CS-CIPHER

Jacques Stern and Serge Vaudenay *

Ecole Normale Supérieure — CNRS
e-mail: {Jacques.Stern,Serge.Vaudenay}@ens.fr

Abstract. This paper presents a new block cipher which offers good encryption rate on any platform. It is particularly optimized for hardware implementation where the expected rate is several Gbps on a small dedicated chip working at 30MHz. Its design combines up to date state of the art concepts in order to make it (hopefully) secure: diffusion network based on the Fast Fourier Transform, multipermutations, highly nonlinear confusion boxes.

Recent explosion of the telecommunication marketplace motivates the research on encryption schemes. Trading security issues pushed the US government to start the development of the *Data Encryption Standard* in the 70's [1], all telecommunication devices now need to be secured by encryption. Many attacks have been proposed against DES including Biham and Shamir's differential cryptanalysis [5, 6] and Matsui's linear cryptanalysis [15, 16]. Still the best practical attack seems to be exhaustive search, which has become a real threat as shown by the recent success of the RSA Challenge [31]. In this paper we propose a new symmetric encryption scheme which has been designed in order to be efficient on any platform, included cheap 8-bit microprocessors (*e.g.* smart cards), modern 32-bit microprocessors (SPARC, Pentium) and dedicated chips.

Notations

- $||$ is the concatenation of two strings
- $\oplus$ is the bitwise exclusive *or* of two bitstrings (with equal lengths)
- R_l rotates a bitstring by one position to the left
- $\wedge$ is the bitwise *and* of two bitstrings (with equal lengths)
- bitstrings are written in hexadecimal by packing four bits into one digit (for instance, $\texttt{d2}_{16}$ denotes the bitstring `11010010`)
- the numbering of bits in bitstrings is from right to left starting with 0 (*i.e.* x_0 denotes the last bit in x)
- bitstring and integers are converted in such a way that $b_{n-1}||\ldots||b_0$ corresponds to an integer $b_{n-1}.2^{n-1}+\ldots+b_0$

* Part of this work has been supported by the COMPAGNIE DES SIGNAUX and may be subject to patent matter.

1 Definition of CS-Cipher

1.1 Use of CS-Cipher

CS-Cipher (as for the French "*Chiffrement Symétrique*", *Symmetric Cipher*) is a symmetric block cipher which can be used in any mode to encrypt a block stream (*e.g.* the Cipher Block Chaining mode, see [2]). Basically, the CS-Cipher encryption function maps a 64-bit plaintext block m onto a 64-bit ciphertext block m' by using a secret key k which is a bitstring with arbitrary length up to 128. The CS-Cipher decryption function maps the ciphertext onto the plaintext by using the same secret key. We assume that m is represented by a bitstring

$$m = m_{63} \ldots m_1 m_0$$

and we similarly write

$$m' = m'_{63} \ldots m'_1 m'_0.$$

We also assume that the string k is padded with trailing zero bits to get a length of 128 bits

$$k = k_{127} \ldots k_1 k_0.$$

(A key k is therefore equivalent to another key k' which consists in padding k with a few zero bits.)

A key scheduling scheme first process the secret key k in order to obtain nine 64-bit subkeys $k^0, \ldots, k^8$ iteratively in this order. If the secret key has to be used several times, we recommend to precompute this sequence which may notably increase the encryption rate.

The encryption algorithm processes iteratively each subkey in the right order $k^0, \ldots, k^8$ whereas the decryption algorithm processes them in the reverse order $k^8, \ldots, k^0$. We thus recommend to keep a storage of all subkeys for decryption or to adapt the key scheduling scheme so that it can generate the subkeys in the reverse order.

1.2 Key scheduling scheme

Let k be the padded 128-bit secret key. We first split the bitstring into two 64-bit strings denoted k^{-2} and k^{-1} such that

$$k = k^{-1} || k^{-2}.$$

Those strings initialize a sequence $k^{-2}, \ldots, k^8$ where $k^0, \ldots, k^8$ are the nine 64-bit subkeys to compute. The sequence comes from a Feistel scheme as

$$k^i = k^{i-2} \oplus F_{c^i}(k^{i-1})$$

for $i = 0, \ldots, 8$ where F_{c^i} is defined below (see Feistel [9]). Figure 1 illustrates the key scheduling scheme together with the encryption itself.

