New Block Ciphers

Akelarre: a new Block Cipher Algorithm

Gonzalo Álvarez, Dolores de la Guía, Fausto Montoya y Alberto Peinado

Departamento de Tratamiento de la Información y Codificación.
Instituto de Física Aplicada, Consejo Superior de Investigaciones Científicas.
Serrano 144, 28006 Madrid.
Tel: +34 (1) 5631284, Fax: +34 (1) 5631371, E-mail: fausto@iec.csic.es

Abstract: *This document presents the operation of the high speed encryption algorithm Akelarre. Akelarre is a secret-key iterated block cipher of great flexibility in its security level, allowing the modification via software of such parameters as the number of rounds and the key length. Presumably, it is cryptographically secure, due to the heavy use of data dependent rotations and the mixing of arithmetic operations from different algebraic groups. The encryption and decryption algorithms are identical and of easy implementation in both hardware and software.*

1. Introduction

Akelarre is a secret key fast block cipher, suitable for hardware or software implementations, the algorithm main operation is the data dependent cyclic rotation of prime-length registers, combined with bit-wise XOR and two's complement addition. The architecture is specially designed to ensure with a 100% probability that *all* the plaintext will always determine *at least one* rotation. The use of prime-length registers avoid invariant rotations (output identical to input), which could appear if composite-length registers were used.

These operations belongs to different algebraic groups, so that they are incompatible in the sense that no pair of the three operations satisfies a distributive law or an associative law. The operations are so arranged that the output of an operation of one type is never used as the input to an operation of the same type. In this way, any relationship between the plaintext, the ciphertext and the key is hidden and the relationship statistics is very complicated.

Akelarre is an iterated r round block cipher. The output of each iteration, or encryption round, is a complicated function of the output of the previous rounds and several sub-blocks derived from the user's secret key. The user may supply a key of variable length in 64 bits

increments, while a key expansion algorithm generates the various 32 bit sub-keys needed be used in the different encryption rounds.

Akelarre's great flexibility relies upon the possibility of freely choosing such parameters as the key length, l, and the number of rounds, r, so that the user can modify the trade-off between computation speed and demanded security.

The computational operations are simple, of easy implementation in high level language and available in the assembler instruction set of all microprocessors and DSP's. The hardware design is still easier and more efficient, with low memory requirements. The cyclic rotations can be readily programmed in high level language like a combination of two displacements.

The algorithm is word-oriented, the basic computational operations work on full computer words of data at a time, saving computer time. In this paper we describe the 32 bit word length version, but other versions for different word lengths have been designed.

As a result, it provides high security when appropriate parameters have been chosen.

2. Notation and Akelarre primitive operations

Akelarre uses the following primitive operations over pairs of 32-bit blocks:

- Two's complement addition (and its inverse), denoted by "+". It corresponds to the additive group in $\mathbb{Z}_{2^{32}}$.
- Bit-wise exclusive-OR, denoted by "⊕". It corresponds to the additive group in $\mathbb{Z}_2$.
- Cyclic left rotation: the cyclic rotation of word x left by y bits is denoted $x \lll y$.

A significant feature in Akelarre is the use of rotations, dependent on both the input data and the key. That is, intermediate result words are rotated an amount determined by other intermediate result words and the key, thus strengthening the cryptographic security of Akelarre, because the bits are rotated to random positions, which are not -and cannot be- predetermined.

Therefore, the strength of Akelarre relies upon the cryptographic properties of data dependent rotations and the mixing of operations from different algebraic groups having the same number of elements.

Rotations as a powerful cryptographic tool were successfully used in the nonlinear generation of pseudo-random sequences [FUS93], structure patented in 1993. Since 1991 a research work has been developed in the C.S.I.C. using rotations as nonlinear structures, as described in the following works: [FUS91], [GUI91], [GUI94].

3. Description of the algorithm

This section describes the design of Akelarre, which consists of two components:

- The encryption/decryption algorithm.
- The key expansion algorithm, intended to generate a long enough key from the secret key entered by the user.

3.1 Encryption algorithm

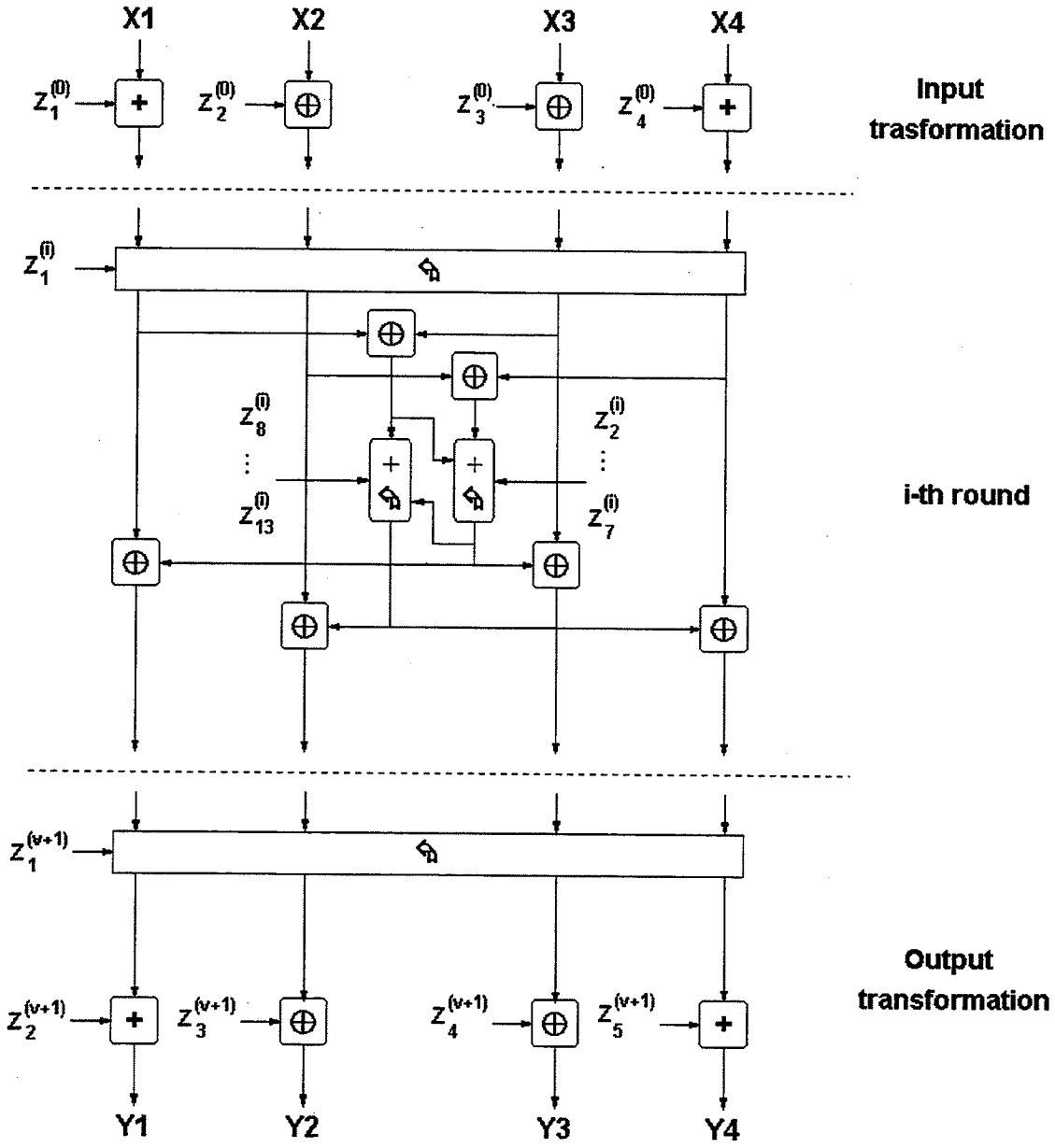
Figure 3.1 is an overview of the Akelarre algorithm. It is composed of three parts: the input transformation, *r* encryption rounds, and the output transformation.

The plaintext is reorganised prior to its entry into the algorithm, being partitioned into 128-bit blocks, denoted by X. These blocks, once fed into the algorithm, are divided into four 32-bit sub-blocks: X1, X2, X3 and X4. This four sub-blocks constitute the initial data to the input transformation.

The input transformation consists of four 32 bit-block operations. The two sub-blocks X1 and X4 are two's complement added with the sub-keys $Z_1^{(0)}$ and $Z_4^{(0)}$, respectively. The other two sub-blocks, X2 and X3, are bit-wise XORed with the sub-keys $Z_2^{(0)}$ and $Z_3^{(0)}$. The goal of this transformation is to prevent the adverse effect of possible input blocks of all-zeroes or all-ones, and to obstruct the differential cryptanalysis.

The first operation of the *i*-th encryption round is a cyclic rotation of the 128 bit block formed by the concatenation of the results of the input transformation or previous encryption round. The rotation is controlled by the seven least significant bits of $Z_1^{(i)}$. Then, the 128 bit resulting block is divided into four 32-bit sub-blocks. The four sub-blocks are combined with

twelve 32-bit sub-keys, represented as $Z_2^{(i)},\ldots, Z_{13}^{(i)}$ according to the addition-rotation structure shown in the figure 3.2.



