Decorrelated Fast Cipher: an AES Candidate

Henri Gilbert[2], Marc Girault[2], Philippe Hoogvorst[1], Fabrice Noilhan[1], Thomas Pornin[1], Guillaume Poupard[1], Jacques Stern[1], and Serge Vaudenay[1]

[1] Ecole Normale Supérieure – CNRS
[2] France Telecom

Contact e-mail: `Serge.Vaudenay@ens.fr`

Version: 1998, May 19th

Abstract. This report presents a response to the call for candidates issued by the National Institute for Standards and Technologies (the Advanced Encryption Standard project). The proposed candidate — called DFC as for "Decorrelated Fast Cipher" — is based on Vaudenay's decorrelation technique. This provides provable security against several classes of attacks which include the basic version of Biham and Shamir's Differential Cryptanalysis as well as Matsui's Linear Cryptanalysis.

Since the beginning of commercial use of symmetric encryption through block ciphers in the seventies, construction design used to be heuristic-based and security was empiric: a given block cipher was considered to be secure until some researcher published an attack on.

The Data Encryption Standard [1] initiated an important open research area, and some important cryptanalysis methods emerged, namely Biham and Shamir's differential cryptanalysis [7] and Matsui's linear cryptanalysis [13], as well as further generalizations. Nyberg and Knudsen [16] showed how to build toy block ciphers which provably resist differential cryptanalysis (and linear cryptanalysis as well as has been shown afterward [4]). This paradigm has successfully been used by Matsui in the MISTY cipher [14, 15]. However Nyberg and Knudsen's method does not provide much freedom for the design, and actually, this paradigm leads to algebraic constructions. This may open the way to other kind of weaknesses as shown by Jakobsen and Knudsen [9] (although no weakness has been discovered in MISTY so far).

Here, we propose a new design which combines heuristic construction with provable security against a wide class of attacks. Unlike the Nyberg-Knudsen paradigm, our approach is combinatorial. It relies on Vaudenay's paradigm [19–21]. This construction provides much more freedom since it can be combined with heuristic designs.

In response to the call for candidate for the Advanced Encryption Standard (AES) which has been issued by the National Institute of Standards and Technology (NIST) we propose the hereafter defined *Decorrelated Fast Cipher* (DFC). It is a block cipher which supports message blocks of length 128 and key string of arbitrary length up to 256. We provide the tests (Known Answer Tests and Monte-Carlo Tests) as requested by NIST. We provide portable implementations in ANSI-C and JAVA™ languages. We also provide an optimized C program (which is a regular ANSI-C with the nowadays usual `long long int` type extension). All these support the Application Program Interface (API) provided by the NIST. In addition we provide implementations in assembly languages for Pentium™, SPARC™, AXP™ and Motorola™ 6805 (which is commonly used in smart cards). We also provide some security analysis.

1 Definition of DFC

1.1 Notations and Other Conventions

All objects are bit strings or integers. The notations used to manipulate them are as follows.

$\mathtt{d43}_{\mathrm{x}}$ We represent bit strings in hexadecimal by packing bits into nibbles. For instance, $\mathtt{d43}_{\mathrm{x}}$ denotes the bit string `110101000011`.

$\bar{s}$ and $|x|_\ell$ To any bit string $s = b_1 \ldots b_\ell$ we associate an integer

$$\bar{s} = 2^{\ell-1} b_1 + \ldots + 2b_{\ell-1} + b_\ell \qquad (1)$$

(the leftmost bit is thus the most significant bit). Integers are denoted in standard decimal notation. The converse operation of representing an integer x as an ℓ-bit string is denoted $|x|_\ell$. For instance, $\overline{\mathtt{d43}_{\mathrm{x}}} = 3395$ and $|3395|_{12} = \mathtt{d43}_{\mathrm{x}}$.

$s\|s'$	The concatenation of two strings s and s' is denoted $s\|s'$.
$\mathrm{trunc}_n(s)$	We can truncate a bit string $s = b_1 \ldots b_\ell$ (of length at least n) to its n leftmost bits $\mathrm{trunc}_n(s) = b_1 \ldots b_n$.
$s \oplus s'$, $s \wedge s'$	The bitwise *exclusive or* of two bitstrings of equal length s and s' is denoted $s \oplus s'$. Their bitwise *and* is denoted $s \wedge s'$.
$\neg s$	The bitwise negation of one bitstring s is denoted $\neg s$.
$+, \times, \bmod$	The arithmetical operations $+$, $\times$, $\bmod$ are the natural operations over the integers.

1.2 High Level Overview

The encryption function DFC_K operates on 128-bit message blocks by means of a secret key K of arbitrary length, up to 256 bits. The corresponding decryption function is DFC_K^{-1} and operates on 128-bit message blocks. Encryption of arbitrary-length messages is performed through standard modes of operation which are independent of the DFC design (see [2]).

The secret key K is first turned into a 1024-bit "Expanded Key" EK through an "Expanding Function" EF, *i.e.* $\mathrm{EK} = \mathrm{EF}(K)$. As explained in Section 1.5, the EF function performs a 4-round Feistel scheme (see Feistel [8]). The encryption itself performs a similar 8-round Feistel scheme. Each round uses the "Round Function" RF. This function maps a 64-bit string onto a 64-bit string by using one 128-bit string parameters. It is defined in Section 1.3.

Given a 128-bit plaintext block PT, we split it into two 64-bit halves R_0 and R_1 so that $\mathrm{PT} = R_0|R_1$. Given the 1024-bit expanded key EK, we split it into eight 128-bit strings

$$\mathrm{EK} = \mathrm{RK}_1|\mathrm{RK}_2|\ldots|\mathrm{RK}_8 \tag{2}$$

where RK_i is the ith "Round Key".

We build a sequence $R_0, \ldots, R_9$ by the Equation

$$R_{i+1} = \mathrm{RF}_{\mathrm{RK}_i}(R_i) \oplus R_{i-1}. \quad (i = 1, \ldots, 8) \tag{3}$$

We then set $\mathrm{CT} = \mathrm{DFC}_K(\mathrm{PT}) = R_9|R_8$ (see Fig. 1).

More generally, given a bitstring s of length multiple of 128, say $128r$, we can split it split into r 128-bit strings

$$s = p_1|p_2|\ldots|p_r.$$

From s we define a permutation Enc_s on the set of 128-bit strings which comes from an r-round Feistel scheme. For any 128-bit string m which is split into two 64-bit halves x_0 and x_1 so that $m = x_0|x_1$. We build a sequence $x_0, \ldots, x_{r+1}$ by the Equation

$$x_{i+1} = \mathrm{RF}_{p_i}(x_i) \oplus x_{i-1} \quad (i = 1, \ldots, r) \tag{4}$$

and we define $\mathrm{Enc}_s(m) = x_{r+1}|x_r$. The DFC_K encryption function is thus obtained as

$$\mathrm{DFC}_K = \mathrm{Enc}_{\mathrm{EF}(K)} \tag{5}$$

(hence an 8-round Feistel Cipher). The EF function uses a 4-round version defined with Enc.

