Design and Implementation of New DES64X and DES128X on 32, 64 Bit Operating System Environments

Ammar H. Jasim
Department of computer Science /College of Science for women
University of Baghdad
E-mail: ammar_hussein_2004@yahoo.com

Abstract

In this paper, a description of a design for new DES block cipher, namely DES64X and DES128X. The goals of this design, a part of its security level, are large implementation flexibility on various operating systems as well as high performances. The high level structure is based on the principle of DES and Feistel schema, and proposes the design of an efficient key-schedule algorithm which will output pseudo-random sequences of sub keys.

The main goal is to reach the highest possible flexibility, in terms of round numbers, key size, and block size. A comparison of the proposed systems on 32-bit, 64-bit operating system, using 32-bit and 64-bit Java Virtual Machine (JVM), showed that the latter has much better performance than the former.

تصميم وتنفيذ طريقة جديدة لتشفير البيانات القياسية في بيئات نظام تشغيل ٣٢ و ٦٤ ثنائية

عمار حسين جاسم
قسم علوم الحاسبات/كلية العلوم للبنات/ جامعة بغداد
البريد الالكتروني: ammar_hussein_2004@yahoo.com

المستخلص

في هذه الورقة البحثية, تم وصف تصميم طريقة جديدة لخوارزمية تشفير البيانات القياسية, اطلق عليها طريقة التشفير القياسية ذات ٦٤ ثنائية و طريقة التشفير القياسية ذات ١٢٨ ثنائية. الاهداف الرئيسية لهذا التصميم, زيادة مستوى الامن و مرونة تطبيقٍ كبيرةٍ على أنظمةِ التشغيل المُختلفةِ بالإضافة إلى الأداء العالي. استند التصميم على مبدا خوارزمية تشفير البيانات القياسية و صيغة فيستل مع اعداد تصميم كفوءة لخوارزمية جدولة المفاتيح؛ والتي يجب ان تكون مخرجاتها سلسلة شبه عشوائية للمفاتيح الجزئية.
الهدف الرئيسي هو الوصول الى اعلى مرونة ممكنة,تبدأ من ناحية الرقم المدّوِر, وسعة المفتاح و سعة حيز الخزن. عرضنا مقارنةَ لتطبيقِ الأنظمةِ المُقتَرَحةِ على نظام تشغيل ٣٢ ثنائية و ٦٤ ثنائية باستعمال ماكنة جافا الإفتراضية وبينا افضلية اداء النظام الثاني الى الاول.

1. Introduction

DES is one of the most popular block ciphers. It is a block cipher encrypting 64 bits of data block with a 56-bit key size. The small key size and increased computing power of modern computers make DES unsafe with exhaustive search attack. Therefore, a cipher based on DES with a larger key size is necessary [1].

DES is used in many and varied crypto-based applications. It is used to protect the secrecy of login passwords, E-mail messages, video transmission (such as pay-per-view movies), stored data files, and internet distributed digital content, etc [2].

In this paper, the objective is to develop a block cipher where the key and block sizes are significantly large. The proposed block cipher relies upon the encryption techniques of confusion and diffusion.

Confusion is accomplished through substitution. Specially chosen sections of data are substituted for corresponding sections from the original data. The choice of the substituted data is based upon the key and the original plaintext. Diffusion is accomplished through permutation. The data is permuted by rearranging the order of the various sections. These permutations, like the substitutions, are based upon the key and the original plaintext [3].

The substitutions and permutations are specified in the proposed system, by the DES algorithm [4]. Chosen sections of the key and the data are manipulated mathematically and then used as an input to a look-up table. These tables are called the S-boxes and the P-boxes, for the substitution tables and the permutation tables. In software terms, these look-up tables are realized as arrays of bytes. Usually the S-box and P-box are combined so that the substitution and the following permutation for each round can be done with a single operation. In order to calculate the inputs to the S-box and P-box arrays, portions of the data are XORed with portions of the key. One of the 32, 64-bit halves of the 64,128-bit data and the 64,128-bit key are used. The S-box, P-box look-up tables, and calculations, upon key and data which generate the inputs will constitute a single round of feedback to the system.

The same process of S-box and P-box substitution and permutation is repeated sixteen times, forming the sixteen rounds of the block cipher algorithm. There are also initial and final permutations which occur before and after the sixteen rounds. These initial and final permutations exist for historical reasons dealing with implementation on hardware and do not improve the security of the algorithm. For this reason they are left out of system implementations at capture time. They are, however, included in this literature analysis as they are part of the technical definition of the proposed system.

2. Proposed Design Description

The design has two versions, 32-bit and 64-bit. The first one was designed for 32-bit operating system and using 64-bit block cipher with 64-bit key, the other for 64-bit operating system with same features as in the 32-bit, however, the key length equal to 128-bit.

2.1 DES64X

As illustrated in figure (1), the DES64X is a 16-time iteration of a round function denoted by ROUND64, which is built as a Feistel schema [5].