X_i: 32-bit plain text sub-block
Y_i: 32-bit cipher text sub-block
$Z_i^{(j)}$: 32-bit key sub-block
+: Addition modulo 2^{32} of 32-bit integers
⊕: Bit-wise exclusive-OR of 32-bit sub-blocks
↰: Rotation

Figure 3.1 Akelarre algorithm design.

The output transformation consists of the cyclic rotation of the 128 bit block formed by the concatenation of the results of the *r*-th round, controlled by the seven least significant bits of $Z_1^{(r+1)}$, followed by four 32 bit-block operations in the same way as in the input trasformation, using the sub-keys $Z_2^{(r+1)}$, $Z_3^{(r+1)}$, $Z_4^{(r+1)}$, $Z_5^{(r+1)}$. After the output transformation, the four sub-blocks Y1, Y2, Y3, and Y4 are reattached to produce the output ciphertext block Y.

The goal of the cyclic rotations of the 128 bit blocks at the beginning of each round and at the beginning of the output transformation, is to enhance the diffusion when more than one round is used, although for one round operation has a low effect. Thus avoiding that the input data to the output transformation were simply the output of the input transformation XORed with the outputs Q1 and Q2 of each addition-rotation structure.

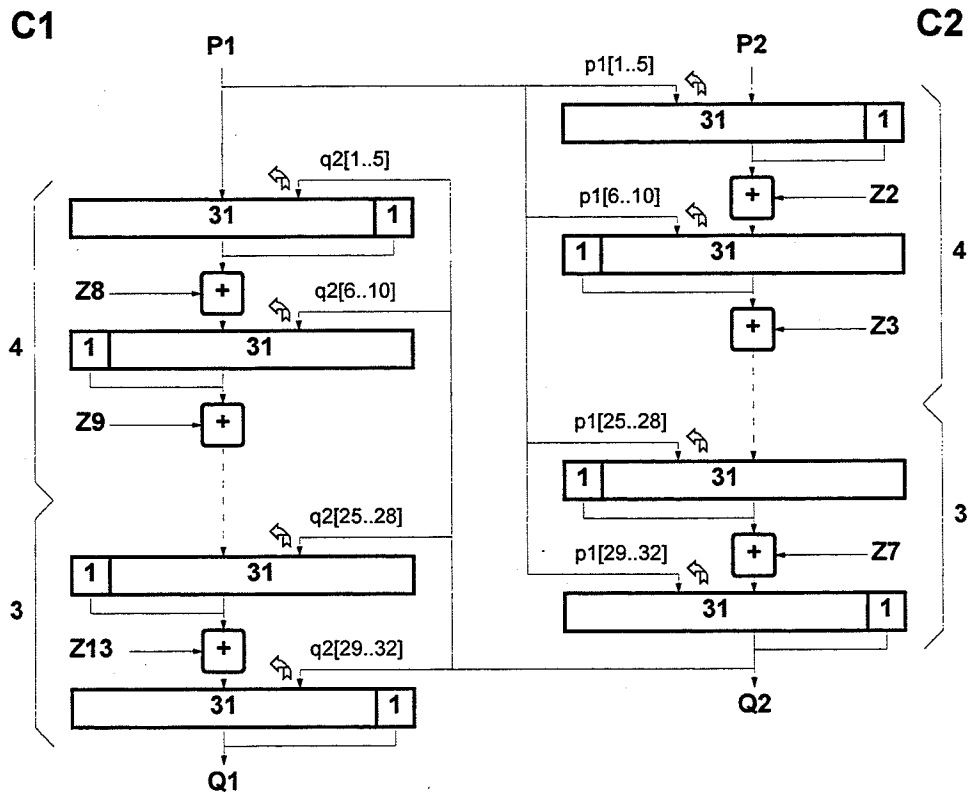


Figure 3.2 Addition-rotation structure

Figure 3.2 shows the addition-rotation structure. It is formed by two columns, each of them with six adders and seven rotators, where the amount rotated is determined by the other

column. This specially designed feature ensures with a 100% probability that *all* the plaintext and key bits will always determine *at least one* rotation. The reason why in each rotation 31 bits instead of the whole word are rotated is that a prime number has no divisors, whereas when a composite number is used, and the data in the rotator have a repetitive pattern, with repetition period equal to divisors of 32, an invariant rotation is produced, whose output result is identical to the input. However, with a prime number, any pattern will lead to a different output for any rotation. The convenience of this operation was studied in [GUI96].

In figure 3.2 it is observed how the 31-bit rotations are performed. In each 32 bit sub-block the least or the most significant bit is left unchanged, alternatively. The remaining 31 bit are rotated to the left an amount determined by the bits of the other column. The first 4 registers of each column are controlled by a set of 5 bits, whereas the 3 last registers are controlled by a set of 4 bits. The column C2 is controlled by the input bits of the column C1, but the column C1 is controlled by the output bits of the column C2. Akelarre is in fact a double iterated cipher, due to it has a variable number of rounds, r, but each round can be seen as the iteration of 14 half-rounds of rotation interleaved with 12 additions.

3.2 Decryption algorithm

The decryption algorithm is essentially the same as that for encryption and the computational graph of figure 3.1 is still of application. The only change is that the decryption key sub-blocks are computed from the encryption key sub-blocks as follows:

Round	*Encryption sub-keys*	*Decryption sub-keys*
Input transformation	$Z_1^{(0)}\ Z_2^{(0)}\ Z_3^{(0)}\ Z_4^{(0)}$	$-Z_2^{(r+1)}\ Z_3^{(r+1)}\ Z_4^{(r+1)}\ -Z_5^{(r+1)}$
1	$Z_1^{(1)}\ Z_2^{(1)} \ldots Z_{13}^{(1)}$	$(Z_1^{(r+1)})^{-1}\ Z_2^{(r)}\ Z_3^{(r)} \ldots Z_{13}^{(r)}$
2	$Z_1^{(2)}\ Z_2^{(2)} \ldots Z_{13}^{(2)}$	$(Z_1^{(r)})^{-1}\ Z_2^{(r-1)}\ Z_3^{(r-1)} \ldots Z_{13}^{(r-1)}$
i	$Z_1^{(i)}\ Z_2^{(i)} \ldots Z_{13}^{(i)}$	$(Z_1^{(r+2-i)})^{-1}\ Z_2^{(r+1-i)}\ Z_3^{(r+1-i)} \ldots Z_{13}^{(r+1-i)}$
r	$Z_1^{(r)}\ Z_2^{(r)} \ldots Z_{13}^{(r)}$	$(Z_1^{(2)})^{-1}\ Z_2^{(1)}\ Z_3^{(1)} \ldots Z_{13}^{(1)}$
Output transformation	$Z_1^{(r+1)}\ Z_2^{(r+1)}\ Z_3^{(r+1)}\ Z_4^{(r+1)}\ Z_5^{(r+1)}$	$(Z_1^{(1)})^{-1}\ -Z_1^{(0)}\ Z_2^{(0)}\ Z_3^{(0)}\ -Z_4^{(0)}$

Table 3.1 Encryption and decryption sub-keys.

3.3 Key expansion algorithm

The user secret key length can be freely chosen, in 64 bits increments. But the algorithm needs a much longer key, being the minimum 22 sub-keys of 32 bits (704 bits), needed for a one round algorithm. Therefore, a key expansion routine is required to transform the user secret key *k* into sub-keys to be used in the different encryption rounds.

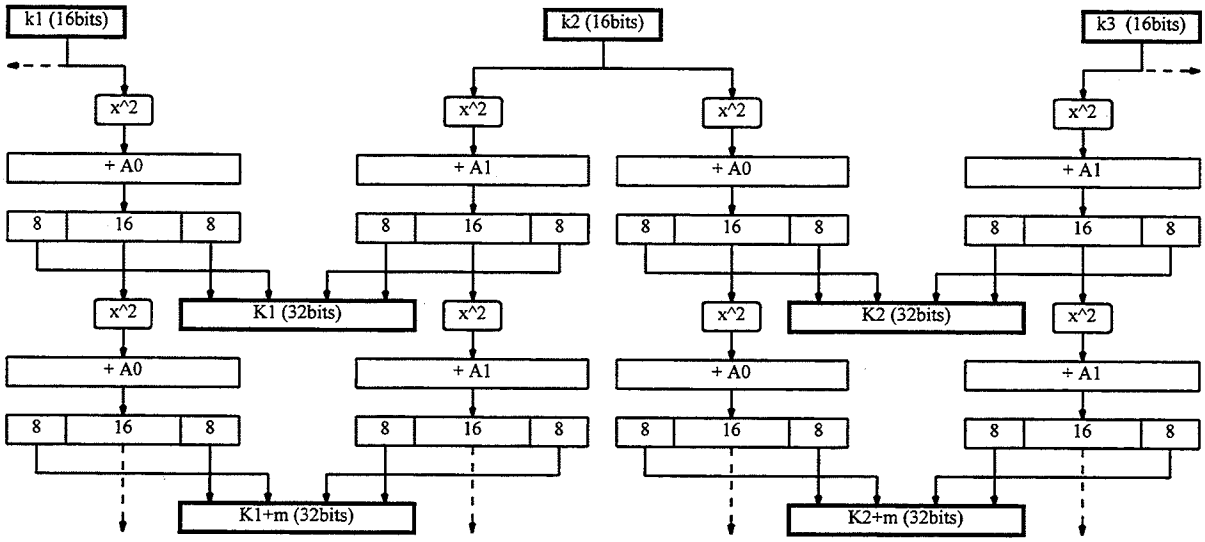


Figure 3.3 Key expansion algorithm.

