1}, -K_2^{(9)}, -K_3^{(9)}, K_4^{(9)-1}, K_5^{(8)}, K_6^{(8)}$
Round 2	$K_1^{(8)-1}, -K_3^{(8)}, -K_2^{(8)}, K_4^{(8)-1}, K_5^{(7)}, K_6^{(7)}$
Round 3	$K_1^{(7)-1}, -K_3^{(7)}, -K_2^{(7)}, K_4^{(7)-1}, K_5^{(6)}, Z_6^{(6)}$
Round 4	$K_1^{(6)-1}, -K_3^{(6)}, -K_2^{(6)}, K_4^{(6)-1}, K_5^{(5)}, K_6^{(5)}$
Round 5	$K_1^{(5)-1}, -K_3^{(5)}, -K_2^{(5)}, K_4^{(5)-1}, K_5^{(4)}, K_6^{(4)}$
Round 6	$K_1^{(4)-1}, -K_3^{(4)}, -K_2^{(4)}, K_4^{(4)-1}, K_5^{(3)}, K_6^{(3)}$
Round 7	$K_1^{(3)-1}, -K_3^{(3)}, -K_2^{(3)}, K_4^{(3)-1}, K_5^{(2)}, K_6^{(2)}$
Round 8	$K_1^{(2)-1}, -K_3^{(2)}, -K_2^{(2)}, K_4^{(2)-1}, K_5^{(1)}, K_6^{(1)}$
Round 9	$K_1^{(1)-1}, -K_2^{(1)}, -K_3^{(1)}, K_4^{(1)-1}$

3 Implementation

3.1 Encryption

IDEA's encryption steps are collected in a function that I called *IDEA_Encryption*. I designed also some other auxiliary functions that are used during the process. *IDEA_Encryption* takes as input a string of 16 bytes in hexadecimal notation as key and a string of 8 bytes in hexadecimal notation as plaintext. The output is a 8 bytes string in hexadecimal notation.

The inputs are first transformed into integers (using the function *StringToInteger*) and then in sequence (using the function *Intseq*). The rest of the algorithm works with sequences.
When the function *Intseq* is used, the lenght of the resultant sequence could not be the required one. To solve this problems I designed the function *RLengtht* that takes as input a sequence and an integer n and add enough zeros to the sequence (zeros are added to the right) to reach the length n. Moreover, the sequences *MAGMA* work with, have the less significant bit on the right and the most significant bit on the left. Therefore I used the function *Reverse* to reverse the sequence. This two operations could be inverted but in this case I should have added the required zeros to the left. I preferred this choice because during the algorithm I had to use many times the function *RLength* and many times, working on the right allows to save some reverse operation.

Division of sequences in sub-blocks is made by a function that I called *SubBlocks*. This function takes as input a sequence and perform at the same time two operation: division in sub-blocks of 16 bits and reverse of every sub-blocks. *SubBlocks* do not use the *MAGMA* function *Reverse*: it reads instead blocks of 16-bit at the time and save it in sequences. in reversed order (in this way the input sequence is read only once). The output is a sequence of sequences that are the sub-blocks of 16-bit. The reverse operation at this level allows to avoid the use of the function *Reverse* until the end of the encryption process: if I had not compute it, every time I would use the functions *Intseq* or *Seqint*, I should use also *Reverse*.

To perform the key-schedule operations, I designed the function *IDEA_Key* that takes as input the 128-bit sequence corresponding to the key and perform shifts and division in sub-blocks. It output a sequence of 56 sequences: the first 52 are the key sub-blocks used in encryption process, the other 4 are useless. The shifts are performed to the right because I worked in the reversed situation.

The round steps are the same for encryption and decryption phase: for this reason I collected them in a function named *Rounds*. This uses other three functions *ProductSubBlocks*, *SumSubBlocks* and *XorSubBlocks* that take as input two 16-bit sequence and perform, respectively, product modulo $2^{16}+1$, sum modulo 2^{16} and the bitwise XOR. I preferred to use modular operations instead of define different rings and then cast over them for efficiency reason (next section deals with this issue). The first two functions use *Seqint* and *Intseq* to switch sequences with integer and vice versa. In the function *ProductSubBlocks*, the dimension of the resultant sequence can exceed 16 bit: in this case, the result is set to the null sequence.
Rounds outputs four sub-blocks that correspond to the ciphertext: I designed the function *UnionSubBlocks* to merge them in a single sequence of 64 bits. I used *MAGMA* functions to turn the sequence into a string in hexadecimal notation, padded with zeros if needed: this is the output of *IDEA_Encryption*.

3.2 Decryption

The decryption steps are collected in a function that I called *IDEA_Decryption*. This function takes as input a string of 16 bytes in hexadecimal notation as key and a string of 8 bytes in hexadecimal notation as ciphertext. The output is another string of 8 bytes in hexadecimal notation that represents the plaintext.

IDEA_Decryption is very similar to *IDEA_Encryption*. The steps are exactly the same with the exception of the key-schedule process. For this purpose I designed the function *IDEA_InvKey* that takes as input the sequence corresponding to the key and output the 52 key sub-blocks for decryption algorithm. This function recalls the function *IDEA_Key* and modify every sub-block as required in the algorithm. Furthermore, some auxiliary functions are designed to perform needful operations. *InvSubBlocks* execute the multiplicative inverse modulo $2^{16}+1$, using the *MAGMA* function *Modinv*. Another function named *MinusSubBlocks* performs the additive inverse modulo 2^{16}. Like in *ProductSubBlocks*, the length of the resultant sequences can be bigger than 16: in this case the output is a sequence of zeros.

4 Timing

The main differences between the team version and my implementation of IDEA cipher are how the sequence are treated and how operations over the different

algebraic groups are done.

In the team version, the sequences are divided in sub-blocks but the sub-blocks are not reversed. This implies that every time it is used the functions *Intseq* or *Seqint* the 16-bit sequences have to be reversed. I tested my implementation of *IDEA_Encryption* and a modified version, where I changed the use of reverse like in the team version: I iterated 100 and 100 times the function *IDEA_Encryption* starting from a key K="80000000000000000000000000000000" and a plaintext P="0000000000000000". The time of executions are collected in the following table.

	100 iterations	1000 iterations
my implementation	0.156	1.601
modified implementation	0.172	1.729

In the team implementation, operations in rounds are handled with casts over appropriate rings. Moreover, the rings are created every time that the function are used. I noted that replacement of casts with modulo operations increases efficiency: like in the previous case, I tested my version of the cipher and a modified one that uses casts instead of modulo operations. The timing of 100 and 100 iterations are collected in the following table.

	100 iterations	1000 iterations
my implementation	0.156	1.601
modified implementation	0.188	1.813

Finally, I tested the team implementation and my version of the cipher over 901 triple (K,P,C) (strings corresponding respectively to keys, plaintexts, ciphertexts) checking it the ciphertexts are equal to the outputs of *IDEA_Encryption* and if the plaintexts are equal to the outputs of *IDEA_Decryption*. It is calculated the time of execution: my versions takes 13.250 seconds, the team version 17.656 seconds.

References

[1] Magma handbook. https://magma.maths.usyd.edu.au/magma/handbook/.

[2] The idea block cipher. cryptonessie.org, 2000.