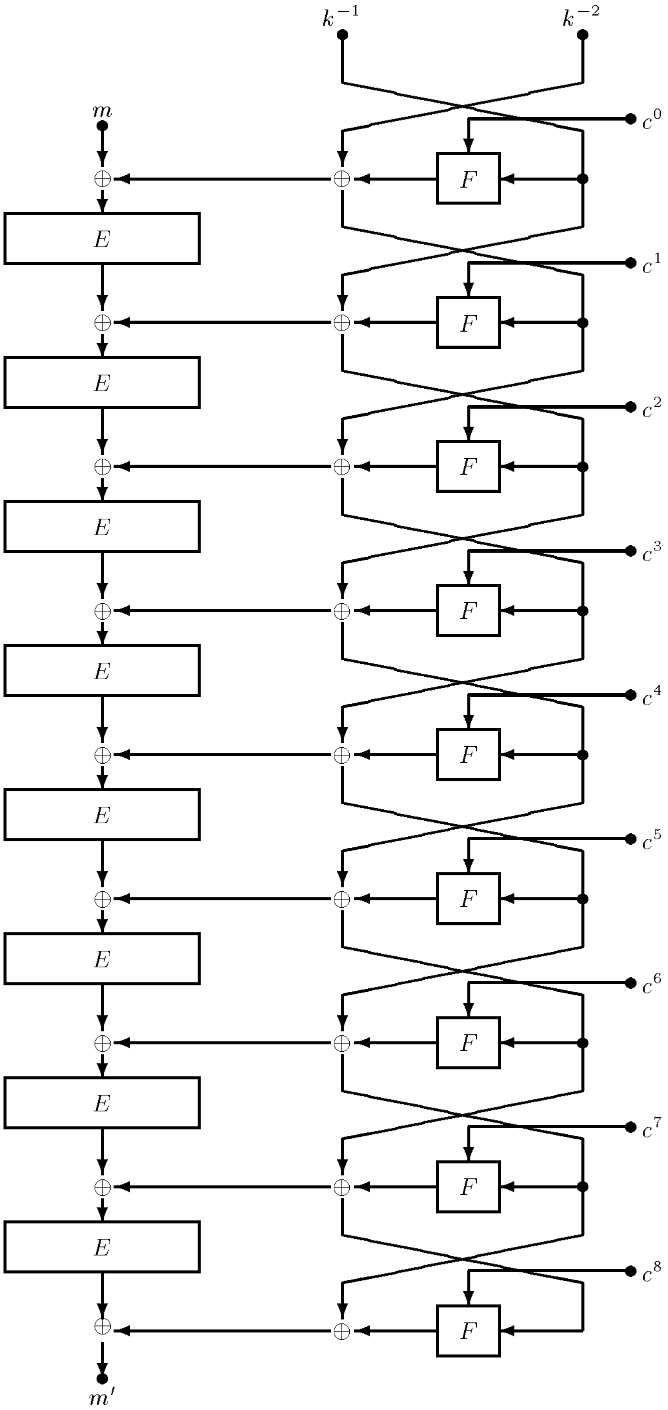

Fig. 1. Encryption process

The F_{c^i} function maps a 64-bit string onto a 64-bit string by using a 64-bit constant c^i. In the definition of CS-CIPHER, $c^0, \ldots, c^8$ are defined as the first bytes of the table of a permutation P which will be defined below:

$$\begin{aligned}
c^0 &= \texttt{290d61409ceb9e8f}_{16} \\
c^1 &= \texttt{1f855f585b013986}_{16} \\
c^2 &= \texttt{972ed7d635ae1716}_{16} \\
c^3 &= \texttt{21b6694ea5728708}_{16} \\
c^4 &= \texttt{3c18e6e7faadb889}_{16} \\
c^5 &= \texttt{b700f76f73841163}_{16} \\
c^6 &= \texttt{3f967f6ebf149dac}_{16} \\
c^7 &= \texttt{a40e7ef6204a6230}_{16} \\
c^8 &= \texttt{03c54b5a46a34465}_{16}.
\end{aligned}$$

F_{c^i} is defined by

$$F_{c^i}(x) = T(P^8(x \oplus c^i)).$$

P^8 is defined by a byte-permutation P which maps an 8-bit string onto an 8-bit string according to a table and T is a bit transposition. (Sofware implementation will use a lookup table for P whereas hardware implementation may use the inner structure of P which will be detailed below.)

Given the 64-bit string $y = x \oplus c^i$, we split it into eight 8-bit strings denoted $y_{63..56}, \ldots, y_{7..0}$ such that $y = y_{63..56}||\ldots||y_{7..0}$. We next apply the permutation P byte-wise *i.e.* we compute

$$P^8(y_{63..56}||\ldots||y_{7..0}) = P(y_{63..56})||\ldots||P(y_{7..0}).$$

The permutation T is the 8×8 bit-matrix transposition. More precisely, given the 64-bit string $z = P^8(x \oplus c^i)$, we first split it into eight 8-bit strings $z_{63..56}, \ldots, z_{7..0}$ as for y above and write it in a 8×8 bit-matrix fashion in such a way that the first row is $z_{63..56}$ and so on. The permutation T simply transposes the matrix so that the first eight bits of $T(z)$ are the first bits of $z_{63..56}, \ldots, z_{7..0}$ in this order, the second eight bits are the seconds bits, and so on. Thus we have

$$T(z) = z_{63}||z_{55}||\ldots||z_7||z_{62}||z_{54}||\ldots||z_0.$$

Figure 2 illustrates how F_{c^i} works in the key scheduling scheme.

1.3 Encryption scheme

The encryption process is performed through eight rounds by using a round-encryption function E which is a permutation on the set of all 64-bit strings. If m denotes the 64-bit plaintext block and $k^0, \ldots, k^8$ is the sequence of the 64-bit subkeys, the ciphertext block is

$$k^8 \oplus E(k^7 \oplus \ldots E(k^1 \oplus E(k^0 \oplus m)) \ldots)$$

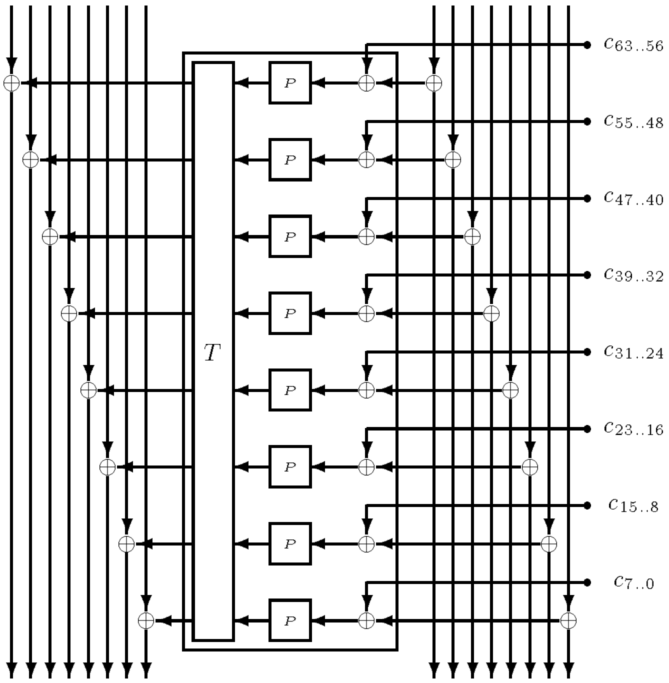

Fig. 2. The F_{c^i} function in the key scheduling scheme

as depicted on Figure 1.