Obviously, we have $\mathrm{DFC}_K^{-1} = \mathrm{Enc}_{\mathrm{revEK}}$ where

$$\mathrm{revEK} = \mathrm{RK}_8|\mathrm{RK}_7|\ldots|\mathrm{RK}_1. \tag{6}$$

1.3 The RF Function

The RF function (as for "Round Function") is fed with one 128-bit parameter, or equivalently two 64-bit parameters: an "a-parameter" and a "b-parameter". It processes a 64-bit input x and outputs a 64-bit string. We define

$$\mathrm{RF}_{a|b}(x) = \mathrm{CP}\left(\left|\left(\left(\bar{a} \times \bar{x} + \bar{b}\right) \bmod (2^{64} + 13)\right) \bmod 2^{64}\right|_{64}\right) \tag{7}$$

where CP is a permutation over the set of all 64-bit strings (which appears in Section 1.4).

1.4 The CP Permutation

The CP permutation (as for "Confusion Permutation") uses a look-up table RT (as for "Round Table") which takes a 6-bit integer as input and provides a 32-bit string output.

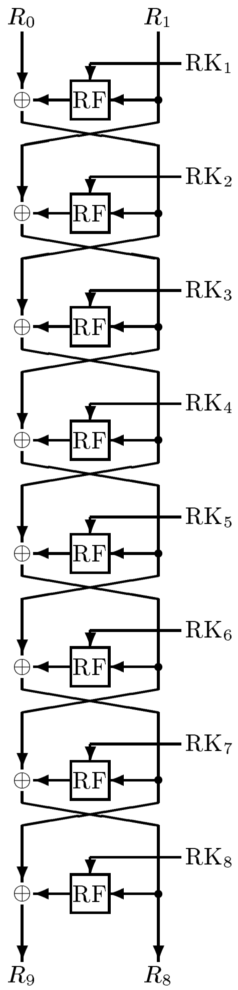

Fig. 1. An 8-Round Feistel Cipher.

Let $y = y_l | y_r$ be the input of CP where y_l and y_r are two 32-bit strings. We define

$$\mathrm{CP}(y) = \left| \overline{(y_r \oplus \mathrm{RT}(\overline{\mathrm{trunc}_6(y_l)})) | (y_l \oplus \mathrm{KC})} + \overline{\mathrm{KD}} \bmod 2^{64} \right|_{64} \tag{8}$$

where KC is a 32-bit constant string, and KD is a 64-bit constant string. Permutation CP is depicted on Fig. 2.

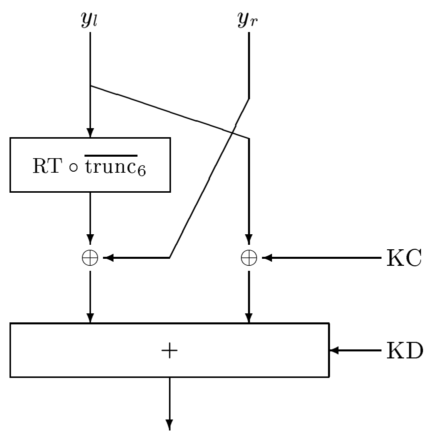

Fig. 2. The CP Permutation.

The constants $\mathrm{RT}(0), \ldots, \mathrm{RT}(63)$, KC and KD will be set in Section 1.6.

1.5 Key Scheduling Algorithm

In order to generate a sequence $\mathrm{RK}_1, \mathrm{RK}_2, \ldots, \mathrm{RK}_8$ from a given key K represented as a bit string of length at most 256, we use the following algorithm. We first pad K with a constant pattern KS in order to make a 256-bit "Padded Key" string by

$$\mathrm{PK} = \mathrm{trunc}_{256}(K|\mathrm{KS}). \tag{9}$$

If K is of length 128, we can observe that only the first 128 bits of KS are used. We define KS of length 256 in order to allow any key size from 0 to 256.

Then we cut PK into eight 32-bit strings $\mathrm{PK}_1, \ldots, \mathrm{PK}_8$ such that $\mathrm{PK} = \mathrm{PK}_1|\ldots|\mathrm{PK}_8$. We define

$$\mathrm{OAP}_1 = \mathrm{PK}_1|\mathrm{PK}_8 \tag{10}$$
$$\mathrm{OBP}_1 = \mathrm{PK}_5|\mathrm{PK}_4 \tag{11}$$
$$\mathrm{EAP}_1 = \mathrm{PK}_2|\mathrm{PK}_7 \tag{12}$$
$$\mathrm{EBP}_1 = \mathrm{PK}_6|\mathrm{PK}_3. \tag{13}$$

We also define

$$\mathrm{OAP}_i = \mathrm{OAP}_1 \oplus \mathrm{KA}_i \tag{14}$$
$$\mathrm{OBP}_i = \mathrm{OBP}_1 \oplus \mathrm{KB}_i \tag{15}$$
$$\mathrm{EAP}_i = \mathrm{EAP}_1 \oplus \mathrm{KA}_i \tag{16}$$
$$\mathrm{EBP}_i = \mathrm{EBP}_1 \oplus \mathrm{KB}_i \tag{17}$$

for $i = 2, 3, 4$ (where KA_i and KB_i are fixed constants defined in Section 1.6). The names of the variables come from "Odd a-Parameter", "Odd b-Parameter", "Even a-Parameter", and "Even b-Parameter" respectively, which will become clearer below.

We define

$$\mathrm{EF}_1(K) = \mathrm{OAP}_1|\mathrm{OBP}_1|\ldots|\mathrm{OAP}_4|\mathrm{OBP}_4. \tag{18}$$

It defines a four-round permutation which is $\mathrm{Enc}_{\mathrm{EF}_1(K)}$. Similarly,

$$\mathrm{EF}_2(K) = \mathrm{EAP}_1|\mathrm{EBP}_1|\ldots|\mathrm{EAP}_4|\mathrm{EBP}_4 \tag{19}$$

defines a four-round encryption function $\mathrm{Enc}_{\mathrm{EF}_2(K)}$.