Formally, these functions take 64-bit $X_{(64)}$, 64-bit round key $rK_{(64)}$ as input, and 64-bit output $Y_{(64)}$. The encryption $C_{(64)}$ of a 64-bit plaintext $P_{(64)}$ is defined as:

$P_{(64)} = IP(m_{(64)})$ (Use IP from table 1 to permute bits)

$T_{(64)} = ROUND\ 64(......(ROUND\ 64(P_{(64)}, rK_{0(64)}),......, rK_{r-2(64)}), rK_{r-1(64)})$

$C_{(64)} = IP^{-1}(T_{(64)})$ (Transpose using inverse IP from table 2)

Where

$rK_{(64)} = rK_{0(64)} \| rK_{1(64)} \| ...... \| rK_{r-1(64)}$, are the sub keys produced by the key schedule algorithm from the key $K_{(64)}$.

The decryption $P_{(64)}$ of 64-bit cipher text $C_{(64)}$ is the same as the above encryption only reversing the process of key scheduling, defined as:

$P_{(64)} = ROUND\ 64(......(ROUND\ 64(C_{(64)}, rK_{r-1(64)}),......, rK_{1(64)}), rK_{0(64)})$

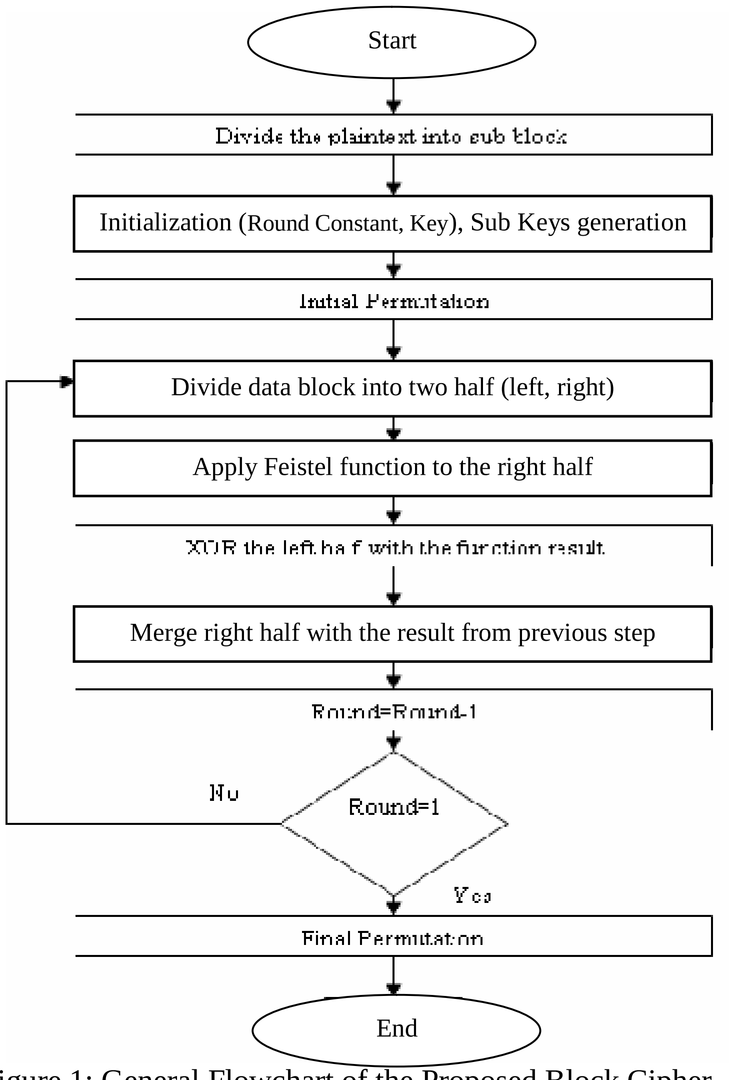

Start
Divide the plaintext into sub block
Initialization (Round Constant, Key), Sub Keys generation
Initial Permutation
Divide data block into two half (left, right)
Apply Feistel function to the right half
XOR the left half with the function result
Merge right half with the result from previous step
Round=Round-1
Round=1
No
Yes
Final Permutation
End

Figure 1: General Flowchart of the Proposed Block Cipher

Table (1): Initial Permutation ***IP*** 64-Bit

	1	2	3	4	5	6	7	8
1	58	50	42	34	26	18	10	2
2	60	52	44	36	28	20	12	4
3	62	54	46	38	30	22	14	6
4	64	56	48	40	32	24	16	8
5	57	49	41	33	25	17	9	1
6	59	51	43	35	27	19	11	3
7	61	53	45	37	29	21	13	5
8	63	55	47	39	31	23	15	7

Table (2): Inverse Initial Permutation ***IP*** 64-Bit

	1	2	3	4	5	6	7	8
1	40	8	48	16	56	24	64	32
2	39	7	47	15	55	23	63	31
3	38	6	46	14	54	22	62	30
4	37	5	45	13	53	21	61	29
5	36	4	44	12	52	20	60	28
6	35	3	43	11	51	19	59	27
7	34	2	42	10	50	18	58	26
8	33	1	41	9	49	17	57	25