The design of the expansion algorithm is depicted in figure 3.3. The user-selected key is partitioned into *m*, k_i. Each sub-block is squared and then added a constant A_0 or A_1 $\bmod 2^{32}$. The main reason that justifies the use of these constants is to avoid the adverse effect of a possible user secret key of all-zeroes. They must be chosen as A_0, $A_1 \neq 2^{32} - n^2$, with *n* integer, and with a good statistical distribution of 1's and 0's, to prevent a possible null result of the sum, either because the sub-key is zero or because the result of the squared is 2^{32}, which would be internally represented as 32 zeroes. In our implementation the values are: A_0=A49ED284H and A_1=735203DEH, they have been chosen carefully, to avoid weak keys. In the worst case, with the values k_i=98F3H and k_{i+1}=k_{i+2}=BDC7H and after the first key expansion routine iteration, K_i will have the form XX00XX00H, and the adjacent sub-key K_{i+1} XXXXXX00H, but in the next iteration of the key expansion routine K_{i+8} and K_{i+9} will be of the form XXXXXXXXH.

The 16 inner bits of the resultant addition are used to generate a new 32-bit word and continue iterating, while the 8 most significant bits and the 8 least significant bits of two adjacent columns are used to build up the sub-key K_i, in the way shown in figure 3.3.

The complicated non-linear function used to derive the sub-keys causes a avalanche effect of bit changes: a change of an input bit will induce a change of 8 output bits, in average.

Note that each 16-bit sub-block is used to generate at least two adjacent sub-keys, and usually much more sub-keys. Hence, an important avalanche effect is produced by the key expansion routine. A typical configuration of 4 rounds of encryption with a 128-bit user key, will need 61 sub-keys, to be generated from a set of 8 16-bit sub-blocks, it means that approximately each sub-block is used to generate 16 sub-keys, so a one bit change of the user key will produce, in average, a 128 bits change of the encryption sub-keys.

This key expansion function has been designed to be absolutely unidirectional, that is, the calculation of a certain K_i is of no avail to compute the previous K_{i-1}. Even the knowledge of K_i turns out to be useless to determine K_{i+1}, i.e., it is not only impossible to go backwards but to move forward as well in order to gain insight of the expanded key from a given K_i. This feature is of great importance as long as it prevents a possible cryptanalyst from obtaining the key expansion starting from a known sub-key K_i. The key expansion is pre-computed only once, thus slightly affecting the execution time of the algorithm.

After calculating all the K_i sub-keys, they are read sequentially to fill the $Z_j^{(i)}$ keys of the table 3.1

4. Discussion on performance and security of Akelarre

Akelarre offers the possibility of freely choosing such parameters as the key length, *l*, and the number of rounds, *r*, so that the user can modify the compromise between speed and security. For most applications, we propose that 4 rounds and 128-bit user secret key are used.

Several tests have been performed on a 130 MHz Intel® Pentium™, using a sample program working under Windows®95 compiled with Microsoft® Visual C++™ 4.0 compiler. It was obtained that Akelarre's speed with the proposed conditions, was 3,22 Mbits/second.

Obviously, this speed will increase or decrease as less or more encryption rounds are used.

We have performed some tests, in the same conditions, with other block cipher algorithms, and we have found that the speeds of IDEA and Akelarre are similar for the same number of rounds, although it must be remembered that the authors of IDEA recommend to use it with eight rounds. Otherwise, RC5 with the author's recommended configuration of 12 rounds, and 16-byte secret key is about two times the speed of Akelarre with the above parameters.

The theoretical security evaluation is not yet completed. However, extensive statistical tests have been performed and they show that there is no resemblance between the plaintext and the ciphertext apart from statistical coincidences. Furthermore, that the distribution of 1's and 0's approximates the ideal case of a perfectly random file.

As example, Figure 4.1. shows the graphs of the cross correlation test between plaintext and ciphertext and the runs test, for 512-byte files of all-zeroes plaintext, user key of all-zeroes, one round and key length of 128 bits.

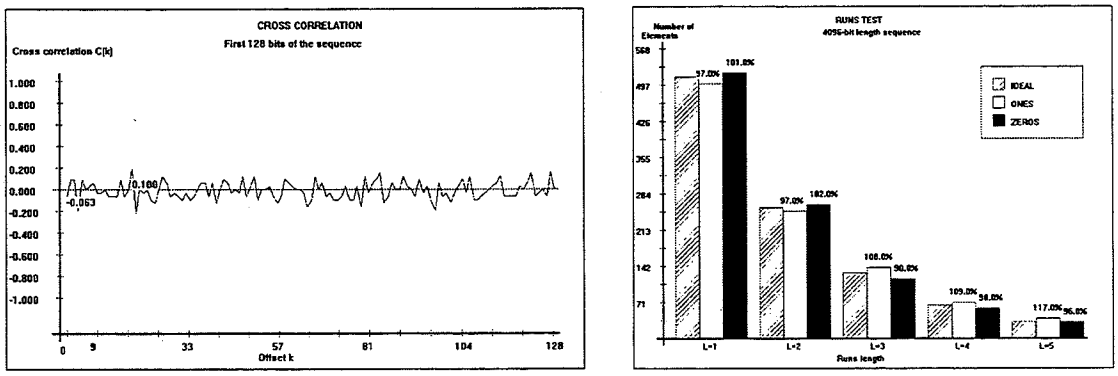


Figure 4.1 Cross correlation test between plaintext and ciphertext files and Runs test of ciphertext file.

In the table 4.1 we can see an example of how diffusion is achieved when a bit change takes place in the key or in the plaintext, with a user key of 64 bits and one round only. It should be highlighted that the change of one input data bit or one key bit is enough to produce a difference of about 50% of encrypted data.

Plaintext	Key	Ciphertext	Difference
0000 0000 0000 0000 0000 0000 0000 0000	0000 0000 0000 0000	6F06 02DF 74B0 117F 8372 49E9 72C2 F0CE	
0000 0000 0000 0000 0000 0000 **1**000 0000	0000 0000 0000 0000	759A D74E 7166 4EA2 9D9F 145A 7698 B79B	61 bits 47.65%
0000 0000 0000 0000 0000 0000 0000 0000	0000 0000 000**1** 0000	B915 06A7 C751 6324 D905 4DB1 2061 A315	66 bits 51.5%

Table 4.1 Comparison between plaintext and ciphertext when one plaintext bit and one key bit is changed with one round only.

Because of the double iterated architecture of Akelarre, differential and linear cryptanalysis seem quite difficult. To simplify both attacks let us ignore the initial transformation and the 128-bit rotations. And then let us operate on the smallest size cipher, that is one round. Then we face up to the addition-rotation structure, which is itself an iteration of a simpler structure, composed of alternated rotations and additions.

This structure presents a notable similarity with RC5 encryption algorithm, and can be analysed in the same way. Kaliski and Yin [Kal 95] have found a way to attack to RC5 with differential analysis focusing on characteristics for which the pair of inputs have the same rotations amount, because "... if a pair of inputs to a half-round have different rotation amounts, then the pair of outputs from the half-round will differ in many bits". They recognise that in the opposite case the differential cryptanalysis will be unfeasible. In our design, the rotation structure guarantees with 100% probability that a change of one bit in the plaintext will produce many rotations. If an input bit to C1 is involved the least rotation number is 1 and the highest is 6, whereas if an input bit to C2 is involved the number of rotations is 8, being the average 5-6 rotations. So we may suppose that Akelarre has a good security against the differential cryptanalysis.

In the same paper, the security of RC5 against linear cryptanalysis is assessed when more than 5 rounds of addition-rotation structure are used. We are using about 6.5 of similar rounds (14 half-rotations and 12 half-additions). Then, we can presume that Akelarre is not unsafe against linear cryptanalysis.

Taking into account the previous arguments, the statistical tests, and the fact that Akelarre will always be used with more than one round, we conclude that, for most applications, 4 rounds and 128-bit user secret key will be suitable. Nevertheless, we are working in a detailed security analysis that we hope to be able to present in next cryptographic meetings.

5. Comparison with other block ciphers

Amongst all block cipher algorithms, the most widespread and probably most secure are DES, IDEA [LAI90], and RC5 [RIV94]. DES, although still secure, is doomed as international standard due to its short key length, of only 56 bits, allowing a brute-force attack to find the key in an average of three hours and a half with a $1 million machine [WIE93]. Furthermore, it has been broken by Shamir and Biham [BIH93] for a reduced number of rounds and for some modified variants and modes of implementation faster than via exhaustive search. RC5 and IDEA are counted amongst the possible candidates to become international data encryption standard. Actually, IDEA has been recently adopted as encryption algorithm by Pretty Good Privacy (PGP).

RC5 relies on data dependent rotations to frustrate differential and linear cryptanalysis. However, in spite of the cryptographic strength of rotations, their use in RC5 turns out to be insufficient, since in each rotation only the $\log_2 w$ least significant bits of the word w commanding the amount rotated are taken into account. Consequently, the other w-$\log_2 w$ have no effect at all on the rotation. In Akelarre, in turn, the 32 bits of each block are used, 5 or 4 bits in each rotation (figures 3.1 and 3.2).