The $\mathrm{Enc}_{\mathrm{EF}_1(K)}$ and $\mathrm{Enc}_{\mathrm{EF}_2(K)}$ enables to define the RK sequence. Namely, we let $\mathrm{RK}_0 = |0|_{128}$ and

$$\mathrm{RK}_i = \begin{cases} \mathrm{Enc}_{\mathrm{EF}_1(K)}(\mathrm{RK}_{i-1}) \text{ if } i \text{ is odd} \\ \mathrm{Enc}_{\mathrm{EF}_2(K)}(\mathrm{RK}_{i-1}) \text{ if } i \text{ is even.} \end{cases} \tag{20}$$

More precisely, we let

$$\mathrm{RK}_{i-1} = \mathrm{RV}_{i,0}|\mathrm{RV}_{i,1} \tag{21}$$

and we define

$$\mathrm{RV}_{i,j+1} = \begin{cases} \mathrm{RF}_{\mathrm{OAP}_j|\mathrm{OBP}_j}(\mathrm{RV}_{i,j}) \oplus \mathrm{RV}_{i,j-1} \text{ if } i \text{ is odd} \\ \mathrm{RF}_{\mathrm{EAP}_j|\mathrm{EBP}_j}(\mathrm{RV}_{i,j}) \oplus \mathrm{RV}_{i,j-1} \text{ if } i \text{ is even} \end{cases} \tag{22}$$

so that

$$\mathrm{RK}_i = \mathrm{RV}_{i,5}|\mathrm{RV}_{i,4}. \tag{23}$$

Finally we have

$$\mathrm{EF}(K) = \mathrm{RK}_1|\mathrm{RK}_2|\ldots|\mathrm{RK}_8. \tag{24}$$

Practically, we

1. compute $\mathrm{OAP}_i, \mathrm{OBP}_i, \mathrm{EAP}_i, \mathrm{EBP}_i$ from Equations (10)–(17);
2. set $\mathrm{RK}_0 = \mathrm{RV}_{1,0}|\mathrm{RV}_{1,1} = |0|_{128}$;
3. compute the $\mathrm{RV}_{1,j}$ values from Equation (22);
4. set $\mathrm{RK}_1 = \mathrm{RV}_{1,5}|\mathrm{RV}_{1,4} = \mathrm{RV}_{2,0}|\mathrm{RV}_{2,1}$;
5. compute the $\mathrm{RV}_{2,j}$ values from Equation (22);
6. ...
7. set $\mathrm{RK}_8 = \mathrm{RV}_{8,5}|\mathrm{RV}_{8,4}$.

1.6 On Defining the Constants

The previously defined algorithm depends on several constants:

- 64 constants $\mathrm{RT}(|0|_6), \ldots, \mathrm{RT}(|63|_6)$ of 32 bits,
- one 64-bit constant KD,
- one 32-bit constant KC,
- three 64-bit constants $\mathrm{KA}_2, \mathrm{KA}_3, \mathrm{KA}_4$,
- three 64-bit constants $\mathrm{KB}_2, \mathrm{KB}_3, \mathrm{KB}_4$,
- one 256-bit constant KS.

Security arguments require that these constants fulfill the following criteria.

1. No collision occurs on table RT, *i.e.* we have $\mathrm{RT}(s) = \mathrm{RT}(s')$ for no $s \neq s'$.
2. KD is odd.

In order to convince that this design hides no trap-door, we choose the constants from the hexadecimal expansion of the mathematical e constant

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 2.\mathtt{b7e151628aed2a6abf7158}_{\mathrm{x}} \ldots \tag{25}$$

If EES is the "e Expansion String" of the first 2144 bits of this expansion (after the decimal point), we define

$$\mathrm{EES} = \mathrm{RT}(0)|\mathrm{RT}(1)|\ldots|\mathrm{RT}(63)|\mathrm{KC}|\mathrm{KD}. \tag{26}$$

In addition we define

$$\mathrm{trunc}_{640}(\mathrm{EES}) = \mathrm{KA}_2|\mathrm{KA}_3|\mathrm{KA}_4|\mathrm{KB}_2|\mathrm{KB}_3|\mathrm{KB}_4|\mathrm{KS}. \tag{27}$$

Thus the table RT is given in rows by

$\mathtt{b7e15162}_x$	$\mathtt{8aed2a6a}_x$	$\mathtt{bf715880}_x$	$\mathtt{9cf4f3c7}_x$
$\mathtt{62e7160f}_x$	$\mathtt{38b4da56}_x$	$\mathtt{a784d904}_x$	$\mathtt{5190cfef}_x$
$\mathtt{324e7738}_x$	$\mathtt{926cfbe5}_x$	$\mathtt{f4bf8d8d}_x$	$\mathtt{8c31d763}_x$
$\mathtt{da06c80a}_x$	$\mathtt{bb1185eb}_x$	$\mathtt{4f7c7b57}_x$	$\mathtt{57f59584}_x$
$\mathtt{90cfd47d}_x$	$\mathtt{7c19bb42}_x$	$\mathtt{158d9554}_x$	$\mathtt{f7b46bce}_x$
$\mathtt{d55c4d79}_x$	$\mathtt{fd5f24d6}_x$	$\mathtt{613c31c3}_x$	$\mathtt{839a2ddf}_x$
$\mathtt{8a9a276b}_x$	$\mathtt{cfbfa1c8}_x$	$\mathtt{77c56284}_x$	$\mathtt{dab79cd4}_x$
$\mathtt{c2b3293d}_x$	$\mathtt{20e9e5ea}_x$	$\mathtt{f02ac60a}_x$	$\mathtt{cc93ed87}_x$
$\mathtt{4422a52e}_x$	$\mathtt{cb238fee}_x$	$\mathtt{e5ab6add}_x$	$\mathtt{835fd1a0}_x$
$\mathtt{753d0a8f}_x$	$\mathtt{78e537d2}_x$	$\mathtt{b95bb79d}_x$	$\mathtt{8dcaec64}_x$
$\mathtt{2c1e9f23}_x$	$\mathtt{b829b5c2}_x$	$\mathtt{780bf387}_x$	$\mathtt{37df8bb3}_x$
$\mathtt{00d01334}_x$	$\mathtt{a0d0bd86}_x$	$\mathtt{45cbfa73}_x$	$\mathtt{a6160ffe}_x$
$\mathtt{393c48cb}_x$	$\mathtt{bbca060f}_x$	$\mathtt{0ff8ec6d}_x$	$\mathtt{31beb5cc}_x$
$\mathtt{eed7f2f0}_x$	$\mathtt{bb088017}_x$	$\mathtt{163bc60d}_x$	$\mathtt{f45a0ecb}_x$
$\mathtt{1bcd289b}_x$	$\mathtt{06cbbfea}_x$	$\mathtt{21ad08e1}_x$	$\mathtt{847f3f73}_x$
$\mathtt{78d56ced}_x$	$\mathtt{94640d6e}_x$	$\mathtt{f0d3d37b}_x$	$\mathtt{e67008e1}_x$

and we also have

$$\begin{aligned}
\mathrm{KD} &= \mathtt{86d1bf275b9b241d}_x\\
\mathrm{KC} &= \mathtt{eb64749a}_x\\
\mathrm{KA}_2 &= \mathtt{b7e151628aed2a6a}_x\\
\mathrm{KA}_3 &= \mathtt{bf7158809cf4f3c7}_x\\
\mathrm{KA}_4 &= \mathtt{62e7160f38b4da56}_x\\
\mathrm{KB}_2 &= \mathtt{a784d9045190cfef}_x\\
\mathrm{KB}_3 &= \mathtt{324e7738926cfbe5}_x\\
\mathrm{KB}_4 &= \mathtt{f4bf8d8d8c31d763}_x\\
\mathrm{KS} &= \mathtt{da06c80abb1185eb4f7c7b5757f59584}_x|\\
&\quad\ \mathtt{90cfd47d7c19bb42158d9554f7b46bce}_x.
\end{aligned}$$

We further check that the previous criteria are satisfied.