2.1.1 The Internal Function

The function, used in DES64X built as Feistel schema, transforms a 64-bit input $X_{(64)}$ split into left and right 32-bit halves $X_{(64)} = X_{l(32)} \| X_{r(32)}$ and a 64-bit round key $rK_{(64)}$ in a 64-bit output $Y_{(64)} = Y_{l(32)} \| Y_{r(32)}$ is done as follows:

$$Y_{(64)} = Y_{l(32)} \| Y_{r(32)} = ROUND64(X_{l(32)} \| X_{r(32)}) = X_{r-1(32)} \| (X_{l-1(32)} \oplus f32(X_{r-1(32)}, rK_{64})$$

Figure (2) illustrates the Round Function.

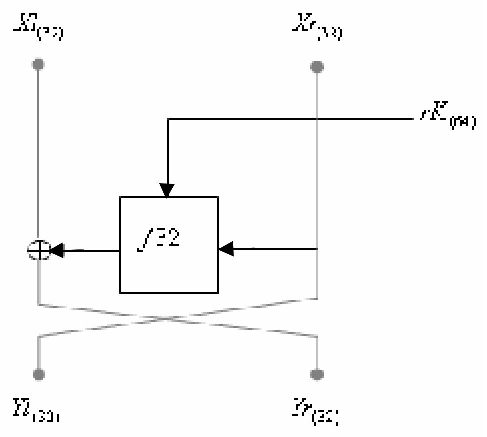


Figure 2: DES64X Round Function

Function $f32$ in DES64X consists of three layers, substitution layer noted as Sub4, diffusion layer noted as Diff4, and Exclusive-OR as XOR layer. The 32-bit $X_{(32)}$, 64-bit round key is used as input and 32-bit $Y_{(32)}$ as an output. Function $f32$ is defined as:

$$f32(X_{(32)}, rK_{(64)}) = Sub4(Diff4(Sub4(X_{(32)} \oplus rK_{0(32)})) \oplus rK_{1(32)}) \oplus rK_{0(32)}$$

The function Sub4 is a method by which as 32-bit as input $X_{(32)} = X_{0(8)} \| X_{1(8)} \| X_{2(8)} \| X_{3(8)}$ and 32-bit as output. Defined as:

$$Y_{(32)} = Sub4(X_{0(8)} \| X_{1(8)} \| X_{2(8)} \| X_{3(8)}) = Sbox(X_{0(8)}) \| Sbox(X_{1(8)}) \| Sbox(X_{2(8)}) \| Sbox(X_{3(8)})$$

This is illustrated in Figure (3).

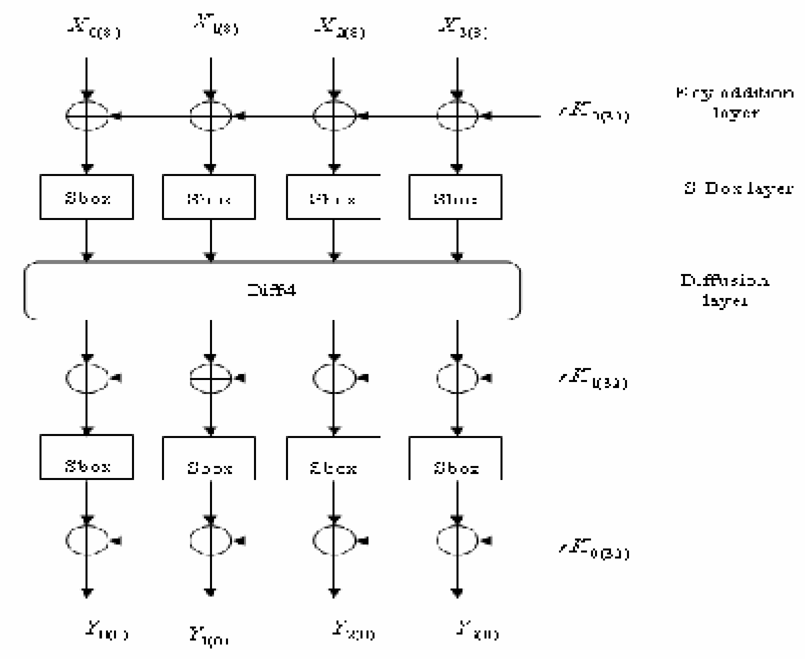


Figure 3: 32-bit Feistel Function

2.2 DES128X

The DES128X is 16 time's iteration of a round function denoted ROUND128 is built as a Feistel schema, with INVR128 used for decryption.