The round-encryption function E is based on the Fast Fourier Transform graph and a 16-bit to 16-bit *mixing* function M as depicted on Figure 3. It also uses two 64-bit constants c and c' defined by the binary expansion of the mathematical constant

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 2,\mathtt{b7e151628aed2a6abf7158809cf4f3c762e7160f}_{16}\ldots$$

Thus we define

$$c = \mathtt{b7e151628aed2a6a}$$
$$c' = \mathtt{bf7158809cf4f3c7}.$$

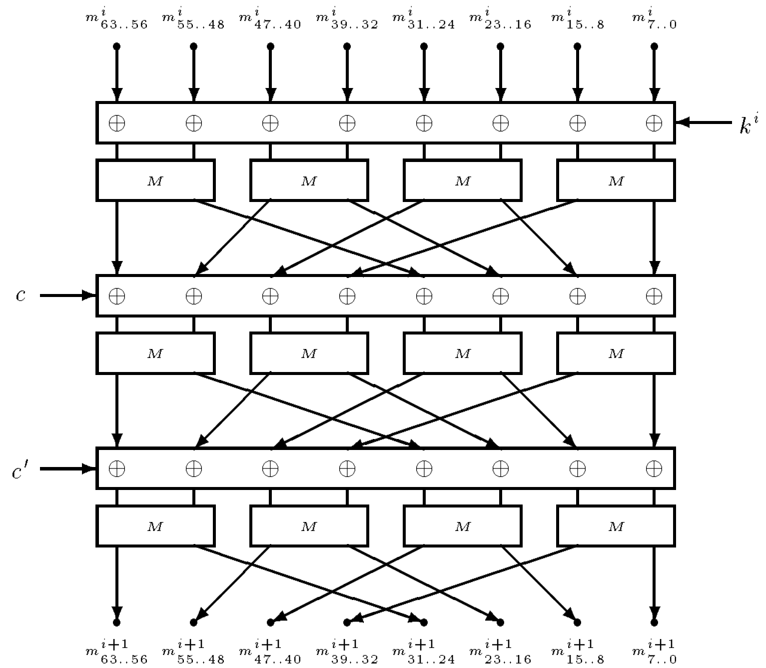

Fig. 3. One encryption round

More precisely, in each encryption round, we iterate the following scheme three times

– we xor with a constant (which is successively the subkey k^i, c and c'),

- we split the 64-bit string into four 16-bit strings and we apply M to each of it, obtaining four 16-bit strings which combine into a 64-bit string,
- we split it again into eight 8-bit strings

$$r_{63..56}||r_{55..48}||r_{47..40}||r_{39..32}||r_{31..24}||r_{23..16}||r_{15..8}||r_{7..0}$$

  and we change their order as

$$r_{63..56}||r_{47..40}||r_{31..24}||r_{15..8}||r_{55..48}||r_{39..32}||r_{23..16}||r_{7..0}.$$

The M function takes a 16-bit string x which is split into two 8-bit strings $x_l||x_r$ and computes $M(x) = y_l||y_r$ by

$$\begin{aligned} y_l &= P(\varphi(x_l) \oplus x_r) \\ y_r &= P(R_l(x_l) \oplus x_r) \end{aligned}$$

where φ is defined by

$$\varphi(x_l) = (R_l(x_l) \wedge \mathtt{55}_{16}) \oplus x_l$$

i.e.

$$\varphi(x_7||\ldots||x_0) = x_7||(x_6 \oplus x_5)||x_5||(x_4 \oplus x_3)||x_3||(x_2 \oplus x_1)||x_1||(x_0 \oplus x_7).$$

The M computation is depicted on Figure 4 (with the xor to its input which is always performed).

The P byte-permutation (which is also used in the key scheduling scheme) is defined by a three-round Feistel cipher represented on Figure 5: the 8-bit input x is split into two 4-bit strings $x_l||x_r$, we compute successively

$$\begin{aligned} y &= x_l \oplus f(x_r) \\ z_r &= x_r \oplus g(y) \\ z_l &= y \oplus f(z_r) \end{aligned}$$

where f and g are two special functions.

The function f is defined by the table

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$f(x)$	f	d	b	b	7	5	7	7	e	d	a	b	e	d	e	f

which comes from

$$f(x) = \overline{\overline{x} \wedge R_l(x)}.$$

The function g is defined by the table

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$g(x)$	a	6	0	2	b	e	1	8	d	4	5	3	f	c	7	9

which does not come from a simple expression.

Finally, the value of $P(xy)$ is given as follows by the table of P.