2 Tests

In addition to the Known Answer Tests (KAT) requested by the NIST, we have included another KAT for the generation of internal key scheduling algorithm. The additional fields of the new KAT are as follows.

`PK`: padded 256-bit key PK, Equation (9)
`OAP`: first odd a-parameter OAP_1, Equation (10)
`OBP`: first odd b-parameter OBP_1, Equation (11)
`EAP`: first even a-parameter EAP_1, Equation (12)
`EBP`: first even b-parameter EBP_1, Equation (13)
`RV`ij: round value $\mathrm{RV}_{i,j}$, Equation (22)
`RK`i: round key RK_i, Equation (23)
`PT`: plaintext PT
`R`i: round value R_i, Equation (3)
`CT`: ciphertext CT

Due to the similarity between the encryption and the decryption, we do not include such a "variable ciphertext KAT".

3 Implementation

3.1 Multiprecision Arithmetic

The internal operations deal with 64- and 128-bit numbers. Few microprocessors can compute directly with such quantities. Generally we shall have to implement multiprecision arithmetic. Such operation is usually best implemented using assembly code. DFC is not an exception: we will see that the performance of the `ANSI-C` implementation is much lower than the performance of the assembly-code implementations.

The key operations are an affine mapping $x \mapsto P = ax + b$ where a, b and x are 64-bit operands and P a 128-bit quantity, followed by two reductions modulo $2^{64} + 13$, and modulo 2^{64} respectively.

The implementation of the multiprecision multiplication follows a classical scheme, without any optimizations, such as Karatsuba's, which would not be worthwhile for so small operands.

The modular reduction must not use division, which is too slow. Instead, we use the following method. First we write

$$P = Q2^{64} + R \tag{28}$$

where R is the remainder of the Euclidean division of P by 2^{64}. (No computation is required for this.) Then we rewrite (28) as:

$$P = Q(2^{64} + 13) + R - 13Q \tag{29}$$

As we want to reduce modulo $2^{64} + 13$, the quantity $Q(2^{64} + 13)$ disappears. However, this is not yet perfect as we have to deal with two cases: $R - 13Q > 0$ and $R - 13Q < 0$. To avoid this, we rewrite (29) as:

$$P = (2^{64} + 13)(Q - 13) + (13(2^{64} - 1 - Q) + 182 + R) \tag{30}$$

In (30), the quantity $2^{64} - 1 - Q$ is the bitwise complement of Q. The modular reduction thus consists of the evaluation of:

$$P' = 13(2^{64} - 1 - Q) + 182 + R \tag{31}$$

modulo $2^{64} + 13$. The result is always positive and, most of the time, greater than $2^{64} + 13$. We thus perform a second reduction, using Formula (29). Let us write it

$$P' = Q'2^{64} + R'.$$

As we now have $Q' \leq 13$, we can store the values of

$$13(2^{64} - 1 - Q') + 182 \bmod (2^{64} + 13)$$

in a table. Further optimizations are dedicated to the architecture.

The computation of the CP function is straightforward and requires no particular comment.

3.2 Application Program Interface

Five versions of the cipher have been implemented using the standard C Application Program Interface (API) provided by the NIST.

The `ANSI-C`, extended-C, AXP^{TM}, SPARC^{TM} and i386 assembly coded implementations can be used through the API. The API uses conditional defines in order to know which version to call.

Although it is possible to use the API to test these implementations (see for instance the `dfc_*_test` programs), benchmarking should not be used with this API since there are many costly conversions between native objects of the implementation and `BYTE` objects of the API. Hence, programs `dfc_*_bench` are provided in order to allow direct calls to the functions of the implementations. The performance tests have been made using this program together with the `time` program of UNIX^{TM} to encrypt 1048576 times a plaintext. We recall that the block-size is 128 bits, and the benchmarks do not depend on the length of the key.

3.3 `ANSI-C` Implementation

In `ANSI-C`, the largest guaranteed integer type is the `unsigned long int`, which may contain 32-bits values. We therefore use only $16 \times 16 \to 32$ multiplications. We represent big integers with arrays of 16-bits integers stored in `unsigned long int`. The code itself is heavily commented about these details.

This implementation should compile with any `ANSI-C` compiler, even on exotic architectures where signed integers are not stored in the two's complement binary representation. It has been tested on various UNIX^{TM} platforms, using whatever `ANSI-C` compiler available. It also works on Windows NT^{TM} using Visual C++^{TM} 4.0.

The `ANSI-C` implementation has been optimized for speed, without sacrificing code readability, so it may be used in production environment. We outline that this implementation corresponds to both the "Reference Implementation" and the "Mathematically Optimized C Implementation" required by NIST for the AES process.

3.4 Extended C Implementation

We provide an additional `C` implementation which is "almost" an `ANSI-C` implementation in the sense that it uses the extension of a 64-bits integer type, which is fairly common among recent C compilers and will become a standard soon. There are two versions of the main calculation (the multiplication and modular reduction): the first one is optimized for native 64-bits architectures (such as the AXP$^{\mathrm{TM}}$), whereas the second one includes some explicit casts to help the compiler bypass useless operations on architectures that may efficiently perform $32 \times 32 \rightarrow 64$ multiplications. On RISC architectures, this implementation is quite efficient (hand-coded assembly is only 30 or 40% faster).

3.5 JAVA$^{\mathrm{TM}}$ Implementation

Implementation of DFC in JAVA$^{\mathrm{TM}}$ uses the new BigInteger library from the JAVA$^{\mathrm{TM}}$ Development Kit (JDK) v 1.1 specifications. Since JAVA$^{\mathrm{TM}}$ does not have any unsigned integer type, the largest unsigned available integers have only 63 bits. Thus, without using the library, 64-bit multiplications would entail many steps and modular reductions would also be hard. The new BigInteger library can do multiplications, additions and modular reductions of integers of arbitrary length; this library allows us to do 64 bits multiplications faster.

DFC is really easy to implement in JAVA$^{\mathrm{TM}}$. The whole job is made using BigIntegers. Since conversions between BigIntegers and arrays of bytes are quite expensive, it is not worth trying to do simpler operations (for example XOR) on arrays of bytes.

Speed of encryption does pay its tribute to the BigInteger library. Even though these operations on BigIntegers take much time, it is still worth using them. Next JAVA$^{\mathrm{TM}}$ Virtual Machines and JAVA$^{\mathrm{TM}}$ processors should have a significant speedup on those new operations so that encryption may be faster in the future. At the moment, one can achieve a disappointing 4KB/s on a Ultrasparc$^{\mathrm{TM}}$ 166MHz running the JDK v 1.2 beta2. This is the consequence of the 64 bit unsigned integer design of DFC, which does not fit JAVA$^{\mathrm{TM}}$ specifications on integers.