Formally, these functions take 128-bit data block $X_{(128)}$, 128-bit round key $rK_{(128)}$ as input and 128-bit output $Y_{(128)}$:

The encryption $C_{(128)}$ of 128-bit plaintext $P_{(128)}$ is defined as:

$P_{(128)} = IP(m_{(128)})$ (Use IP from table 3 to permute bits)

$T_{(128)} = ROUND\ 128\ (......(ROUND\ 128\ (P_{(128)}, rK_{0(128)}),......, rK_{r-2(128)}), rK_{r-1(128)})$

$C_{(128)} = IP^{-1}(T_{(128)})$ (Transpose using inverse IP from table 4)

Where

$rK_{(128)} = rK_{0(128)} \| rK_{1(128)} \| ...... \| rK_{r-1(128)}$

Note that, the sub keys produced by the key schedule algorithm from the key $K_{(128)}$.

The decryption $P_{(128)}$ of 128-bit cipher text $C_{(128)}$ is the same as the encryption but in reverse order of key scheduling, defined as:

$P_{(128)} = ROUND\ 128\ (......(ROUND\ 128\ (C_{(128)}, rK_{r-1(128)}),......, rK_{1(128)}), rK_{0(128)})$

Table (3): Initial permutation ***IP*** 128-bit

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	121	113	105	97	89	81	73	65	57	49	41	33	25	17	9	1
2	123	115	107	99	91	83	75	67	59	51	43	35	27	19	11	3
3	125	117	109	101	93	85	77	69	61	53	45	37	29	21	13	5
4	127	119	111	103	95	87	79	71	63	55	47	39	31	23	15	7
5	122	114	106	98	90	82	74	66	58	50	42	34	26	18	10	2
6	124	116	108	100	92	84	76	68	60	52	44	36	28	20	12	4
7	126	118	110	102	94	86	78	70	62	54	46	38	30	22	14	6
8	128	120	112	104	96	88	80	72	64	56	48	40	32	24	16	8

Table (4): Inverse Initial permutation ***IP*** 128-bit

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	72	8	80	16	88	24	96	32	104	40	112	48	120	56	128	64
2	71	7	79	15	87	23	95	31	103	39	111	47	119	55	127	63
3	70	6	78	14	86	22	94	30	102	38	110	46	118	54	126	62
4	69	5	77	13	85	21	93	29	101	37	109	45	117	53	125	61
5	68	4	76	12	84	20	92	28	100	36	108	44	116	52	124	60
6	67	3	75	11	83	19	91	27	99	35	107	43	115	51	123	59
7	66	2	74	10	82	18	90	26	98	34	106	42	114	50	122	58
8	65	1	73	9	81	17	89	25	97	33	105	41	113	49	121	57

2.2.1 The internal function

The function used in DES128X is built as Feistel schema, which transform a 128-bit input $X_{(128)}$ split into left and right 64-bit halves $X_{(128)} = X_{ll(32)} \| X_{lr(32)} \| X_{rl(32)} \| X_{rr(32)}$ and a 128-bit round key $rK_{(128)}$ in a 128-bit output $Y_{(128)} = Y_{ll(32)} \| Y_{lr(32)} \| Y_{rl(32)} \| Y_{rr(32)}$ as follow:

$$Y_{(128)} = Y_{l(32)} \| Y_{r(32)} = ROUND128(X_{ll(32)} \| X_{lr(32)} \| X_{rl(32)} \| X_{rr(32)}) = (X_{lr-1(32)} \| (X_{ll-1(32)} \oplus f64(X_{lr-1(32)}, rK_{64})) \|$$
$$(X_{rr-1(32)} \| (X_{rl-1(32)} \oplus f64(X_{rr-1(32)}, rK_{64}))$$

Figure (4) illustrates the Round Function.

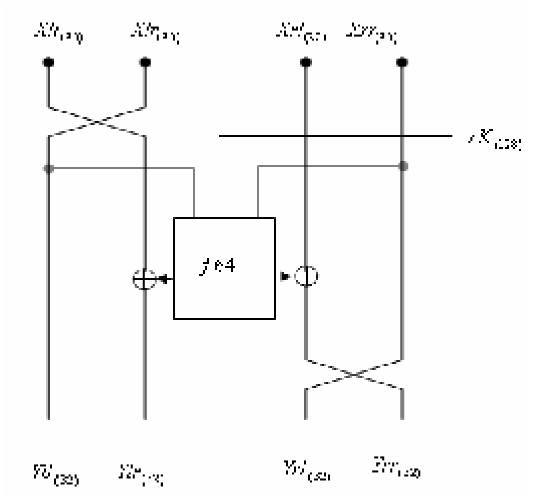


Figure 4: DES128X Round Function

Function $f64$ in DES128X consists of three parts, substitution part noted as Sub8, diffusion part noted as Diff8, round key part. This function takes 64-bit $X_{(64)}$, 128-bit round key as input and 64-bit $Y_{(64)}$ as an output.