Given that the key expansion process adopted by RC5 allows the cryptanalyst to compute the expanded key table S entry by entry in reverse order, it was possible to recover that table without carrying out an exhaustive search. More specifically, the way in which rotations are used in RC5 helps an attacker considering that information about one bit of a half-input register can be spread by the rotation in the last half-round to give information about every bit of the key S[$2r$+1] [KAL95].

On the contrary, in Akelarre, due to the design structure and the key expansion method, it is impossible to perform a cryptanalysis in which the key expansion is computed, because the

knowledge of a certain K_i does not allow to compute K_{i-1} nor K_{i+1}. Nevertheless, Akelarre equals the desirable flexibility of RC5, concerning the possibility of choosing between such parameters as the number of encryption rounds and key length.

Although in [LAI90] it is stated that the 3 operations are incompatible in the sense that no pair out of them satisfies a distributive law, multiplication and integer addition satisfy a partial distributive law, stemming from arithmetic modulo $2^{16}+1$, which has been exploited in the cryptanalysis of the first two rounds of IDEA [MEI93], even though it is not suitable for the attack of complete 8-round IDEA. However, Akelarre overcomes this drawback by not using multiplications but data dependent rotations.

Moreover, regarding facility of implementation and speed, Akelarre has the advantage over IDEA of not using multiplications as nonlinear operations, which are time and resource consuming, but rotations, of immediate computation in all microprocessors. However, because of the heavy use of rotations, Akelarre's and IDEA's encryption round speeds are similar, for the same number of rounds. Presumably, hardware implementations of Akelarre would be faster and more feasible, provided that hardware rotation is much easier and much more economic than multiplication.

Another shortcoming in IDEA is its key schedule: the 128-bit key is partitioned into 8 sub-blocks that are directly used as the first eight key sub-blocks. Next, the key is shifted to the left by 25 positions, after which the resulting 128-bit block is again partitioned into eight sub-blocks that are taken as the next eight key sub-blocks, and this procedure is repeated until all 52 key sub-blocks have been generated. Hence, the knowledge of sub-keys gives the attacker information about the original 128-bit user key, which could be completely obtained, yet with Akelarre's expansion method it is not only impossible to know the key sub-blocks previous to a recovered k_i but to compute key sub-blocks posterior to k_i as well.

7. Conclusions

A new block cipher, Akelarre has been proposed. One of its characteristic features is the heavy use of data dependent rotations of prime-length registers, that is, the amount of rotations performed depends on the input data and the key and, therefore, it is not predetermined.

A second distinguishing feature is the design concept of mixing arithmetic operations from different algebraic groups having the same number of elements.

The complete diffusion requirement is satisfied after a single round, as it can be observed in table 4.1. The influence of individual plaintext or key bits should spread over all the ciphertext bits, so that the change in one plaintext or key bit causes the change of about 50% of the ciphertext bits, uniformly distributed all along the block. Diffusion is provided by the structure called addition-rotation, represented in figure 3.2, which is an invertible transformation and has a complete diffusion effect in the sense that each output sub-block bit depends on each input sub-block and key bit.

The similarity of encryption and decryption means that in order to decrypt Akelarre it is enough to repeat the same transformations performed throughout the encryption process. As we are dealing with an involution, its repeated application with suitable keys, leads to the original data.

The key expansion algorithm allows the generation of as many sub-keys as necessary from a user-defined key of arbitrary length. These sub-keys are pre-computed in a way which prevents the recovery of the initial user key or the computing of the expansion starting from a known sub-key.

It is still soon to resolve about Akelarre's security, but it seems that either data dependent rotations, the mixing of operations from different groups and the key expansion scheme adopted, work together against possible cryptanalysis. After the first tests we recommend the use of Akelarre with four rounds and 128-bit length key for most purposes, but more research on algorithm security is been carried on, and final recommendations may be modified.

Acknowledgement - This paper was supported with C.I.C.Y.T., Spain, under grant TIC95-0080 (project "*Sistema criptográfico de protección de datos para red digital de servicios integrados RDSI*").

8. Bibliography

[BIH 93] Eli Biham y Adi Shamir: Differential Cryptanalysis of the Data Encription Standard. Springer-Verlag, 1993.

[FUS 91] A. Fúster, D. de la Guía, J. Negrillo, F. Montoya. Diseño e implementación de algoritmos de generación de secuencias binarias. Actas de la I Reunion Española sobre Criptografia. Palma de Mallorca, 1991.

[FUS 93] A. Fúster, D. de la Guía, J. Negrillo, F. Montoya, Estructura no lineal para la generación de secuencias pseudoaleatorias. Patent application 9300561. Spain 1993

[GUI 91] A. Fúster, D. de la Guía, J. Negrillo, F. Montoya. Estructuras no lineales para la generación de secuencias binarias de Aplicación Criptográfica. Actas del IV Simposium Nacional de la Unión Científica Internacional de Radio, 904 -- 908.Cáceres, 1991.

[GUI 94] D. de la Guía Martínez, F. Montoya Vitini, E. Valderrama, L. Porta, ASIC_CRIPTO: un circuito integrado para el modulo de seguridad del PLANBA. Actas de la III Reunión Española sobre Criptografía. Barcelona, 1994.

[GUI 96] D. de la Guía, A. Fúster. Estudio de autómatas celulares para criptografía. To be published in Actas de la IV Reunión Española sobre Criptografía. Valladolid, 1996.

[KAL 95] B. S. Kalisky, Y. L. Yin, On Differential and Linear Cryptanalysis of the RC5 Encryption Algorithm. Advances in Cryptology-CRYPTO'95, 171--184, 1995.

[LAI 90] X. Lai, J. Massey, A Proposal for a New Block Encryption Standard. EUROCRYPT 90, 389-404. Springer Verlag, Berlin 1991

[MEI 93] W. Meier, On the Security of the IDEA Block Cipher. Advances in Cryptology-EUROCRYPT'93, 371--385, 1993.

[RIV 94] R.L. Rivest, The RC5 Encryption Algorithm. In Proceedings of the Workshop on Cryptographic Algorithms, K. U. Leuven, 1994.

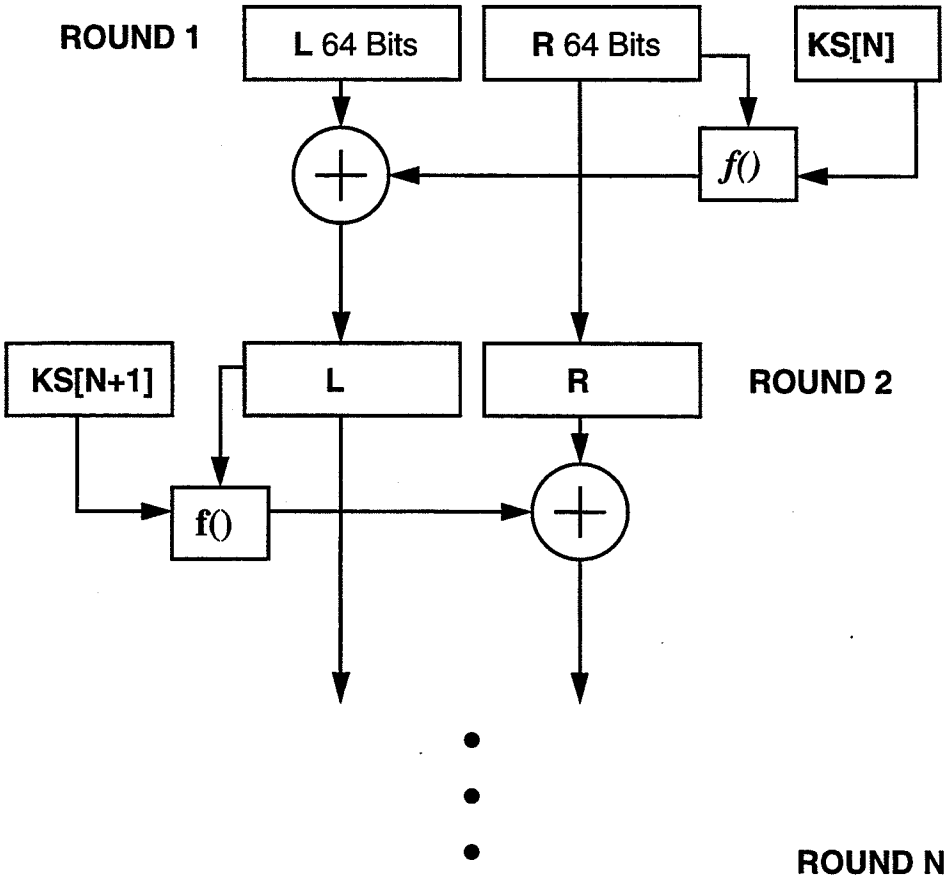
[WIE 93] M. J. Wiener, Efficient DES Key Search. CRYPTO '93, 1993.