3.6 Assembly Code Implementations

Pentium$^{\mathrm{TM}}$. The difficult points of this implementation are:

– the small number of registers,

– the ability of the CPU to execute two instructions together, if some conditions are met.
– the cost of mispredicted branches.

The lack of registers is a problem that cannot be solved: we have to cope with it. Our implementation computes the round function entirely in the CPU's registers. Only the constants are kept in memory. This makes it interesting to buffer as many plaintexts as possible before calling the encryption function: if sufficiently many encryptions are done in a row, the code will be in the program cache and the constants will all be in the data cache, thus speeding up the encryption process by a factor greater than 20.

The Pentium™ features two execution units: the "U" pipe and the "V" pipe. Only simple instructions can be executed in the "V" pipe. The V-pipe instruction must also not use the result of the U-pipe instruction. The condition for "pairing" instructions are detailed in the Intel™ documentation. Our implementation of DFC contains very few "pairing stalls".

The Pentium™ has prediction rules for branches. It allows it to pre-decode the instructions. Its prediction rules are relatively efficient but, in case of misprediction, the penalty can be high in terms of wasted cycles. In our implementation, we have solved this problem by eliminating all conditional branches.

SPARC™. The implementation on SPARC™ is easier due to the large number of 32-bit registers: we can freely use 24 of them. It is interesting to keep the scalar constants (KC, KD and the address of RT) in registers. This also implies that an optimal performance is obtained by buffering the data to be processed.

AXP™. The AXP™ architecture has unique properties that make implementation of DFC especially efficient. First of all, it is a native 64 bits processor, and the $64 \times 64 \rightarrow 128$ multiplication may be implemented as two successive opcodes, each one yielding 64 bits of the result.

The modular reduction uses Equation (30).

There are three generations in the AXP™ family. The first one is the 21064, that runs up to 300MHz and initiates 2 instructions per cycle, as long as there is no conflict in registers or memory usage.

A $64 \times 64 \to 64$ multiplication locks the multiplier for 21 cycles, and the result may be used after 23 cycles. Two such multiplications are needed at the beginning of each round of DFC, but we do not implement the multiplication by 13 with the internal multiplier: it is time-effective to perform it "by hand" with shifts and adds, as this may be done in parallel to the calculus of R (the low 64 bits of the 128 bits first result). A 21064 at 266MHz may encipher 4.7 megabytes per second (not counting the key scheduling).

The second AXP™ generation is the 21164. It may be clocked up to 600MHz, initiates 4 instructions per cycle, and locks the multiplier only 8 cycles (the result is available after 12 cycles). The tests yield a performance of about 18.8 megabytes per second. There are some resource conflicts, so this figure could be raised with an implementation performing two ciphers in parallel (slightly out of phase, in order to avoid multiplier usage conflict). There are 31 integer registers, so there is no register shortage problem.

The third generation is the new 21264, that should become available in its 600MHz version soon; a 750MHz model is announced for September 1998. It may initiate 8 instructions per cycle, and will reorder automatically the instructions. This should ease implementations. The trick of performing two different but interleaved encryptions will become almost mandatory to achieve ultimate performance. 30 megabytes per second is a reasonable expected performance.

As far as DFC is concerned, there is no cache size or memory speed issue, as the constants are in a small enough number to fit in the internal high-speed cache memory (8 KB data, 8 KB code).

Performances. We compared the performance of DFC on three architectures: Pentium™, SPARC™ and AXP™ stations. The results are summarized in the Tables 3 to 7.

3.7 Implementation on Smart Cards

We have implemented DFC on a cheap smart card based on a 8-bit processor Motorola™ 6805 which can perform byte multiplications. Two versions are proposed:

- the first one performs the key scheduling just once and stores EK 1024-bit string in RAM. It needs about 200 bytes of memory.

– the second one needs less than 100 bytes of RAM but is much slower because it computes the extended key during encryption of each block. Anyway, it may be interesting for some applications to reduce the size of the RAM and accordingly the cost of the smart cards used.

The amount of ROM needed to store the program and constant data is less than two kilo-bytes. The time needed to encrypt a 128-bit block is about 35000 cycles (resp. 200000 cycles for the second version). With a clock frequency of 3.57MHz, our implementation is able to encrypt 1632 bytes per second (resp. 285 bytes per second). This is much faster than what can be obtained with commercial implementations of 3DES on the same platform.

3.8 Time Efficiencies

Tables 1 and 2 summarize the time efficiencies (in cycles) of the various implementations. All are for a message block length of 128 and do not depend on the key length (*i.e.* 128- 192- and 256-bit key produce the same results). For this we just used the values in Tables 3 to 7 (provided at the end of this report) as $\mathrm{frq}/\mathrm{enc}_{\mathrm{time}}$, and multiply by four for the key setup or key change. The two 6805 implementations are the 200B RAM and 100B RAM versions. The latter computes the subkeys for each encryption (due to the lack of memory space).

Algorithm setup is free. Decryption is as long as encryption. Key setup and key change do not differ. We did not include the efficiency of the JAVA™ implementation, because the number of clock cycles does not make so much sense.

Version	AXP™	Pentium™ Pro	SPARC™	6805 200B	6805 100B
ANSI-C	2562	2592	5380		
Extended C	708	2432	1115		
Assembly code	558	754	802	35000	200000

Table 1. Time Efficiencies of One Block Encryption.

Version	AXP$^{\mathrm{TM}}$	Pentium$^{\mathrm{TM}}$ Pro	SPARC$^{\mathrm{TM}}$	6805 200B	6805 100B
ANSI-C	12810	12960	26900		
Extended C	3540	12160	5575		
Assembly code	2790	3770	4010	140000	1000

Table 2. Time Efficiencies of One Key Setup or Key Change.

	seconds	Kbits/s	cycles/block
ANSI-C	4.27	30696	2562
Extended C	1.18	111077	708
Assembly code	0.93	140937	558

Table 3. Encryption of 1048576 blocks on an Alpha 21164 600 MHz 128 MB RAM with OSF1 V4.0 878. Compiled with DEC$^{\mathrm{TM}}$ cc 5.6-071.

4 Extension and Uses

4.1 Different Modes

The DFC algorithm is presented as a 128-bit message block cipher. It can be extended by standard ways in order to encrypt a message of arbitrary length, for instance, with the CBC mode [2].

4.2 Encryption with 64-Bit Blocks

Some applications do not need 128-bit message blocks, and 64-bit ones are enough. We can adapt the DFC algorithm in a straightforward way.

We use a 8-round Feistel cipher with a new RF$'$ function which acts on 32-bit strings with a 64-bit string parameter $a|b$. We let

$$\mathrm{RF}'_{a|b}(x) = \mathrm{CP}'\left(\left|\left(\left(\bar{a} \times \bar{x} + \bar{b}\right) \bmod (2^{32}+15)\right) \bmod 2^{32}\right|_{32}\right)$$

where CP$'$ is a new function. For a 32-bit value $y = y_l|y_r$ where y_l is of length six and y_r is of length 26, we define

$$\mathrm{CP}'(y) = \left|\overline{(y_r \oplus \mathrm{trunc}_{26}(\mathrm{RT}(\overline{y_l})))|y_l} + \overline{\mathrm{KD}'} \bmod 2^{32}\right|_{32}$$

where KD$'$ is the 32-bit string of the 32 last bits of KD (so that KD$'$ is still odd for sure).