The function f64 defined as:

$$f64(X_{(64)}, rK_{(128)}) = Sub8(Diff8(Sub8(X_{(64)} \oplus rK_{0(64)})) \oplus rK_{1(64)}) \oplus rK_{0(64)}$$

The function Sub8 is a method that takes 64-bit as input $X_{(64)} = X_{0(8)} \| X_{1(8)} \| X_{2(8)} \| X_{3(8)} \| X_{4(8)} \| X_{5(8)} \| X_{6(8)} \| X_{7(8)}$ and 64-bit as output. Defined as:

$$Y_{(64)} = Sub8(X_{0(8)} \| X_{1(8)} \| X_{2(8)} \| X_{3(8)} \| X_{4(8)} \| X_{5(8)} \| X_{6(8)} \| X_{7(8)})$$

$$= Sbox(X_{0(8)}) \| Sbox(X_{1(8)}) \| Sbox(X_{2(8)}) \| Sbox(X_{3(8)}) \| Sbox(X_{4(8)}) \| Sbox(X_{5(8)}) \| Sbox(X_{6(8)}) \| Sbox(X_{7(8)})$$

This is illustrated in Figure (5).

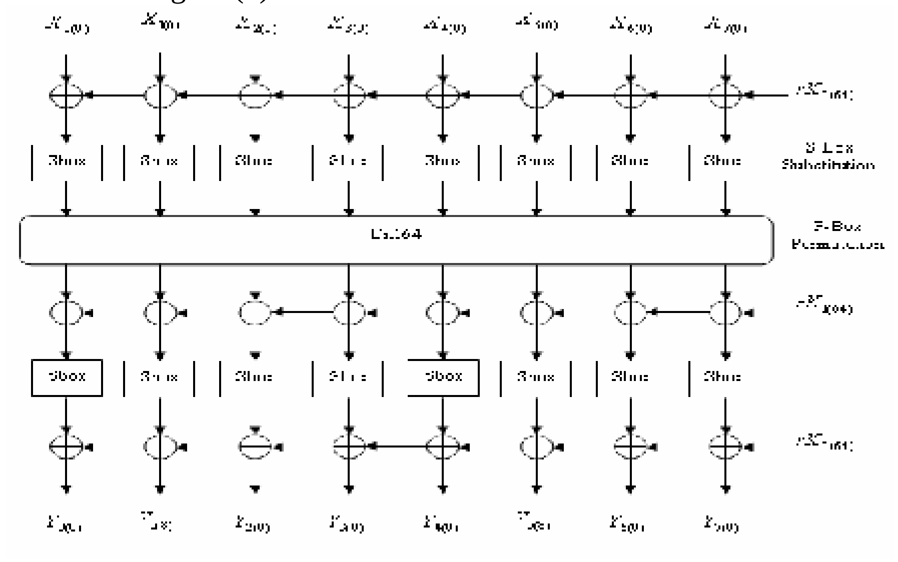
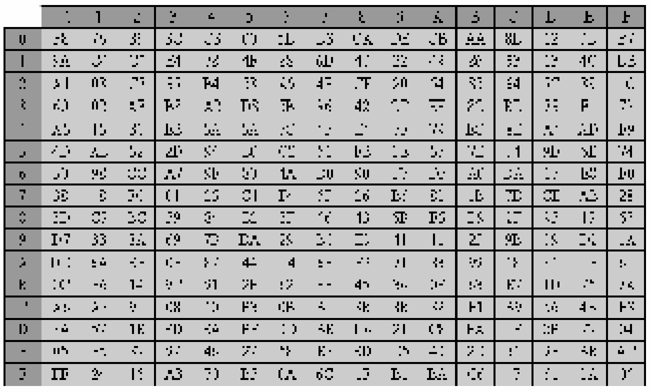


Figure 5: 64-Bit Feistel Function

And the S-Box function is a look up table defined in Table (5):

Table 5: DES64X, DES128X S-Box



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
4	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
5	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
6	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
7	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
8	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
9	[illegible]	33	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	11	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
A	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
B	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
C	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
D	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
E	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
F	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

3. Key-Schedule Algorithm

Key scheduling is used to derive the sub keys and it will be used in the following manner: first, the key block is divided into halves many times depending on the size of key block. Then, the halves are circularly shifted left by either one or two bits, depending on the round. After being shifted, permutation operation is implemented on these halves, and finally key production for each round is produced. Because of the shifting process, a different subset of key bits is used in each sub key. The key scheduling algorithm is illustrated in Figures 6 and 7.