CRISP: A Feistel cipher with hardened key-scheduling

Marcus Leech, Nortel Technologies

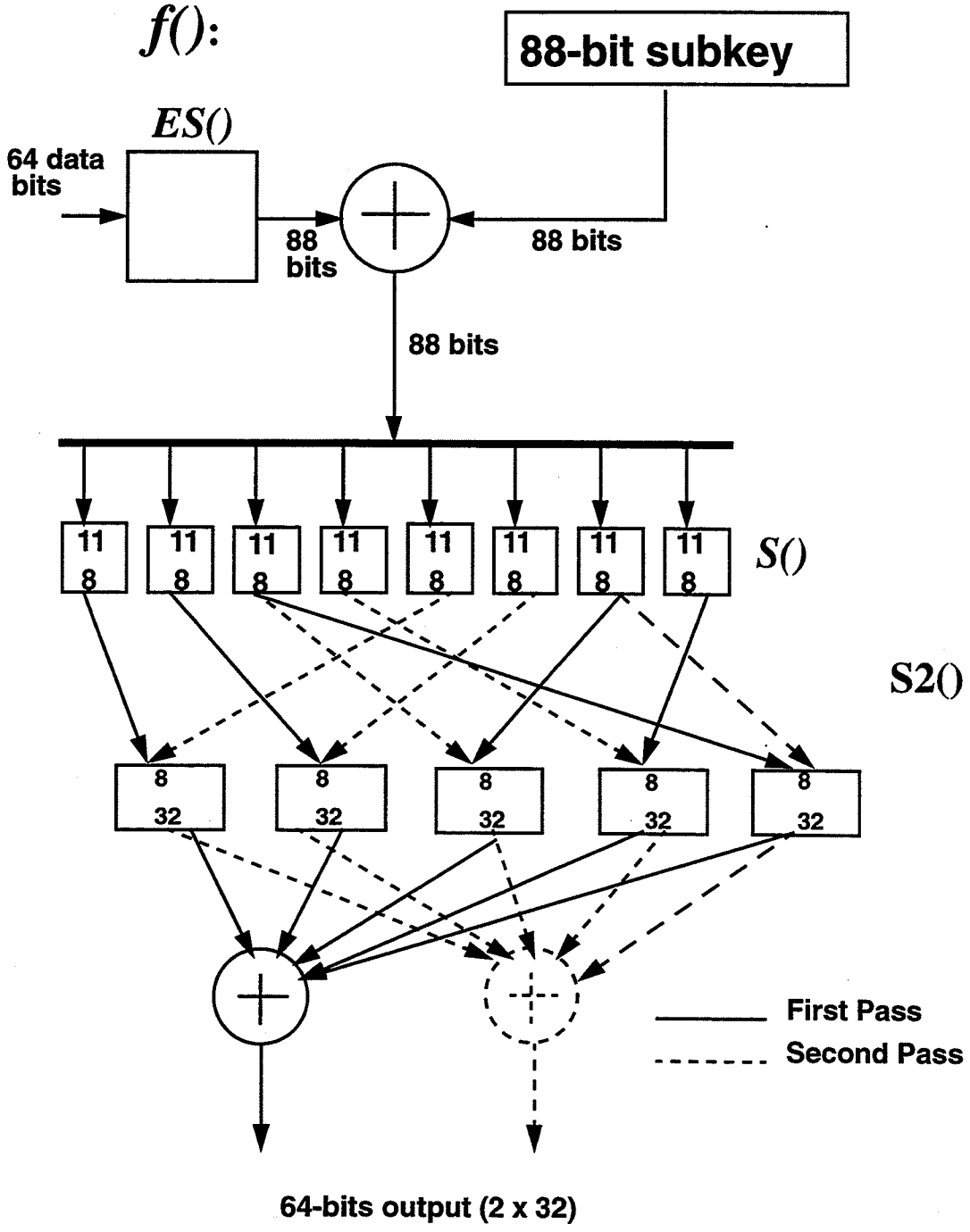
Algorithm

This paper describes a new Feistel block cipher, *CRISP*, that uses itself as a PRNG in the key-scheduling function. The cipher consists of 6 rounds in which the left and right half input blocks are alternately modulo-2 added to a non-linear function of the other half input block, and the current key schedule bits.

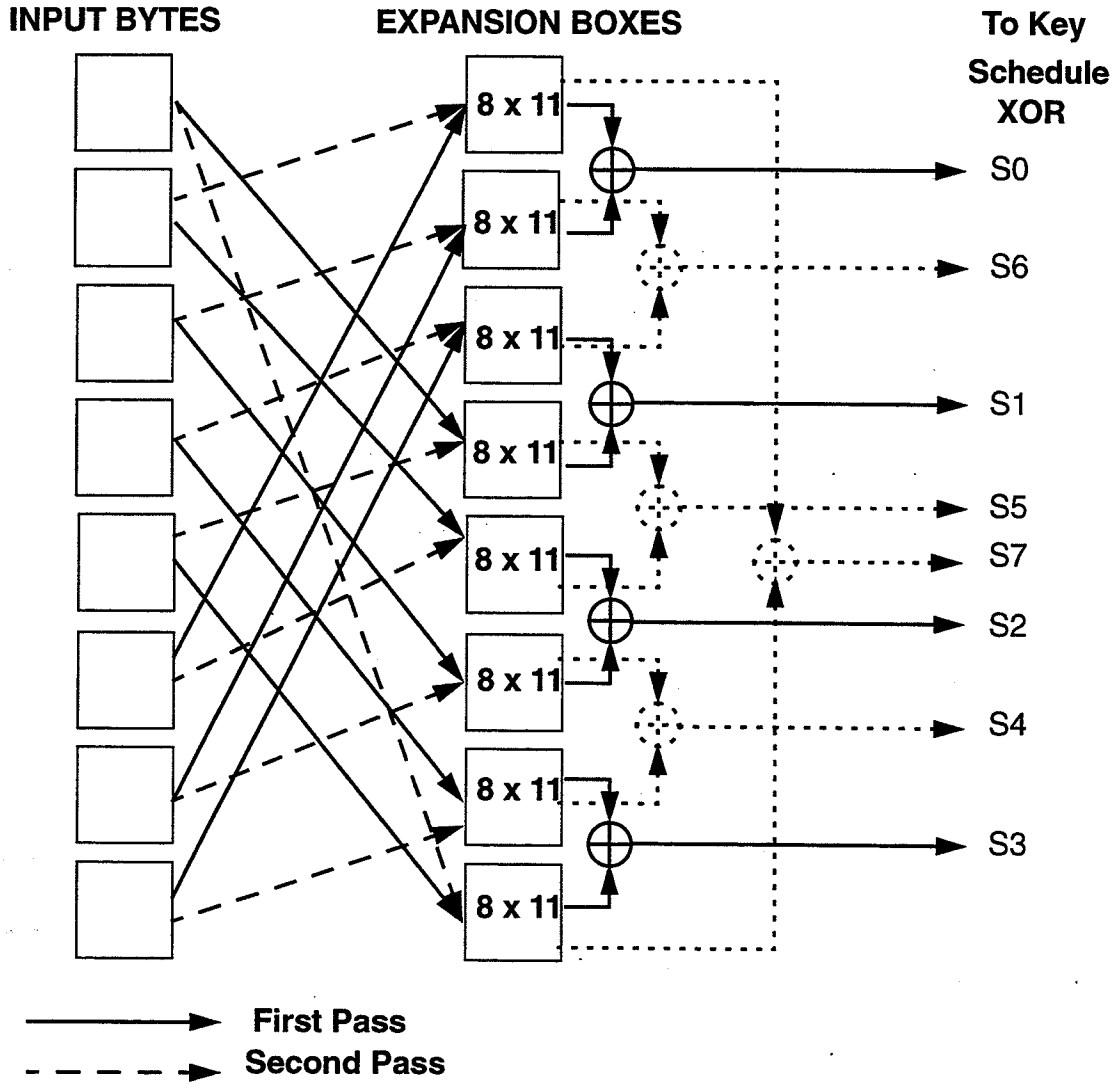


The cipher uses a 128-bit key, with a 128-bit data block. The non-linear round function, *f()*, computes a permute-substitute function of the current 88-bit key-schedule bits with the 64 input bits. The 64-bit input is first expanded to 88-bits using XOR-combined 8x11 bit S-boxes. The result of the expansion is then XORed with the 88 key bits, and fed through eight 11x8 S-boxes. The output of the S-boxes is then processed through *S2()*, a function that uses five 8x32 S-boxes that are

generated in a key-dependant way.



The *ES() expansion* function is defined as follows:



Example values for the *ES()* function tables are listed in Appendix A.

S-Box design methodology

The primary S-boxes in *CRISP* are constructed according to design principals described in an article on S-box design by Gordon and Retkin[1]. In that article, they make the claim that the probability of linearity of an S-box is proportional to the inverse of the factorial of its size. Each S-box is based on a composition of eight 8x8 reversible, randomly permuted S-boxes. That is, for an 11-bit input, the low-order 8 bits select an 8-bit value from an 8x8 S-box, while the high-order 3 bits select which 8x8 S-box to use. This is identical to the structure used in the DES S-boxes.

The primary S-boxes are generated using a C program designed to find S-boxes with a given threshold pairs-XOR count and threshold linearity; the program generates random S-boxes, measures the maximum pairs-XOR count, and linearity and discards any S-boxes whose pairs-XOR count is above the threshold, or whose linearity is above the threshold. Linearity is computed by

measuring the hamming distance between all possible output vectors (combined under XOR) against all linear-boolean function vectors of the input bits. The resulting minimum hamming distance is then compared to the threshold; S-boxes with a lower minimum hamming distance are rejected.

If the resulting S-box passes both the differential and linearity tests, it is also tested against the first-order *Bit Independence Criterion* test, to ensure that no pairs of S-box output bits change together more than 50% of the time, when the input changes by a single bit.