The key scheduling algorithm is as described in Section 1.5 but for the changes as follows.

	seconds	Kbits/s	cycles/block
ANSI-C	12.96	10113	2592
Extended C	12.16	10779	2432
Assembly code	3.77	34767	754

Table 4. Encryption of 1048576 blocks on a Pentium$^{\mathrm{TM}}$ Pro 200 MHz 128 MB RAM with Linux 2.1.95. Compiled with gcc 2.7.2.1.

	seconds	Kbits/s	cycles/block
ANSI-C	63.90	2051	5751
Extended C	33.43	3920	3008
Assembly code	11.78	11126	1060

Table 5. Encryption of 1048576 blocks on a Pentium$^{\mathrm{TM}}$ 90 MHz 16 MB RAM with Linux 2.0.30. Compiled with gcc 2.7.2.2.

- $\mathrm{PK} = \mathrm{trunc}_{128}(K|\mathrm{KS})$;
- the PK_is are 16-bit strings;
- the OAP_i, OBP_i, EAP_i, EBP_i, and $\mathrm{RV}_{i,j}$s are 32-bit strings;
- the KA_i and KB_is are truncated on their first 32 bits;
- the RK_is are 64-bit strings;
- RF' is used instead of RF.

We observe that the resulting algorithm (which we call DFC64) accepts any key with length up to 128 (all bits after the 128th are ignored). Implementation results shall be roughly four times as fast as the original DFC algorithm.

4.3 Stream Cipher and Pseudorandom Generator

Stream ciphers are traditionally the one-time pad algorithm in which the key stream is spanned by a pseudorandom generator. The US Standard [2] tells how to transform a block cipher into such stream cipher with the OFB mode.

For this we first have to choose a parameter $1 \leq t \leq 128$ and an initial 128-bit value IV. The pseudorandom generator is a finite automaton which is defined by the key K in a state ST which is initialized to $\mathrm{ST} = \mathrm{IV}$. Each iteration outputs $\mathrm{trunc}_t(\mathrm{DFC}_K(\mathrm{ST}))$ and replace ST by the last $128 - t$ bits of ST concatenated to the output. The output can be XOR-ed to a t-bit plaintext block. We

	seconds	Kbits/s	cycles/block
ANSI-C	31.65	4141	5380
Extended C	6.56	19980	1115
Assembly code	4.72	27769	802

Table 6. Encryption of 1048576 blocks on a SPARC™ 170 MHz 64 MB RAM with SunOS™ 5.5. Compiled with Workshop Compilers™ 4.2.

	seconds	Kbits/s	cycles/block
ANSI-C	18.00	7281	3600

Table 7. Encryption of 1048576 blocks on a Pentium™ Pro 200 MHz 160 MB RAM with Windows NT™ 3.51. Compiled with Visual C++™ 4.0.

recommend to use $t \leq 64$ in both the stream cipher mode and the pseudorandom generator mode.

4.4 Hashing

We believe that the DFC algorithm can be adapted into a hash function and a Message Authentication Code (MAC) algorithm by standard methods. Those approach have been discussed by various authors, *e.g.* Knudsen and Preneel [11].

A traditional way is the so called Davies–Meyer scheme which has been specified in the ISO/IEC norm [3] with the Merkle-Damgård strengthening. A message to be hashed is first padded with sufficiently enough 0 bits, one 1 bit and the original length coded in binary so that the total length is a multiple of 256. The padded message is then cut into 256-bit pieces $M_1, \ldots, M_n$. We define a 128-bit initial value $H_0 = \mathrm{IV}$. Then we iteratively define

$$H_i = \mathrm{DFC}_{M_i}(H_{i-1}) \oplus H_{i-1}.$$

The hashed value is H_n.

For a MAC, we can propose that we simply take the (eventually truncated) last block of the CBC mode encryption of the message (see Bellare-Kilian-Rogaway [5]). We can alternatively use any standard method.

4.5 Possible Platforms

We have shown that the DFC algorithm can be efficiently implemented on popular 32- and 64-bit microprocessor-based platforms, as well as cheap smart cards. Hardware-based implementations require to implement the multiplication which may be painful for the designer, but efficiently possible. Although we did not investigate all possible applications, we believe that there is no restriction on the implementability of DFC.

5 Security Analysis, Tentative Attacks

5.1 Security Results

The design construction is based on decorrelation techniques developed by Vaudenay [19–21] (see Appendix). From the results recalled in Appendix C, we know that if $a|b$ is a uniformly distributed 128-bit string we have

$$||[\mathrm{RF}_{a|b}]^2 - [R]^2|| \leq 0{,}813.2^{-58} \tag{32}$$

(see Appendix A for definitions of the notations) where R is a truly random 64-bit to 64-bit function. Thus a three-round Feistel cipher which uses RF has a pairwise decorrelation distance less than $0{,}641.2^{-56}$ to the Perfect Cipher, and a six-round Feistel cipher which uses RF has a pairwise decorrelation distance less than $0{,}821.2^{-113}$ to the Perfect Cipher.

From the results in Appendix B we thus know that a differential distinguisher requires more than 2^{107} chosen plaintexts pairs in order to achieve an advantage of 1%. Similarly, a linear distinguisher requires more than 2^{81} known plaintexts in order to achieve an advantage of 1%. These are attacks against a six-round encryption function. Attacks against eight rounds need even greater complexities.

Another (weaker but more general) result proves that any known plaintext attack with order 1 needs more than 2^{43} known plaintexts in order to achieve an advantage of 1% against the six-round cipher. This suggests that the key should not be used more than 2^{43} times *i.e.* that we should not encrypt 128TB with the same key. We believe that this restricts no practical application.

The key-scheduling algorithm breaks the uniformity of the expanded key sequence. The results above only hold for uniformly distributed expanded keys. The key scheduling algorithm makes only four calls to each of the tiny ciphers $\mathrm{Enc}_{\mathrm{EF}_1(K)}$ and $\mathrm{Enc}_{\mathrm{EF}_2(K)}$. We can thus prove that the previous security results are still valid by making assumptions like

> "we cannot built any distinguisher between $\mathrm{Enc}_{\mathrm{EF}_1(K)}$ (resp. $\mathrm{Enc}_{\mathrm{EF}_2(K)}$) and a truly random permutation which is limited to four calls and which achieves an advantage greater than 1% with a limited budget of US$1,000,000,000."

A more precise treatment on this problem is stated in Appendix D.