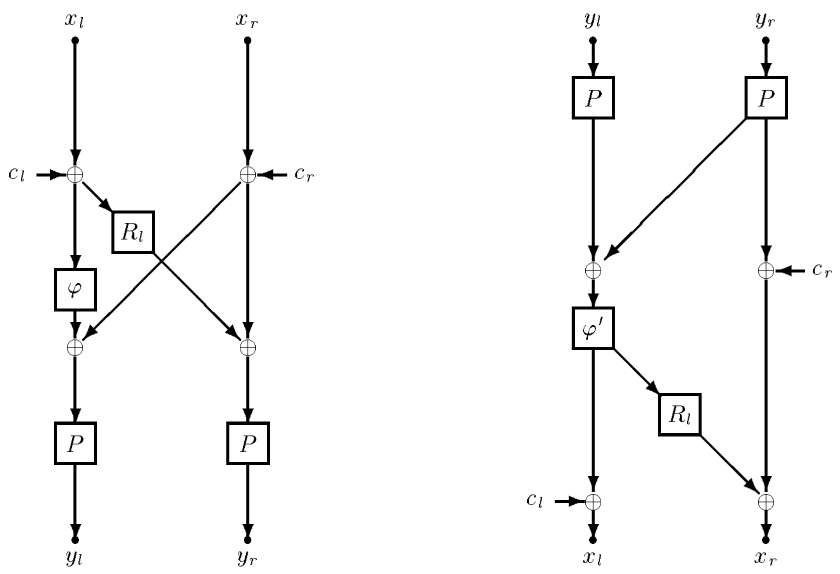

Fig. 4. Computation graph of M and M^{-1}

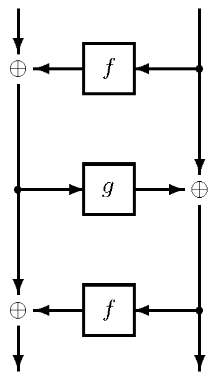

Fig. 5. The permutation P

xy	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f
0.	29	0d	61	40	9c	eb	9e	8f	1f	85	5f	58	5b	01	39	86
1.	97	2e	d7	d6	35	ae	17	16	21	b6	69	4e	a5	72	87	08
2.	3c	18	e6	e7	fa	ad	b8	89	b7	00	f7	6f	73	84	11	63
3.	3f	96	7f	6e	bf	14	9d	ac	a4	0e	7e	f6	20	4a	62	30
4.	03	c5	4b	5a	46	a3	44	65	7d	4d	3d	42	79	49	1b	5c
5.	f5	6c	b5	94	54	ff	56	57	0b	f4	43	0c	4f	70	6d	0a
6.	e4	02	3e	2f	a2	47	e0	c1	d5	1a	95	a7	51	5e	33	2b
7.	5d	d4	1d	2c	ee	75	ec	dd	7c	4c	a6	b4	78	48	3a	32
8.	98	af	c0	e1	2d	09	0f	1e	b9	27	8a	e9	bd	e3	9f	07
9.	b1	ea	92	93	53	6a	31	10	80	f2	d8	9b	04	36	06	8e
a.	be	a9	64	45	38	1c	7a	6b	f3	a1	f0	cd	37	25	15	81
b.	fb	90	e8	d9	7b	52	19	28	26	88	fc	d1	e2	8c	a0	34
c.	82	67	da	cb	c7	41	e5	c4	c8	ef	db	c3	cc	ab	ce	ed
d.	d0	bb	d3	d2	71	68	13	12	9a	b3	c2	ca	de	77	dc	df
e.	66	83	bc	8d	60	c6	22	23	b2	8b	91	05	76	cf	74	c9
f.	aa	f1	99	a8	59	50	3b	2a	fe	f9	24	b0	ba	fd	f8	55

For instance, we have $P(\mathtt{26}) = \mathtt{b8}$ since $f(\mathtt{6}) = \mathtt{7}$, $\mathtt{2} \oplus \mathtt{7} = \mathtt{5}$, $g(\mathtt{5}) = \mathtt{e}$, $\mathtt{6} \oplus \mathtt{e} = \mathtt{8}$, $f(\mathtt{8}) = \mathtt{e}$ and finally $\mathtt{5} \oplus \mathtt{e} = \mathtt{b}$.

1.4 Decryption scheme

Decryption is performed by iterating a decryption-round function represented on Figure 6.

Details of the decryption are left to the reader. We simply observe that $(x_l||x_r) = M^{-1}(y_l||y_r)$ can be computed by

$$x_l = \varphi'(P(y_l) \oplus P(y_r))$$
$$x_r = R_l(x_l) \oplus P(y_r)$$

where

$$\varphi'(x) = (R_l(x) \wedge \mathtt{aa}_{16}) \oplus x.$$

1.5 Test values

As an example we encrypt the plaintext $\mathtt{0123456789abcdef}_{16}$ with the secret key $\mathtt{0123456789abcdeffedcba9876543210}_{16}$. The subkeys sequence is

$$k^{-2} = \mathtt{fedcba9876543210}_{16}$$
$$k^{-1} = \mathtt{0123456789abcdef}_{16}$$
$$k^{0} = \mathtt{45fd137a4edf9ec4}_{16}$$
$$k^{1} = \mathtt{1dd43f03e6f7564c}_{16}$$
$$k^{2} = \mathtt{ebe26756de9937c7}_{16}$$
$$k^{3} = \mathtt{961704e945bad4fb}_{16}$$