The evaluated version of *CRISP* uses S-boxes with a pairs-XOR threshold value of 30, and minimum hamming distance of 0.45215 (926 /2048). The value 30 for the pairs-XOR threshold was chosen because of a currently-uninvestigated runtime complexity phenomenon. When generating random S-boxes in this way, the execution time of the generator increases non-linearly as the threshold value decreases. It was determined that below a threshold value of 30, the program tended towards infinite execution time. Initially, it was thought to be an artifact of the random-number generator in use, so a new one was inserted, with exactly the same result. In any case, the goal of the generator program is to reduce the maximum pairs-XOR count towards the perfect value, which in the case of 11x8 S-boxes is 8 ($2^{11} / 2^8$). The value 30 corresponds to a single-round, single S-box probability of 1.46×10^{-2}.

The *S()* generation process requires approximately 40 CPU-hours on an HP9000/735.

Examples of S-boxes that correspond to the selection criteria are listed in Appendix B.

S2() function design

The S2() consists of five 8x32 S-boxes, generated in a key-dependant way. These 8x32 S-boxes are used twice on the 8-bit outputs of the primary S-boxes to produce a 64-bit final result.

The *S2()* function provides an extra stage of confusion and diffusion within the round function. It has the important added benefit of adding to the overall complexity of differential cryptanalysis, reducing the single round, single S-box probability from 1.46×10^{-2} to 8.82×10^{-7} ($0.0146 * (0.0078)^2$). This comes from recent results[2] on the differential cryptanalysis properties of random 8x32 S-boxes.

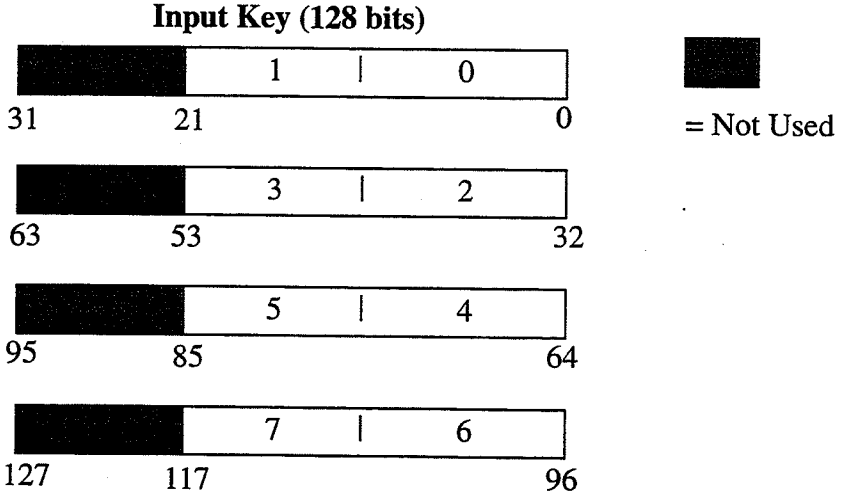
Since the contents of the *S2()* S-boxes are unknown to the cryptanalyst, both linear and differential cryptanalysis are significantly hampered.

Subkey generation

Subkeys are generated in such a way that if a given subkey is determined by cryptanalysis, it is cryptographically difficult to determine the other subkeys from the known subkey.

This is achieved by using the basic *CRISP* encryption function as a pseudo-random number generator, using the key as a seed. This is accomplished in a multi-step process, described below.

First, a "standard" key-schedule is loaded into the *CRISP* function, the standard key-schedule is derived from the first 48 entries in table 0 and table 7 in the *ES()* function, combined with XOR. This standard key-schedule is then perturbed by selecting bits from the input key, and XOR combining them with the "standard" key schedule, 11 bits at a time. A total of 88 bits from the input key are selected for use in perturbing the "standard" key schedule, as follows:



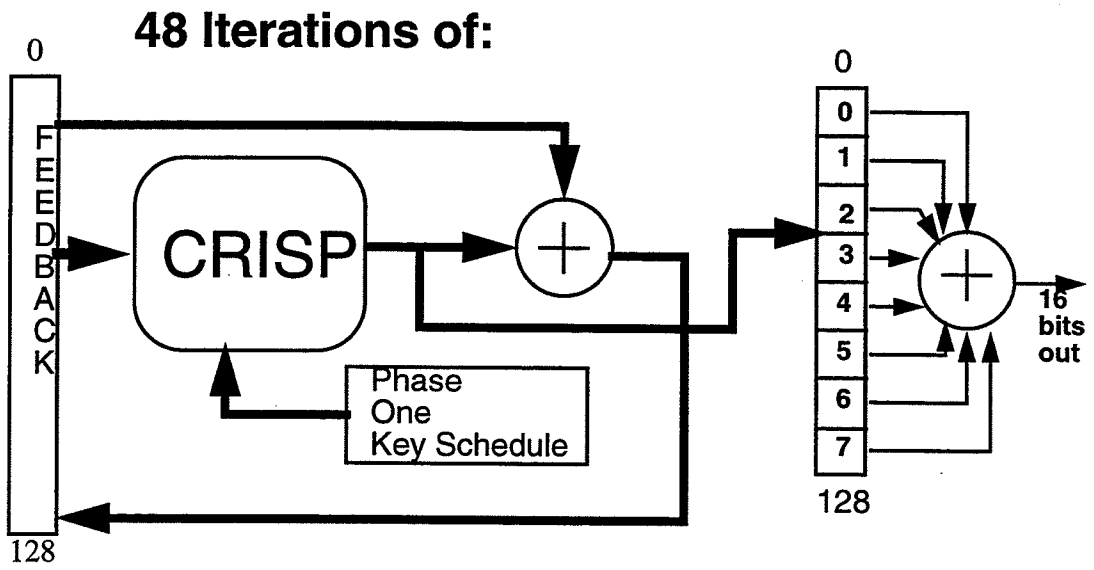
The 48 entries from the ES[0,7] XOR are grouped into six sets of eight elements, producing the following table.

Table 1: Standard Key Schedule

Round	0	1	2	3	4	5	6	7
1	4CC	079	4AA	7BC	6C8	573	3DE	5EC
2	63F	6EF	2BF	1AE	7F2	253	595	42E
3	5E3	24B	7CB	1D9	324	341	2E6	1E2
4	142	47C	26D	593	151	028	23D	004
5	527	39F	30C	217	01D	7A6	55B	1DB
6	7FA	271	64E	4B4	316	53A	2B8	3A9

Each row in this table is XORed with the corresponding (0 through 7) 11-bit value extracted from the key. This slightly-perturbed key-schedule (*phase one* key-schedule) is then used in a feedback execution of *CRISP*, to produce a new key-schedule. The feedback begins by using the key as the initial cleartext, on each iteration, the feedback buffer is updated by XOR with the CRISP ciphertext output. This *phase two* key-schedule is produced by using each output of the feedback execution of *CRISP* to produce 11-bit key-schedule elements that update the *phase one* schedule by one

element on each iteration, for a total of 48 iterations. The diagram below illustrates this concept:



The final key schedule is produced by again using *CRISP* in a feedback mode, with the input key as the initial cleartext, using the *phase two* key schedule, and the standard *S2()* function. Each ciphertext output is considered as eight 16-bit values, each of which is XORed together, then masked down to 11 bits to produce a key-schedule element. This process is repeated until all of the key-schedule elements have been filled. There are eight 11-bit elements per round, with six rounds in the evaluated implementation, for a total of 48 key schedule elements or 528 key schedule bits.

Generation of the *S2()* function

The *S2()* function is computed in a similar fashion to the final key-schedule, using *CRISP* in feedback mode. This feedback execution is a continuation of the feedback execution used in generating the final key-schedule. Each output of the *CRISP* execution is considered as four 32-bit values. The values are combined using XOR, with the resulting value being placed in the next available *S2()* table element. If the 32-bit value has already been used in an *S2()* table element, it is discarded and a new value is generated.

There are five *S2()* tables, each with 256 entries, for a total of 1280 32-bit elements.

Comparison of *CRISP* and *DES* round functions

The round function of DES takes a 32-bit input, and computes a non-linear function of that 32-bit input. It accomplishes this using four discrete steps. The 32-bit data input is expanded using the E expansion, then mixed with the 48-bit key-schedule bits. The resulting 48-bit value is then non-linearly substituted using the eight 6x4 S-boxes. The final step is to permute the 32-bit S-box output using the P permutation.

When examining the *E* expansion in DES, notice that it provides no guarantee that a given input bit can affect more than one S-box. This makes differential cryptanalysis easier, since single S-

boxes can be "isolated" for differential cryptanalysis purposes.

The cryptographic significance of the *P* permutation is assumed to be for the purposes of improving the diffusion properties of the round function, since the *E* expansion provides rather less diffusion.

The *CRISP* algorithm has the same basic structure in its round function as DES. The round function takes a 64-bit input, expands it to 88 bits using the ES function, mixes it with the key, and non-linearly substitutes the 88-bits using eight 11x8 S-boxes. When examining the ES function, observe that each input bit affects two S-boxes, thus making differential cryptanalysis somewhat harder. The ES function also provides, as a secondary effect, a small amount of non-linearity, since it acts as a 16x11 S-box. The *CRISP* S-boxes, due to their size, provide higher a degree of resistance both to differential and linear cryptanalysis than DES.