5.2 Exhaustive Search

Assuming that the best implementation achieves a 30MBps encryption rate, we can estimate the real cost of exhaustive search. Secret key trials require to optimize the key scheduling algorithm. Its cost is roughly four times the cost of one encryption. Hence we can try up to 400,000 keys per second with the best implementation. In order to upper bound the complexity of the exhaustive search, we make some pessimistic assumption. We assume that the opponent can get one million of such devices with a technology 1000 times as fast (which approximates what would be the fastest platforms we can get in a 15 years according to Moore's empiric law). The opponent can then try up to 2^{49} keys per second. Exhaustive search on k-bit keys thus requires $2^{k-65.4}$ days. Exhaustive search on 64-bit keys thus requires at least 13 hours. Exhaustive search on 80-bit keys requires one century with this technology. This is of course not accurate since technology makes chips faster and faster. We can thus reasonably conclude that no 80-bit key search is possible within less than several computation decades.

We can make more accurate estimates by assuming Moore's law. We assume that the opponent can try up to $f_0 = 400{,}000{,}000{,}000$ keys per second at time $t = 0$ (*i.e.* now). At time t, he can try $f_0.2^{\frac{t}{\tau}}$ keys per second where $\tau = 1.5$ years. Then he can try

$$\int_0^T f_0.2^{\frac{t}{\tau}} dt = \frac{f_0 \tau}{\log 2} 2^{\frac{T}{\tau}}$$

keys within a time T. Exhaustive search of a k-bit key requires on average 2^{k-1} trials. Hence the average time of the exhaustive search is

$$T = k\tau + \tau\left(\log_2 \frac{\log 2}{2 f_0 \tau}\right) \approx \tau(k - 65.6).$$

Each bit thus requires 1.5 years more on average. Table 8 estimates the average complexity of exhaustive search for various key size in this setting.

key length	complexity
80	21.65
128	93.65
192	126.43
256	190.43

Table 8. Estimates of the average time complexity (in years) of the exhaustive search for various key size assuming that we can try up to $400.10^9.2^{\frac{t}{1.5 \text{ years}}}$ keys per second at time t.

5.3 Higher Order Differentials

The present cipher resists to classical differential cryptanalysis by making some pairwise decorrelation distance small. However we can still consider higher order attacks. For instance we can consider differentials of order two.

Let x_1, x_2, x_3, x_4 be four inputs of RF with *exclusive or* zero: $x_1 \oplus x_2 \oplus x_3 \oplus x_4 = 0$. Let us first consider a weakened RF function defined by

$$\mathrm{RF}_{a|b}(x) = \mathrm{CP}\left((a * x) \oplus b\right)$$

where $*$ denotes the $\mathrm{GF}(2^{64})$-product, and with

$$\mathrm{CP}(y) = (y_r \oplus \mathrm{RT}(\overline{\mathrm{trunc}_6(y_l)}))|(y_l \oplus \mathrm{KC})$$

($i.e.$ we consider $\mathrm{KD} = 0$). Let $y_i = (a * x_i) \oplus b$. Obviously, we have $y_1 \oplus y_2 \oplus y_3 \oplus y_4 = 0$. The four values of $\mathrm{RT}(\overline{\mathrm{trunc}_6(y_i)})$ are pairwise equal with probability 3.2^{-6}. Hence the four outputs of RF have an *exclusive or* zero with this probability. This can be used to mount an attack on the whole cipher with this weakened RF function. Namely, we consider the following distinguisher.

Input: an oracle $m \mapsto C(m)$

1. for j from 1 to n, do
   (a) pick a random constant m^r and four values m_i^l, $i = 1, 2, 3, 4$ with an *exclusive or* of zero
   (b) get $C(m_i^l|m^r)$, $i = 1, 2, 3, 4$
   (c) let u_i denote the 64 rightmost bits of $C(m_i^l|m^r)$
   (d) if $\bigoplus_{i=1}^{4} u_i = 0$, stop and output 1
2. output 0

The main loop will stop and output 1 with probability $(3.2^{-6})^6 \approx 1.42 \times 2^{-27}$. Thus we will distinguish the simplified cipher from a truly random permutation for $n \approx 1.40 \times 2^{26}$.

We argue that this attack cannot work on DFC for several reasons.

First, let us consider the case

$$\mathrm{RF}_{a|b}(x) = |\bar{a}\bar{x} + \bar{b} \bmod (2^{64} + 13) \bmod 2^{64}|_{64}.$$

We can reasonably expect that the $x_1 \oplus \ldots \oplus x_4 = 0$ property is transformed into a $y_1 - y_2 \equiv y_3 - y_4 \pmod{2^{64}}$ (for instance when $x_1 \oplus x_2$ has a very low Hamming weight), and that the y_is are almost uniformly distributed. But then the following result shows that this property is very different to the $y_1 \oplus \ldots \oplus y_4 = 0$ one, so we cannot propagate it in the Feistel scheme.

Theorem 1. *Let $Y_1, \ldots, Y_4$ be random n-bit strings with uniform and independent distribution. We have*

$$\Pr[\overline{Y_1 \oplus \ldots \oplus Y_4} = 0/\bar{Y}_1 - \bar{Y}_2 \equiv \bar{Y}_3 - \bar{Y}_4 \pmod{2^n}] = \left(\frac{3}{4}\right)^n.$$

Proof. From the relation

$$\overline{x \oplus y} = \bar{x} - \bar{y} + 2\overline{y \wedge \neg x} \bmod 2^n$$

where n is the length of x and y, we obtain that $\overline{Y_1 \oplus \ldots \oplus Y_4} = 0$ if, and only if

$$2\overline{|\bar{Y}_1 + \delta| \wedge \neg Y_1} = 2\overline{|\bar{Y}_3 + \delta| \wedge \neg Y_3}$$

where $\delta = \bar{Y}_1 - \bar{Y}_2 = \bar{Y}_3 - \bar{Y}_4$ (mod 2^n operations are omitted here). We let $f_n(\delta)$ denotes the probability that this holds. We have $f_n(2\delta) = f_{n-1}(\delta)$ and

$$f_n(2\delta+1) = \frac{1}{4} f_{n-1}(\delta) + \frac{1}{4} f_{n-1}(\delta + 1 \bmod 2^{n-1}).$$

This equation comes from the fact that the least significant bit of Y_1 and Y_3 must be equal (with probability $\frac{1}{2}$) and from separating the cases where this common bit is 0 or 1. Then it is straightforward that $E(f_n(\delta)) = \left(\frac{3}{4}\right)^n$ by induction where δ is uniformly distributed. $\square$

Second, let z_i denotes the result of the actual $\mathrm{RF}_{a|b}(x)$ value before being added to KD. We assume that we can expect that $z_1 \oplus \ldots \oplus z_4 = 0$. The following theorem indicates why we choose a KD constant with its rightmost bit set to one.