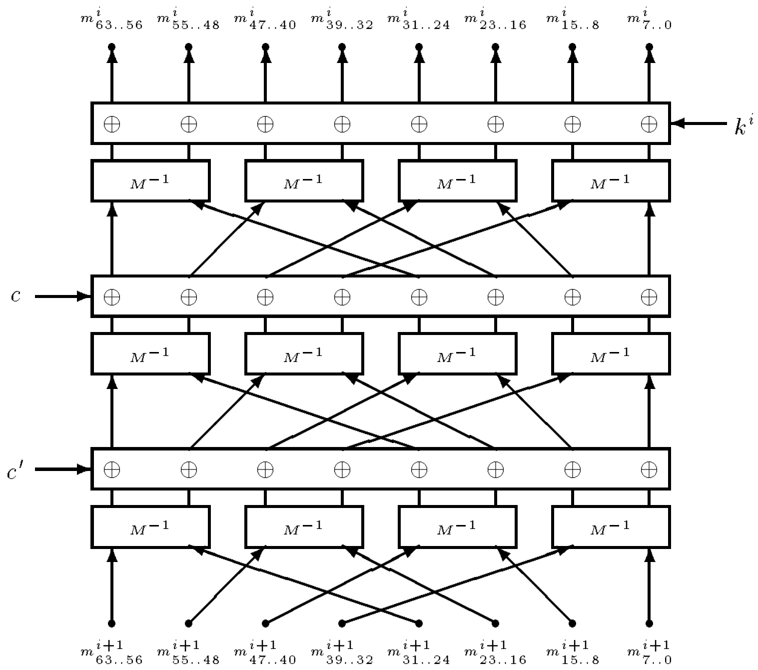

Fig. 6. One decryption round

$$k^4 = \texttt{0b60dfe9eff473d4}_{16}$$
$$k^5 = \texttt{76d3e7cf52c466cf}_{16}$$
$$k^6 = \texttt{75ec8cef767d3a0d}_{16}$$
$$k^7 = \texttt{82da3337b598fd6d}_{16}$$
$$k^8 = \texttt{fbd820da8dc8af8c}_{16}$$

For instance, the first generated subkey $k^0 = \texttt{45fd137a4edf9ec4}_{16}$ is

$$\begin{aligned} k^0 &= k^{-2} \oplus T(P^8(k^{-1} \oplus \texttt{290d61409ceb9e8f}_{16})) \\ &= k^{-2} \oplus T(\texttt{b711fa89ae0394e4}_{16}) \\ &= k^{-2} \oplus \texttt{bb21a9e2388bacd4}_{16}. \end{aligned}$$

The messages which enter into each round are

$$m^0 = \texttt{0123456789abcdef}_{16}$$
$$m^1 = \texttt{c3feb96c0cf4b649}_{16}$$
$$m^2 = \texttt{3f54e0c8e61a84d1}_{16}$$
$$m^3 = \texttt{b15cb4af3786976e}_{16}$$
$$m^4 = \texttt{76c122b7a562ac45}_{16}$$
$$m^5 = \texttt{21300b6ccfaa08d8}_{16}$$
$$m^6 = \texttt{99b8d8ab9034ec9a}_{16}$$
$$m^7 = \texttt{a2245ba3697445d2}_{16}$$

and the ciphertext is $\texttt{88fddfbe954479d7}_{16}$. In the first round, the message m^0 is transformed though three layers into m^1. The intermediate results between the layers are $\texttt{d85c19785690b0e3}_{16}$ and $\texttt{0f4bfb9e2f8ac7e2}_{16}$. For instance, in the first layer we take m^0, xor it with k^0, apply M, permute the bytes and get $\texttt{d85c19785690b0e3}_{16}$.

As an implementation test, we mention that if we iterate one million times the encryption on the all-zero bitstring with the previous key, we obtain the final ciphertext $\texttt{fd5c9c6889784b1c}_{16}$.

2 Design arguments

The Fast Fourier Transform used in the round-encryption function E has been used in several cryptographic designs including Schnorr's FFT-Hashing [22], FFT Hash II [23], Schnorr and Vaudenay's Parallel FFT-Hashing [24], and Massey's SAFER [13, 14]. This graph has been proved to have very good diffusion properties when done twice (see [25, 26, 30]).

The M structure implements a *multipermutation* as defined by Schnorr and Vaudenay (see [25, 29, 30]). In this case, it means that M is a permutation over the set of all 16-bit strings, and that fixing any of the two 8-bit inputs arbitrarily makes both 8-bit outputs be permutations of the other one. This is due to a very particular property of φ, namely that both φ and $x \mapsto \varphi(x) \oplus R_l(x)$ (which is in fact φ') are permutations. Actually, φ and φ' are linear involutions.

Those properties make E be what we call a *mixing function*, *i.e.* such that if we arbitrarily fix seven of the eight inputs, all outputs are permutation of the remaining free input. This performs a good diffusion.

The best general attack methods on block ciphers have been introduced by Gilbert, Chassé, Tardy-Corfdir, Biham, Shamir and Matsui (see [11, 28, 5, 6, 15, 16, 10]). They are now known as differential and linear cryptanalysis. We know study how CS-CIPHER has been protected against it.

The permutation P has been chosen to be an nonlinear involution in the sense that both differential and linear cryptanalysis are hard. Nonlinearity has one measure corresponding to differential cryptanalysis (which has been defined by Nyberg [19]) and one measure corresponding to linear cryptanalysis (which has been defined by Chabaud and Vaudenay [7]). Here we use the formalism introduced by Matsui [17]:

$$\mathrm{DP}_{\max}(f) = \max_{a \neq 0, b} \Pr_{X \text{ uniform}} [f(X \oplus a) \oplus f(X) = b]$$

$$\mathrm{LP}_{\max}(f) = \max_{a, b \neq 0} \left(2 \Pr_{X \text{ uniform}} [X \cdot a = f(X) \cdot b] - 1 \right)^2 .$$

The functions f and g are such that $\mathrm{DP}_{\max}(f) \leq 2^{-2}$ and $\mathrm{LP}_{\max}(f) \leq 2^{-2}$. If the Theorem of Aoki and Ohta [3] (which generalizes the Theorem of Nyberg and Knudsen [20]) were applicable in this setting, we would then obtain $\mathrm{DP}_{\max}(P) \leq 2^{-4}$ and $\mathrm{LP}_{\max}(P) \leq 2^{-4}$. Both properties are however still satisfied as the experiment shows. From [19, 7] it is known that for any function f on the set of all n-bit strings we have $\mathrm{DP}_{\max}(f) \geq 2^{1-n}$ and $\mathrm{LP}_{\max}(f) \geq 2^{1-n}$, but it is conjectured that 2^{2-n} is a better bound for even n (see Dobbertin [8] for instance). So our functions are reasonably nonlinear. Since it is well known that the heuristic complexity of differential or linear cryptanalysis is greater than the inverse of the product of the $\mathrm{DP}_{\max}$ or $\mathrm{LP}_{\max}$ of all active P boxes (see for instance Heys and Tavares [12]), having mixing functions makes at least five P box per round to be active, so no four rounds of CS-CIPHER have any efficient differential or linear characteristic.