In *CRISP*, the post S-box function, *S2*, corresponds roughly to the *P* permutation in DES. Observe that *S2* provides a non-linear transform of the S-box outputs, while the *P* function in DES is entirely linear. The *S2* function also improves resistance to both differential and linear cryptanalysis, since the *S2* table elements are unknown to the cryptanalyst. Even if the cryptanalyst is able to determine the contents of S2, it is assumed that the analysis of random 8x32 S-boxes, as described in [2], would hold for the *S2* function within *CRISP*.

Analysis of key-scheduling and *S2()* generation

The strength of the key-scheduling and *S2()* function generation algorithm is predicated on the ability of the concatenation of round functions to act as a random, non-linear transform of the input key material. The avalanche results shown later tend to suggest that *CRISP* does act as a strong random transform, with good per-round non-linearity; the assumption is that the concatenation of rounds produces a non-linearity that is close to the product of the non-linearity of the round function.

The purpose of the key-schedule algorithm is to produce a sequence of bits from the input key material that can be used as per-round keys. Many encryption functions use a key-schedule algorithm in which the round key bits are related to the input key in a way that is linear. The DES, for example, uses a series of rotates and selects to produce round key material. This makes DES slightly more vulnerable to differential cryptanalysis[4] than DES with purely-random, independent round keys. This occurs because determination of one or more per-round key bits results in determination of related bits in other rounds.

It has been proposed, most recently in [5], that the DES can be strengthened somewhat against both linear and differential cryptanalysis by using the DES cipher itself as a PRNG in the generation of key-schedule bits.

The Blowfish[7] cipher also uses itself as a PRNG in the generation of both its S-boxes, and in the generation of the *P()* round function.

An early version of *CRISP* used MD5[6] as a PRNG in the generation of round keys, but it was

felt to be overly complex, and not as compact as a key-scheduling algorithm that uses the *CRISP* cipher itself as a PRNG.

If we assume that the cryptanalyst is able to determine the round key at a particular round, they must be able to determine the plaintexts that correspond to the partial ciphertexts that constitute the determined round key. Each 11-bit round-key element is the XOR of eight 11-bit sections of a *CRISP* ciphertext, under an unknown key, with unknown plaintext. The problem, then, is to determine the full 128-bit ciphertext, and the corresponding plaintext, under an unknown key.

The cryptanalyst has a similar problem to solve when they are able to determine some values within *S2()*. They must first determine, for a given determined 32-bit S-box entry value, the corresponding 128-bit ciphertext, then the key-schedule and plaintext that produced it.

The performance of key-scheduling is entirely dependant on the performance of the *CRISP* algorithm itself. The *CRISP* algorithm is called approximately[1] 1376 times in setting a new key. On the hardware the algorithm was tested on, this corresponds to 20 milliseconds of real time. This could be improved by changing the algorithm that produces *S2()* to produce four *S2()* elements at a time from the full-width 128-bit output of *CRISP*.

Avalanche Results

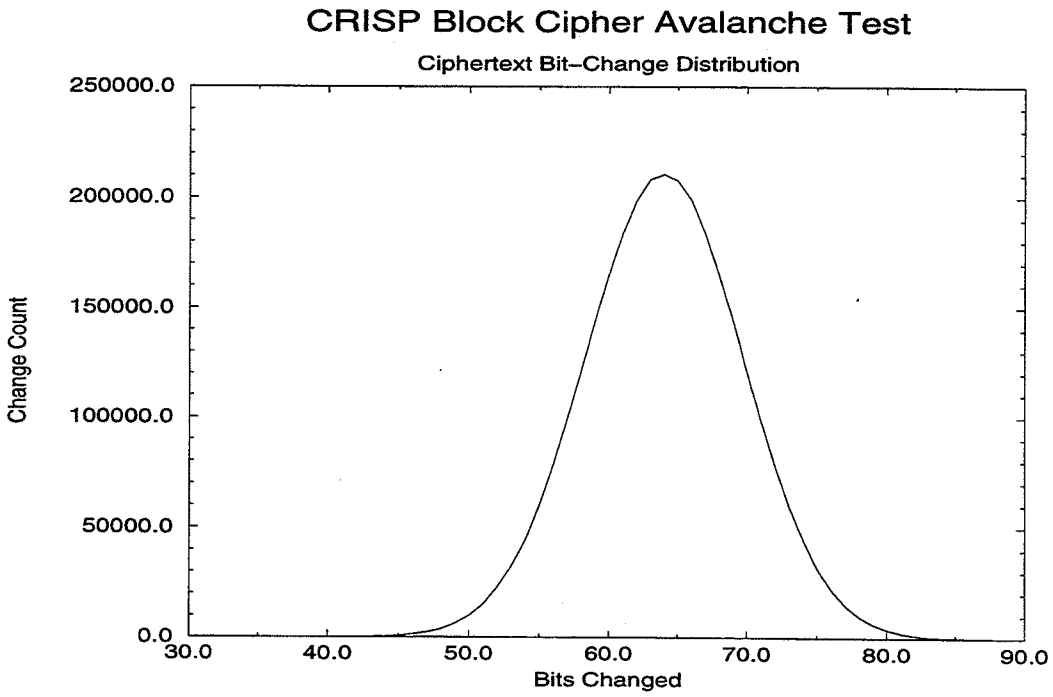
The avalanche properties were measured by iterating the *CRISP* encryption function 3,000,000 times, using a fixed, random key, and a random data input that is modified randomly by one bit on each iteration. This process was repeated several times.

This results in an average change in the resulting ciphertext of 64 bits, which is 50% of the total ciphertext bits. The minimum change ranges from 35 to 37, while the maximum ranges from 85 to 92. The minimum ciphertext change corresponds to somewhat more (0.273 to 0.289) than 25% of the total bits in the ciphertext.

The DES under similar test conditions tends to produce an average ciphertext change of exactly 50% of the bits, while the minimum is usually somewhat less (0.203 to 0.234). The following

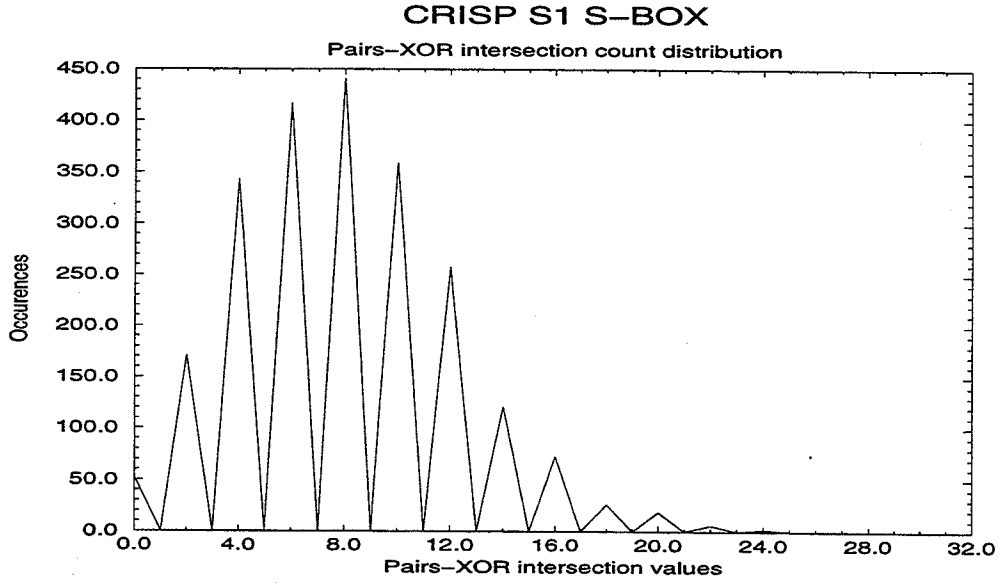
1. Since the *S2()* generation process can potentially call *CRISP* a variable number of times, due to its selection criterion.

graph shows the distribution of ciphertext bit-changes for 3,000,000 iterations of the function:



Differential Cryptanalysis Results

The pairs-XOR count distribution graph shows that the mean value for a pairs-XOR "intersection" over the entire S1 box from *S()* is 8:



This graph requires some explanation. The usual method to show pairs-XOR distribution uses a table, with the output-XOR as columns, and the input-XOR as rows. The elements of such a table convey the distribution of pairs-XOR values over the given S-box. The *CRISP* S-boxes are 11x8, which means the resulting table would have 256 columns and 2048 rows; values that yield a rather ungainly tabular display. The graph conveys the same overall information, showing that the "intersection" values are clustered around the so-called perfect distribution that would be achieved if each intersection in the pairs-XOR table were equally likely. In the *CRISP* case, that perfect distribution value would be 8 ($2^{11} / 2^8$).

We can observe from the pairs-XOR distribution, that all of the primary S-boxes have the same maximum pairs-XOR value of 30 (or a probability of 0.0146). There is no obvious advantage to attacking a particular S-box over another.

The *ES()* function effectively acts as a fixed 16-to11 bit mapping between bits of the round function input, and input bits to a single S-box. The following table illustrates the mapping:.

Table 2: ES function input mapping

Input Octet Pair	ES box pair	S-box affected
5 and 6	0 and 1	S0
7 and 0	2 and 3	S1
1 and 2	4 and 5	S2
3 and 4	6 and 7	S3
7 and 6	6 and 5	S4
5 and 4	4 and 3	S5
3 and 2	2 and 1	S6
1 and 0	0 and 7	S7

In effect, a new set of 16-by-11 bit S-boxes are synthesized by the input mapping to *ES()*.