For DES64X

$rK_{(64)} = rK_{0(64)} \| rK_{1(64)} \| .... \| rK_{15(64)}$

For DES128X

$rK_{(128)} = rK_{0(128)} \| rK_{1(128)} \| .... \| rK_{15(128)}$

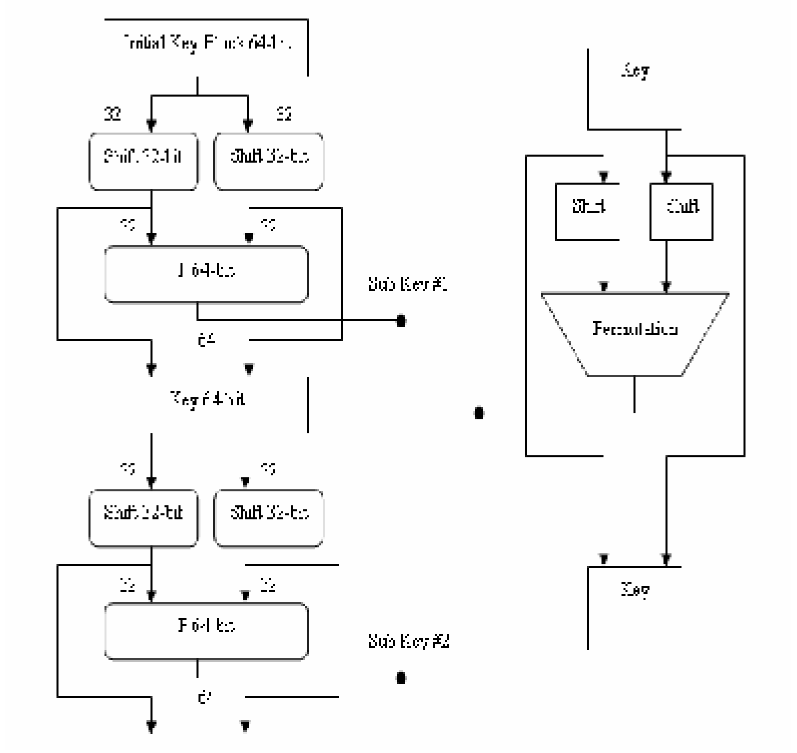


Figure 6: 64-bit key Schedule (KSched64X)

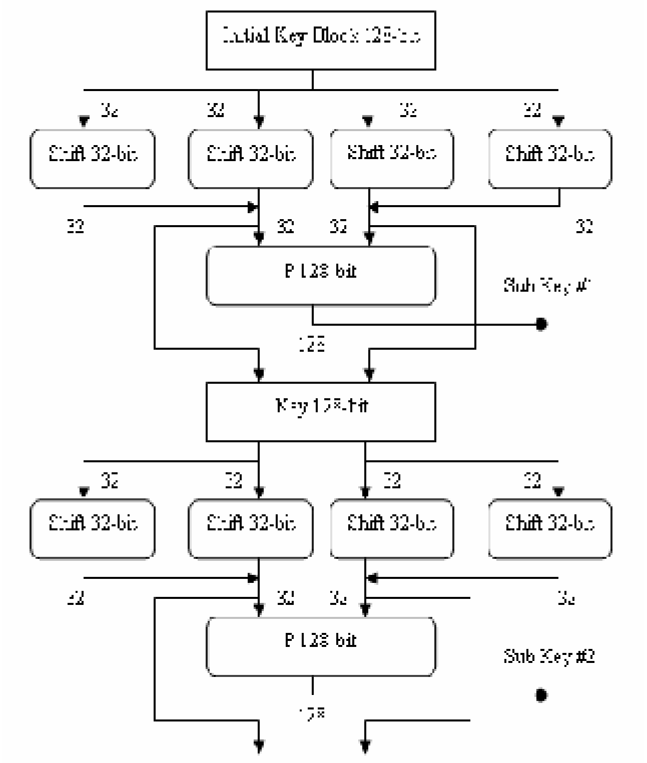


Figure 7: 128-bit key Schedule (KSched128X)

4. Core System Implementation

Java language is popular because of its platform independence, making it useful in varieties of technologies ranging from embedded devices to high-performance systems. The platform-independent property of Java, which is visible at the Java bytecode level, is only made possible owing to the availability of a Virtual Machine (VM), which needs to be designed specifically for each underlying hardware platform. More specifically, the same Java bytecode should run properly on a 32-bit or a 64-bit VM. In this paper, compare the behavioral characteristics of 32-bit and 64-bit VMs using the proposed design. This is done using the DES64X and DES128X using JAVA builder 7.0.
The proposed design implementations of DES64X and DES128X are written for two types of operating systems. By taking the advantage of the 64-bit operating system, system implementation is done as follows; for the DES128X, store the two block halves of each round operation in two separate 64-bit arrays. However, instead of storing them in a 32-bit format, these are stored in a 64-bit format which resulted from applying the permutation to a 64-bit array.

Each round then proceeds as follows; the right half, which is already in a 64-bit array, is XORed with the first half of the subkey, which is also contained in a 64-bit array. The resulting 64-bit array is divided into eight groups of eight bits, each of which is used as an index to the S-box. Then the 128-bit permutation is applied to the 64-bit array and the result is XORed with the second half of subkey. This mechanism treats the S-box as a straightforward look up table. The look up table produces a 128-byte array, rather than a 32-byte array. This result is then XORed with the left half, which is also stored in a 64-bit format.

The primary benefits of a 64-bit OS are in the increased computing capacity of having twice the bandwidth of data flow and the ability to use more system memory (RAM) than the 32-bit operating system [6].