3 Implementation

In any kind of implementation, the key scheduling scheme is assumed to be precomputed. This part of CS-CIPHER has not be designed to have special implementation optimization. The authors believe that every time one changes the secret key, one has to perform expensive computations (such as asymmetric cryptography, key exchange protocol or key transfer protocols) so optimizing the precomputation of the subkey sequence is meaningless. In the following Sections we only discuss implementation of the encryption (or decryption) scheme.

3.1 VLSI implementation

CS-CIPHER is highly optimized for VLSI implementations. It may be noticed that the g function has been designed to get a friendly boolean circuit implementation. Actually, Figure 7 illustrates a cheap nand-circuit with depth 4 and only 16 nand gates.

We propose two possible easy implementations. In the first one, we really implement one third of a single round encryption. It has two 64-bit input registers and one 64-bit output register. It is easy to see that an encryption can be performed by iterating this circuit 24 times and loading the subkey sequence $k^0, c, c', k^1, c, c', \ldots$ Straightforward estimates shows this circuit requires 1216 nand-gates with depth 26. This implementation can be added in any microprocessor within less than $1\mathrm{mm}^2$ in order to get a simple microcoded encryption instruction. One 30MHz-clock cycle is far enough to compute one layer, thus one 64-bit encryption requires 24 clock cycles, which leads to a 73Mbps, which is quite fast for such a cheap technology.

The second implementation consists in making a dedicated chip which consists of 24 times the previous circuit in a pipeline architecture. We estimate we need $15\mathrm{mm}^2$ in order to implement a 30000 nand-gate circuit which performs a 64-bit encryption within one 30MHz-clock cycle, which leads to an encryption rate of 2Gbps. This can be used to encrypt ATM network communications or PCI bus.

$$
\begin{array}{lllll}
\text{layer4}: & g_0 = g_4 n g_5 & g_1 = g_6 n g_7 & g_2 = g_8 n g_9 & g_3 = g_{10} n g_{11} \\
\text{layer3}: & g_5 = g_6 n g_{14} & g_8 = g_{17} n g_4 & g_9 = g_7 n g_{14} & g_{10} = g_6 n g_{16} \\
\text{layer2}: & g_6 = g_{14} n g_{12} & g_7 = g_{15} n g_{16} & g_{11} = g_{13} n g_4 & g_{17} = g_{15} n g_{19} \\
\text{layer1}: & g_4 = g_{12} n g_{13} & g_{14} = g_{18} n g_{18} & g_{15} = g_{18} n g_{12} & g_{19} = g_{16} n g_{13} \\
\text{layer0}: & g_{12} & g_{13} & g_{16} & g_{18}
\end{array}
$$

$$
\begin{array}{ll}
\text{Input}: & g_{16} g_{13} g_{18} g_{12} \\
\text{Output}: & g_0 g_1 g_2 g_3
\end{array}
$$

Fig. 7. Implementation of g

Those results can be compared to MISTY which has been implemented by Mitsubishi. In Matsui [18], this chip is specified to require 65000 gates, working at 14MHz and encrypting at 450Mbps.

3.2 Software implementation on modern microprocessors

A straightforward non-optimized implementation of CS-CIPHER in standard C on a Pentium 133MHz (see Appendix) gives an encryption rate of 2.1Mbps which is reasonably fast compared to similar implementations of DES.

Another (non-optimized) implementation in assembly code enables the Pentium to perform a 64-bit encryption within 973 clock cycles, which leads to 8.34Mbps working at 133MHz.

An evaluation similar to the VLSI-implementation estimates shows that the number of "usual" boolean gate (xor, and, or not) required to implement 64 parallel 64-bit encryptions using Biham's bit-slice trick on a 64-bit microprocessor is 11968, which is substantially less than Biham's implementation of DES which requires about 16000 instructions (see [4]). Therefore, if we use a 300MHz Alpha microprocessor which requires .5cycles per instructions (as in [4]), we obtain an encryption rate of about 196Mbps.

3.3 Software implementation on 8-bit microprocessors

An implementation has been done for a cheap smart card platform. A compact 6805 assembly code of roughly 500 bytes can encrypt a 64-bit string in its buffer RAM by using only 6 extra byte-registers within 12633 clock cycles. This means that a cheap smart card working at 4MHz can encrypt within 3,16ms (*i.e.* at a 19,8Kbps rate), which is better than optimized implementations of DES[1]. This implementation of CS-CIPHER can still be optimized.

platform	clock frequency	encryption rate	note
VLSI 1216nand 1mm^2	30MHz	73Mbps	estimate
VLSI 30000nand 15mm^2	30MHz	2Gbps	estimate
standard C 32bits	133MHz	2Mbps	see Appendix
bit slice (Pentium)	133MHz	11Mbps	estimate
bit slice (Alpha)	300MHz	196Mbps	estimate
Pentium assembly code	133MHz	8Mbps	non-optimized
6805 assembly code	4MHz	20Kbps	non-optimized

Fig. 8. Implementations of CS-CIPHER

4 Conclusion

CS-CIPHER has been shown to offer quite fast encryption rates on several kinds of platforms, which is suitable for telecommunication applications. Figure 8 summerizes the implementation results. Its security is based on heuristic arguments.

All attacks are welcome...

Acknowledgements

We wish to thank the COMPAGNIE DES SIGNAUX for having initiated and supported this work.