The problem for the cryptanalyst, then, is to find input octet pairs that produce the maximum differential probability (by minimizing the number of S-boxes involved, and by selecting the highest probability for each S-box involved). In DES, the pre-S-box expansion function, *E*, has the property that for input pair X_1 and X_2 the equation $E(X_1)\ XOR\ E(X_2) = E(X_1\ XOR\ X_2)$ is always true. In *CRISP*, the equivalent $ES(X_1)\ XOR\ ES(X_2) = ES(X_1\ XOR\ X_2)$ is true for only a small number of input pairs (approximately 1 in 2500). This means that a different approach is required when engaging in differential cryptanalysis. The cryptanalyst must search for input pairs that satisfy the equation $ES(X_1)\ XOR\ ES(X_2) = I$, where I is a desirable input XOR to a target S-box. Because each such input pair controls both the so-called *target* S-box, and partially controls the input to

two other S-boxes, via different *ES()* boxes, it is thought to be difficult to find input pairs that simultaneously satisfy desirable input XOR conditions for the S-boxes they control.

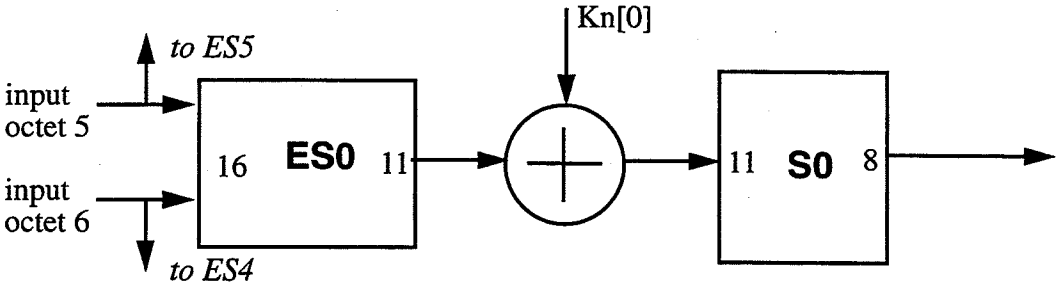
Initial analysis of the density of pairs satisfying the criterion of a high-probability XOR in one S-box, while having a zero XOR in the two other S-boxes linked via common input octets is estimated to be 1 in 2^{28}, for randomly selected inputs. For example, S-box 0 is linked to S-box 4 and S-box 5 via input octets 5 and 6. This means that the input to these three S-boxes is fully defined by 4 input octets, (octets 4,5,6 and 7). The following input pairs for octets 4,5,6,7 satisfy the above criterion:

X_1=7068790E X_2=D68D3980
X_1=09FC3D28 X_2=2898D918
X_1=493238F2 X_2=CBF8D398
X_1=40D72FF9 X_2=5465FB57

The above input values do not necessarily guarantee zero input XOR to the other S-boxes affected by input octets 4 and 7 (S-boxes 3 and 1).

It is surmised that inputs satisfying the more stringent criterion of having a high-probability input XOR in one S-box, while having zero XOR in all the others have a very low density. Similarly, inputs satisfying the criterion of having high-probability in two S-boxes, while having zero XOR in all other S-boxes is of a similar density. Initial testing shows that the density may be less than 1 pair in 2^{34} random input pairs.

The combined ES() and S() can be re-arranged (with S0 as example), as follows:



Given the above arrangement, we can compute the input XOR distribution for ES() that yields the high-probability input XORs to the corresponding S() S-box. The S0 box, for example, has 12 high-probability inputs XORs (that is input XORs, that have output XORs occurring with probability 0.0146). In S0, a high-probability input XOR is X'6AE'. An input XOR of X'03AA' to ES0 leads to an ES0 output XOR of X'6AE' which in turn leads to an output XOR in S0 of X'17', with compound probability 1.79 x 10^{-5}. The following table shows examples of the highest com-

pound probabilities of ES0 input XORs producing a given S0 output XOR.

Table 3: Example ES0/S0 XOR probabilities

ES[0] Input XOR	S[0] Output XOR	Probability
X'054F'	X'E9'	2.15E-5
X'0DAD'	X'B8'	2.06E-5
X'6635'	X'97'	2.06E-5
X'700E'	X'17'	2.24E-5
X'7B13'	X'97'	2.15E-5
X'81CB'	X'DA'	2.06E-5
X'8233'	X'4D'	2.06E-5
X'FD28'	X'97'	2.06E-5

Since each input octet controls two S-boxes, a reasonable assumption to make is that at least two S-boxes must be "involved" in a given single-round characteristic, thus giving a maximum single-round probability near 4.8×10^{-10} (X'700E' to X'17'). If we assume that a six-round characteristic can be constructed in which half the rounds have probability 1, and half the rounds have the probability 4.8×10^{-10}, then without S2(), *CRISP* is theoretically vulnerable to differential cryptanalysis, since the resulting probability near 1.10×10^{-28} is greater than the probability near 2.9×10^{-40} that would be necessary to make *CRISP* unconditionally resistant to differential cryptanalysis. If, however, the best characteristic that can be constructed uses the probability of 4.8×10^{-10} in all but one round, with a probability 1 characteristic in one round, then *CRISP* would be unconditionally resistant to differential cryptanalysis, since the resulting probability is near 2.5×10^{-47}. No attempt has yet been made so find the best 6-round characteristic for *CRISP*, since *S2()* is assumed to defeat differential cryptanalysis.

If an *average-case S2()* function is factored into the differential probability analysis, then the algorithm is unconditionally resistant if the best six-round characteristic has probability 1 in three of the rounds, and probability near 2.92×10^{-14} in the other rounds, producing an aggregate probability near 2.55×10^{-41}.

Appendix C contains complete tables of high-probability XORs for the eight ES/S combinations.

Linear Cryptanalysis Results

Work on linear cryptanalysis is in progress at the time of writing.

Resistance to the Birthday Paradox under CBC

Under Cipher Block Chaining mode, this cipher is more resistant than ciphers with smaller block

sizes to the Birthday Paradox, which states that, if $2^{n/2}$ plaintexts are encrypted under CBC (where n is the blocksize in bits), the probability of there being two equal ciphertexts is 0.5. If two identical ciphertexts correspond to different plaintexts, then there exists a known XOR relation between the two plaintexts.

Since *CRISP* has a 128-bit block, the probability is vastly less than with 64-bit ciphers.

Resistance to attacks based on non-surjective round functions

An early version of the algorithm used four S-boxes in *S2()*. This produced a round function that was non-surjective (approximately 40% of the 2^{64} outputs were impossible). This lead to the round function being theoretically vulnerable to an attack described by Bart Preneel in [3].

In practice, such an attack is unlikely to succeed, due to the very large tables that must be constructed, on the order of $2^{62.5}$ elements, or approximately 2^{65} bytes. Because the round-keys are 88 bits, even with a conservative estimate that the entropy is lower than the 88 bits suggested by the key size, an attack is also unlikely to succeed, even when enough table space is available.

CRISP was made substantially more resistant to this attack by the addition of a fifth S-box in the *S2()* function, thus making less than 4% of the 2^{64} outputs impossible.

The performance penalty for implementing this was approximately 12.5%, with a 1K byte memory penalty.

Performance and Memory Requirements

The algorithm was implemented in C, using the GNU-C compiler on an HP-9000/735. The 6-round version produces an encryption rate of approximately 45,000 encryptions per second, or an equivalent data rate of 6.14Mbits per second. Using the native HP/UX compiler produces an approximate 4% performance improvement. There are opportunities for optimization; in particular, the S-box outputs may be composed with the *S2()* function in a single table, reducing the number of table-lookups required in *f()* by 30%.

The tables used to implement *ES()*, *S()*, and *S2()* consume only 25K bytes of memory, which is easily within reach for a microprocessor/embedded controller implementation. The executable code on an HP9000/7XX system is approximately 3K bytes. Implementation complexity can be reduced by changing the key-scheduling algorithm to produce only the key-schedule elements, and dispense with key-dependance in the *S2()* algorithm; the resulting cipher is then potentially subject to standard differential, and linear cryptanalytic attack.

Acknowledgments

The author gratefully acknowledges the patient tutoring, and expert advice of Carlisle Adams, both in the analysis of the *CRISP* algorithm, and in reviewing early drafts of this paper.

The author would also like to thank his management at *Nortel Technology* for allowing him to pursue topics that are only tangentially related to his main job.

References

[1] J.A. Gordon, H. Retkin, *Are big S-boxes best?*, Lecture Notes in Computer Science nr. 149, 1983.

[2] J. Lee, H. Heys, S. Tavares, *On the Resistance of the CAST Encryption Algorithm to Differential Cryptanalysis, SAC '95: Workshop Record*, pages 107-119.

[3] Preneel, B., *On Weaknesses of Non-Surjective Round Functions, SAC '95: Workshop Record*, pages 100-106

[4] E. Biham, A. Shamir, *Differential Cryptanalysis of DES-Like Cryptosystems*, Journal of Cryptology, vol.4, 1991, pages 3-72.

[5] U. Blumenthal, S. Bellovin, *A Better Key Schedule for DES-like Ciphers*, paper to appear at PragoCrypt-96, September, 1996.

[6] R. Rivest, *The MD5 Message Digest Algorithm*, published as RFC1321, April 1992.

[7] B. Schneier, *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*, Fast Software Encryption, Cambridge Workshop Proceedings, 1994, pages 191-204.