5. Proposed Implementation

As mentioned before, the complete design was implemented with the use of 32, 64-bit OS; figure (1) showed the proposed system. Each of the DES64X and DES128X should support encryption and decryption. Decryption, in each case, uses the same algorithm as encryption. The only difference is that the sub keys have to be generated in a reverse order, as compared with encryption.

Each DESXX begins with Initial Permutation IP and ends with the inverse of the initial permutation IP^{-1}. This system has the possibility to process 64,128-bit independent data blocks, which increases the operation throughput. The DESXX key scheduling can be performed on the fly. The sub-keys generated by using the proposed key schedule algorithm.

The key generator consists of 16 rounds. The 64,128-bits input key is initially divide into two parts and goes through the appropriate shift operation and finally passed through a second round permutation for each sub-key, as illustrated in figures 6 and 7.

At the start, the 64-bits data block, encryption key is applied on the key scheduler to pre compute the sixteen 64,128-bit sub keys and store it as an array of bytes in order to force the appropriate key at the appropriate time. Finally, the encryption key is forced and DESXX operates in the encryption mode.

6. Functional Description

After an initial permutation, the input data is split into two half words, left and right. This is followed by 16 rounds of identical operations. The right word is processed with Feistel function that includes XOR operation, S boxes substitution and diffusion operation as depicted in figures 5 and 6. The output of the S boxes is permuted and then XORed with the left word. The result is used to update the right word array at the end of each round. Also, the previous right word is stored in the left word array. The processed key changes at each round as well, owing to shift and permutation operations in key schedule algorithm. At the end of the 16 rounds the left and right words are reassembled together and passed through the inverse of the initial permutation. The DESXX core is partitioned into two modules as showed in figures 2 and 4.

6.1 Key Process and Initial Permutation methods

In key process, a class is responsible for dividing the input key that is used at every round. However, initial permutation it simply performs an initial permutation of input data bits.

6.2 S-Box Tables

This is a group of 8 input and 8 output look up tables that maps the incoming 8-bit word into an 8-bit one for DES128X and four-8 input and 8 output that will map the incoming 8-bit. S-box is usually implemented as an array of constants that is indexed by the 8-bit input.

6.3 Permutation and Final Permutation Methods.

Permutation method performs a permutation on input data bits while final Permutation method performs a final permutation of the bits of the output data.

6.4 Mode

This unit controls the mode of the proposed system (if mode=1 the system is an encryption mode else the system in decryption mode).

7. Overall System Scenario

The proposed system is a block cipher designed to use simple whole-byte operations. The system is secure and versatile because it uses large blocks of data and a key. Both key and block size can be chosen to be 64,128-bits. The cipher uses a fix number of rounds equal to 16.

Four different stages are used during encryption and decryption, as can be seen in figure 1, including permutation process, applying Feistel function, XOR operation, and mix operation. The XOR, Substitution, and permutation stages are explained as follow: The substitute bytes transformation (S-box) is a simple lookup table. Proposed system defines a 16 x 16 array of byte values, the S-box, which contains a permutation of all possible 256 8-bit values. Each byte is mapped to a new byte in the following manner: The leftmost 4 bits are used as a column value. Row and column values use as indexes into the S-box to select 8-bit output.

XOR stage performs a bitwise XOR on 64, 128-bits of the state with the 64, 128-bits of the round key. The flow of one round of the proposed block cipher is seen in figures 2 and 4.

The proposed block cipher is implemented using JAVA Builder 7.0 with a 64,128-bit block size, a 64,128-bit key size, and Feistel schema. Using two key sizes is sufficient to describe the performance of the block cipher algorithm in Java. The program is built such that all arguments are passed from the command line, enabling the program to be called from scripts. Five files are expected: the text file, a file where the encrypted data is written, a decrypted file where the decrypted contents of the encrypted file are written. Two files containing the 64,128-bit key in hex format, and a results file. The main method initializes the variables and ensures that the correct number of arguments is passed from the command line, initializing the constructor and then calls the test method. The test method performs timing functions, results compilation, and calls encrypt and decrypt methods, which takes the key as a parameter. The encrypt and decrypt functions initialize the cipher with the key and read in the text file or ciphertext then perform the encryption or decryption and writing out the results to output files.

8. Testing Data File

The program is designed to encrypt and decrypt five files of different sizes of 1·0KB, 2MB, 20MB, 200MB and 300MB. The plaintext file was encrypted and written to a ciphertext file then the ciphertext file was decrypted and written to a different plaintext file with the speed of encryption and decryption being timed (capturing several data components during the process, including the encrypting, decrypting and current system times). The resulting times of each file being encrypted or decrypted written to a file. The results were then compiled and analyzed.

9. Experimental Results

Test results were being done using Windows XP SP3 professional 32-bit and 64-bit operating system with an Intel® Core 2Due Processor T7250 @2.00GHz-2MB L2 Cache and 1024 MB RAM, the compilers and libraries used are:

- Java Builder 7.0, SDK Standard Edition Version 6.5
- Java Cryptography Extension (JCE) using Crypto++ library and JCE API [7].