References

1. Data Encryption Standard. *Federal Information Processing Standard Publication 46*, U. S. National Bureau of Standards, 1977.
2. DES Modes of Operation. *Federal Information Processing Standard Publication 81*, U. S. National Bureau of Standards, 1980.
3. K. Aoki, K. Ohta. Strict evaluation of the maximum average of differential probability and the maximum average of linear probability. *IEICE Transactions on Fundamentals*, vol. E80-A, pp. 1–8, 1997.
4. E. Biham. A fast new DES implementation in software. In *Fast Software Encryption*, Haifa, Israel, Lectures Notes in Computer Science 1267, pp. 260–272, Springer-Verlag, 1997.
5. E. Biham, A. Shamir. Differential cryptanalysis of the full 16-round DES. In *Advances in Cryptology CRYPTO'92*, Santa Barbara, California, U.S.A., Lectures Notes in Computer Science 740, pp. 487–496, Springer-Verlag, 1993.
6. E. Biham, A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.
7. F. Chabaud, S. Vaudenay. Links between differential and linear cryptanalysis. In *Advances in Cryptology EUROCRYPT'94*, Perugia, Italy, Lectures Notes in Computer Science 950, pp. 356–365, Springer-Verlag, 1995.
8. H. Dobbertin. Almost Perfect nonlinear power functions on GF(2^n). *IEEE Trans. Inf. Theory*, submitted.
9. H. Feistel. Cryptography and computer privacy. *Scientific american*, vol. 228, pp. 15–23, 1973.
10. H. Gilbert. *Cryptanalyse Statistique des Algorithmes de Chiffrement et Sécurité des Schémas d'Authentification*, Thèse de Doctorat de l'Université de Paris 11, 1997.
11. H. Gilbert, G. Chassé. A statistical attack of the FEAL-8 cryptosystem. In *Advances in Cryptology CRYPTO'90*, Santa Barbara, California, U.S.A., Lectures Notes in Computer Science 537, pp. 22–33, Springer-Verlag, 1991.
12. H. M. Heys, S. E. Tavares. Substitution-Permutation Networks resistant to differential and linear cryptanalysis. *Journal of Cryptology*, vol. 9, pp. 1–19, 1996.
13. J. L. Massey. SAFER K-64: a byte-oriented block-ciphering algorithm. In *Fast Software Encryption*, Cambridge, United Kingdom, Lectures Notes in Computer Science 809, pp. 1–17, Springer-Verlag, 1994.
14. J. L. Massey. SAFER K–64: one year later. In *Fast Software Encryption*, Cambridge, United Kingdom, Lectures Notes in Computer Science 809, pp. 212–241, Springer-Verlag, 1994.
15. M. Matsui. Linear cryptanalysis methods for DES cipher. In *Advances in Cryptology EUROCRYPT'93*, Lofthus, Norway, Lectures Notes in Computer Science 765, pp. 386–397, Springer-Verlag, 1994.
16. M. Matsui. The first experimental cryptanalysis of the Data Encryption Standard. In *Advances in Cryptology CRYPTO'94*, Santa Barbara, California, U.S.A., Lectures Notes in Computer Science 839, pp. 1–11, Springer-Verlag, 1994.
17. M. Matsui. New structure of block ciphers with provable security against differential and linear cryptanalysis. In *Fast Software Encryption*, Cambridge, United Kingdom, Lectures Notes in Computer Science 1039, pp. 205–218, Springer-Verlag, 1996.
18. M. Matsui. New block encryption algorithm MISTY. In *Fast Software Encryption*, Haifa, Israel, Lectures Notes in Computer Science 1267, pp. 54–68, Springer-Verlag, 1997.

19. K. Nyberg. Perfect nonlinear S-boxes. In *Advances in Cryptology EUROCRYPT'91*, Brighton, United Kingdom, Lectures Notes in Computer Science 547, pp. 378–385, Springer-Verlag, 1991.
20. K. Nyberg, L. R. Knudsen. Provable security against a differential cryptanalysis. *Journal of Cryptology*, vol. 8, pp. 27–37, 1995.
21. Organisation for Economic Co-operation and Development *Cryptography Policy Guidelines*, 27 March, 1997.
22. C. P. Schnorr. FFT-Hashing: an efficient cryptographic hash function. Présenté à CRYPTO'91. Non publié.
23. C. P. Schnorr. FFT-Hash II, efficient cryptographic hashing. In *Advances in Cryptology EUROCRYPT'92*, Balatonfüred, Hungary, Lectures Notes in Computer Science 658, pp. 45–54, Springer-Verlag, 1993.
24. C.-P. Schnorr, S. Vaudenay. Parallel FFT-hashing. In *Fast Software Encryption*, Cambridge, United Kingdom, Lectures Notes in Computer Science 809, pp. 149–156, Springer-Verlag, 1994.
25. C.-P. Schnorr, S. Vaudenay. Black box cryptanalysis of hash networks based on multipermutations. In *Advances in Cryptology EUROCRYPT'94*, Perugia, Italy, Lectures Notes in Computer Science 950, pp. 47–57, Springer-Verlag, 1995.
26. C. P. Schnorr, S. Vaudenay. Black box cryptanalysis of cryptographic primitives. Submitted. Early version available as LIENS Report 95–28, Laboratoire d'Informatique de l'Ecole Normale Supérieure, 1995. `ftp://ftp.ens.fr/pub/reports/liens/liens-95-28.A4.ps.Z`
27. C. E. Shannon. Communication theory of secrecy systems. *Bell system technical journal*, vol. 28, pp. 656–715, 1949.
28. A. Tardy-Corfdir, H. Gilbert. A known plaintext attack of FEAL-4 and FEAL-6. In *Advances in Cryptology CRYPTO'91*, Santa Barbara, California, U.S.A., Lectures Notes in Computer Science 576, pp. 172–181, Springer-Verlag, 1992.
29. S. Vaudenay. On the need for multipermutations: cryptanalysis of MD4 and SAFER. In *Fast Software Encryption*, Leuven, Belgium, Lectures Notes in Computer Science 1008, pp. 286–297, Springer-Verlag, 1995.
30. S. Vaudenay. *La Sécurité des Primitives Cryptographiques*, Thèse de Doctorat de l'Université de Paris 7, Technical Report LIENS-95-10 of the Laboratoire d'Informatique de l'Ecole Normale Supérieure, 1995.
31. Rocke Verser. Strong cryptography makes the world a safer place. `http://www.frii.com/~rcv/deschall.htm`

Appendix

Here is a sample implementation of the heart of CS-Cipher. The procedure takes plaintext block `m` and a precomputed subkey sequence `k` (as a 9×8 bytes array). This program is highly optimizable.

```
typedef unsigned char uint8;
#define CSC_C00 0xb7
#define CSC_C01 0xe1
#define CSC_C02 0x51
#define CSC_C03 0x62
#define CSC_C04 0x8a
#define CSC_C05 0xed
#define CSC_C06 0x2a
#define CSC_C07 0x6a
```

```
#define CSC_C10 0xbf
#define CSC_C11 0x71
#define CSC_C12 0x58
#define CSC_C13 0x80
#define CSC_C14 0x9c
#define CSC_C15 0xf4
#define CSC_C16 0xf3
#define CSC_C17 0xc7
uint8 tbp[256]={
  0x29,0x0d,0x61,0x40,0x9c,0xeb,0x9e,0x8f,
  0x1f,0x85,0x5f,0x58,0x5b,0x01,0x39,0x86,
  0x97,0x2e,0xd7,0xd6,0x35,0xae,0x17,0x16,
  0x21,0xb6,0x69,0x4e,0xa5,0x72,0x87,0x08,
  0x3c,0x18,0xe6,0xe7,0xfa,0xad,0xb8,0x89,
  0xb7,0x00,0xf7,0x6f,0x73,0x84,0x11,0x63,
  0x3f,0x96,0x7f,0x6e,0xbf,0x14,0x9d,0xac,
  0xa4,0x0e,0x7e,0xf6,0x20,0x4a,0x62,0x30,
  0x03,0xc5,0x4b,0x5a,0x46,0xa3,0x44,0x65,
  0x7d,0x4d,0x3d,0x42,0x79,0x49,0x1b,0x5c,
  0xf5,0x6c,0xb5,0x94,0x54,0xff,0x56,0x57,
  0x0b,0xf4,0x43,0x0c,0x4f,0x70,0x6d,0x0a,
  0xe4,0x02,0x3e,0x2f,0xa2,0x47,0xe0,0xc1,
  0xd5,0x1a,0x95,0xa7,0x51,0x5e,0x33,0x2b,
  0x5d,0xd4,0x1d,0x2c,0xee,0x75,0xec,0xdd,
  0x7c,0x4c,0xa6,0xb4,0x78,0x48,0x3a,0x32,
  0x98,0xaf,0xc0,0xe1,0x2d,0x09,0x0f,0x1e,
  0xb9,0x27,0x8a,0xe9,0xbd,0xe3,0x9f,0x07,
  0xb1,0xea,0x92,0x93,0x53,0x6a,0x31,0x10,
  0x80,0xf2,0xd8,0x9b,0x04,0x36,0x06,0x8e,
  0xbe,0xa9,0x64,0x45,0x38,0x1c,0x7a,0x6b,
  0xf3,0xa1,0xf0,0xcd,0x37,0x25,0x15,0x81,
  0xfb,0x90,0xe8,0xd9,0x7b,0x52,0x19,0x28,
  0x26,0x88,0xfc,0xd1,0xe2,0x8c,0xa0,0x34,
  0x82,0x67,0xda,0xcb,0xc7,0x41,0xe5,0xc4,
  0xc8,0xef,0xdb,0xc3,0xcc,0xab,0xce,0xed,
  0xd0,0xbb,0xd3,0xd2,0x71,0x68,0x13,0x12,
  0x9a,0xb3,0xc2,0xca,0xde,0x77,0xdc,0xdf,
  0x66,0x83,0xbc,0x8d,0x60,0xc6,0x22,0x23,
  0xb2,0x8b,0x91,0x05,0x76,0xcf,0x74,0xc9,
  0xaa,0xf1,0x99,0xa8,0x59,0x50,0x3b,0x2a,
  0xfe,0xf9,0x24,0xb0,0xba,0xfd,0xf8,0x55,
};
void enc_csc(uint8 m[8],uint8* k) {
  uint8 tmpx,tmprx,tmpy;
  int i;
  #define APPLY_M(cl,cr,adl,adr) \
    tmpx=m[adl]^cl; \
    tmprx=(tmpx<<1)^(tmpx>>7); \
    tmpy=m[adr]^cr; \
    m[adl]=tbp[(tmprx&0x55)^tmpx^tmpy]; \
    m[adr]=tbp[tmprx^tmpy];
  for(i=0;i<8;i++,k+=8) {
    APPLY_M(k[0],k[1],0,1)
    APPLY_M(k[2],k[3],2,3)
    APPLY_M(k[4],k[5],4,5)
    APPLY_M(k[6],k[7],6,7)
    APPLY_M(CSC_C00,CSC_C01,0,2)
    APPLY_M(CSC_C02,CSC_C03,4,6)
    APPLY_M(CSC_C04,CSC_C05,1,3)
    APPLY_M(CSC_C06,CSC_C07,5,7)
    APPLY_M(CSC_C10,CSC_C11,0,4)
    APPLY_M(CSC_C12,CSC_C13,1,5)
    APPLY_M(CSC_C14,CSC_C15,2,6)
    APPLY_M(CSC_C16,CSC_C17,3,7)
  }
  for(i=0;i<8;i++) m[i]^=k[i];
}
```