Time of encryption and decryption is calculated by capturing the current system time using Java system calls immediately before calling either the encrypt or decrypt methods and capturing the current system time immediately after the method returns. Finally, the end time is subtracted from the start time and the results are written to file.

Look at the overall performance (encryption) for different file sizes, for the system. Execution time for each algorithm is calculated as execution time without file I/O (only the cipher block without I/O memory). Timing is calculated as an average of 5 runs for each algorithm for more accurate result. The five different file sizes are considered to observe the performance of the algorithm, and resultant times are recorded.

As mentioned above, five files of different sizes were encrypted and decrypted by each combination: 100KB, 2MB, 20MB, 200MB and 300MB. The plaintext file was encrypted and written to a ciphertext file then the ciphertext file was decrypted and written to a different plaintext file with the speed of encryption and decryption being timed as in tables 6,7,8,9. The encryption and decryption process was repeated 5 times to assure that the results are consistent and are valid to compare the performance on the operating systems. The resulting times of each run being written to file. The results were then compiled and analyzed.

Comparison of execution times for the DES64X and DES128X encryption using different file sizes are shown in tables 6 and 7, and comparison of execution times for decryption the same files are shown in tables 8 and 9. A comparison is conducted between the results of encryption and decryption schemes in terms of encryption, decryption time and throughputs. A study is performed on the effect of changing file size on throughput, and CPU time for each mode of proposed system.

Table 6: Time consumption of DES64X for encrypt different file sizes without File I/O (in millisecond)

File size	32-bit OS	64-bit OS
100KB	24.39	12.5
2MB	487.8	250
20MB	4852	2350
200MB	48780.5	24250
300MB	73170.73	37500

Table 7: Time consumption of DES128X for encrypt different file sizes without File I/O (in millisecond)

File size	32-bit OS	64-bit OS
100KB	37.037	18.867
2MB	740.7	377.35
20MB	7500.4	3675.5
200MB	73075	37850
300MB	110250.1	56603.7

Table 8: Time consumption of DES64X for decrypt different file sizes without File I/O (in millisecond)

File size	32-bit OS	64-bit OS
100KB	16.94	10.2
2MB	338.9	204.08
20MB	3375.8	2075
200MB	33890.2	20408
300MB	50847.5	30612

Table 9: Time consumption of DES128X for decrypt different file sizes without File I/O (in millisecond)

File size	32-bit OS	64-bit OS
100KB	23.25	15.38
2MB	465.1	307.65
20MB	4575	3075.9
200MB	46511.6	30796.2
300MB	69767.4	46153.9

This implementation achieved a throughput as shown in tables 10 and 11.

Table 10: Throughput of DES64X to encrypt and decrypt different file size (Megabytes/Second)

	32-bit OS	64-bit OS
Encryption	4.1	8
Decryption	5.9	9.8

Table 11: Throughput of DES128X to encrypt and decrypt different file size (Megabytes/Second)

	32-bit OS	64-bit OS
Encryption	2.7	5.3
Decryption	4.3	6.5

10. Conclusions

The work presented here is primarily concerned with the design and implementation of new DES64X and DES128X on 32, 64 Bit Operating System Environments. The DES128X system is more secure but it slows down the encryption when implemented on 32-bit platform, because it has to do more work for the same amount of input data in a single execution cycle.

This paper focuses on system implementation based on a 64-bit platform. The proposed DES64X on 64-bit OS implementation is faster compared with that on 32-bit. From these results, it is easy to observe that 64-bit operating system has an advantage over other 32-bit operating systems in terms of throughput for large data. Also DES64X on 64-bit has almost approximately twice the throughput of DES64X on 32-bit, in other words it needs half of the time as compared with DES64X when implemented on 32-bit, to process the same amount of data.

References

[1] Stallings W., **"Cryptography and Network Security"**, Principles and Practice, 3rd Edition, Prentice Hill, 2002.

[2] Ashish Patel and Ajay Kumar Garg, **"Study and Implementation of Cryptographic Algorithms"**, 2008.

[3] K. Anup Kumar and S. Udaya Kumar, **"Block cipher using key based random permutations and key based random substitutions"**, March 2008.

[4] Lars Ramkilde Knudsen, **"Block Ciphers Analysis, Design and Applications"**, PhD thesis, Aarhus University, Denmark, July 1, 1994.

[5] T. Shirai and K. Shibutani, **"On Feistel Structures Using a Diffusion Switching Mechanism"**, Springer- Verlag, 2006.

[6] Microsoft Help and Support, **"Overview of the compatibility considerations for 32-bit programs on 64-bit versions of Windows"**, http://support.microsoft.com/kb/896456#XSLTH3120121124120121120120

[7] Scott oaks, **"JAVA Security"** 2nd Edition, 2002, O'Reilly & Associates, Inc.