Theorem 2. *Let $Z_1, \ldots, Z_4$ be random n-bit strings with uniform and independent distributions. We let $g(z)$ denotes $|\bar{z} + \overline{\mathrm{KD}} \bmod 2^n|_n$. Let $g(z)_i$ be the ith bit of $g(z)$ (with the convention that the leftmost one is the first). We have*

$$\Pr[g(Z_1)_i \oplus \ldots \oplus g(Z_4)_i = 0 / Z_1 \oplus Z_2 \oplus Z_3 \oplus Z_4 = 0] = 1$$

for $i > j - 2$ where j is the number of the rightmost nonzero bit of KD. *The probability for $i = j - 2$ is $\frac{1}{2} + \frac{1}{8}$.*

Proof. We use the formula

$$\bar{z} + \overline{\mathrm{KD}} \bmod 2^n = \overline{z \oplus \mathrm{KD}} + 2(\overline{z \wedge \mathrm{KD}}) \bmod 2^n.$$

Thus $\bar{z} + \overline{\mathrm{KD}} \bmod 2^n$ is equal to

$$\begin{array}{l} \overline{z \oplus \mathrm{KD} \oplus (Sz \wedge S\mathrm{KD})} + \\ \quad \overline{(Sz \wedge S^2 z \wedge S^2\mathrm{KD}) \oplus (S\mathrm{KD} \wedge S^2 z \wedge S^2 \mathrm{KD})} \bmod 2^n \end{array}$$

where S is the logical shift operation from one position to the left (with the leftmost bit dropped and one rightmost zero bit pushed). From this the Theorem 2 is straightforward for $i > j-2$. For $i = j-2$, let $u = Z_1 \oplus Z_2$ and $v = Z_3 \oplus Z_4$. We show that $Z_1 \oplus Z_2 \oplus Z_3 \oplus Z_4 = 0$ implies that the ith bit is equal to $u_j v_{j-1} \oplus u_{j-1} v_j$. $\square$

5.4 Weak Keys with ARK $= |0|_{64}$

(We let RK = ARK|BRK for convenience.) If the a parameter of RF is equal to zero, then $\mathrm{RF}_{a|b}(x)$ is a constant which does not depend on x. Hence, if we assume that $\mathrm{ARK}_2 = \mathrm{ARK}_4 = \mathrm{ARK}_6 = |0|_{64}$, we have a trivial distinguisher.

Input: an oracle $m \mapsto C(m)$ with $\mathrm{ARK}_2 = \mathrm{ARK}_4 = \mathrm{ARK}_6 = |0|_{64}$

1. for j from 1 to 4 do
   (a) pick a random constant m^r and two values m_i^l, $i = 1, 2$
   (b) get $C(m_i^l|m^r)$, $i = 1, 2$
   (c) let u_i denotes the 64 rightmost bits of $C(m_i^l|m^r)$
   (d) if $u_1 \oplus u_2 \neq m_1^l \oplus m_2^l$, stop and output 0
2. output 1

The advantage is almost 1 if, and only if $\mathrm{ARK}_2 = \mathrm{ARK}_4 = \mathrm{ARK}_6 = |0|_{64}$ here. This proves that the keys for which we have $\mathrm{ARK}_2 = \mathrm{ARK}_4 = \mathrm{ARK}_6 = |0|_{64}$ are fairly weak. It is however quite unlikely that $\mathrm{ARK}_2 = \mathrm{ARK}_4 = \mathrm{ARK}_6 = |0|_{64}$ occurs since it happens with probability 2^{-192}.

We outline another property which may come from a bad choice of the KA_i and KB_i strings. If we have $\mathrm{KA}_3 = \mathrm{KB}_3 = |0|_{64}$, $\mathrm{KA}_2 = \mathrm{KA}_4$ and $\mathrm{KB}_2 = \mathrm{KB}_4$, then we have a very weak keys K such that $\mathrm{OAP}_2 = \mathrm{EAP}_2 = |0|_{64}$. For this we obtain that $\mathrm{OAP}_1 = \mathrm{OAP}_3$ and $\mathrm{OAP}_2 = \mathrm{OAP}_4 = |0|_{64}$ so that $\mathrm{Enc}_{\mathrm{EF}_1(K)}(u|v) = v|u$. Similarly we have $\mathrm{Enc}_{\mathrm{EF}_2(K)}(u|v) = v|u$. Thus we obtain that $\mathrm{ARK}_i = \mathrm{BRK}_i = |0|_{64}$ for all i and thus $\mathrm{DFC}_K(u|v) = (v \oplus c)|(u \oplus c)$ for some given c strings, which is a very insecure permutation. This class of weak keys vanishes with a good choice of the KA_is and KB_is.

5.5 Other Weak Keys

The (a, b) pairs subject to the relation $-14\bar{a} + \bar{b} \equiv -14 \pmod{2^{64} + 13}$ can be used with entry pairs such that $\bar{x} + \bar{x}' = 2^{64} - 1$. For these pairs we have $\bar{x} + \bar{x}' = \overline{x \oplus x'}$. We have

$$(\bar{a}\bar{x} + \bar{b}) + (\bar{a}\bar{x}' + \bar{b}) = \bar{a}(\bar{x} + \bar{x}') + \bar{b} \equiv 2^{64} - 1 \pmod{(2^{64} + 13)}.$$

Hence is x and x' are bitwise complement, y and y' are bitwise complement with high probability (where $\bar{y} = \bar{a}\bar{x} + \bar{b} \bmod (2^{64} + 13) \bmod 2^{64}$).

Although we do not know how to use this property, we believe that it is a weakness so that $(\mathrm{ARK}, \mathrm{BRK})$ subkeys such that $-14\overline{\mathrm{ARK}} + \overline{\mathrm{BRK}} \equiv -14 \pmod{(2^{64}+13)}$ may be considered as weak subkeys.

5.6 Photofinishing Attack

Biham introduced an efficient implementation technique based on 1-bit SIMD microprocessor approach. This kind of implementation is made hard by the carry propagation in modular additions and multiplications. This has the positive consequence that Shamir's photofinishing attack [18] is made impossible.

5.7 Timing Attacks

Implementations must be such that computations of the modular multiplication does not leak any information by time measurement (*i.e.* we must avoid tests and conditional branches that depend on the computation). Otherwise this may leak some information by Kocher's timing attacks [10]. Our assembly code implementations fulfill this requirement.

5.8 Attacks on Four Rounds

We can attack DFC with a number of rounds reduced down to four. We outline three attacks below. All these attacks are chosen plaintext attacks where we query the encryption oracle $\mathrm{Enc}_{\mathrm{RK}_1,\ldots,\mathrm{RK}_4}$ with random messages $m = x|c$ where x is a random 64-bit string and c is a 64-bit constant. We let

$$\mathrm{Enc}_{\mathrm{RK}_1,\ldots,\mathrm{RK}_4}(x|c) = h(x)|g(x)$$

so that we can write

$$\begin{aligned}
y &= x \oplus \mathrm{RF}_{\mathrm{RK}_1}(c)\\
g(y) &= \mathrm{RF}_{\mathrm{RK}_2}(y)\\
f(x) &= y \oplus \mathrm{RF}_{\mathrm{RK}_3}(c \oplus g(y))\\
h(x) &= c \oplus g(y) \oplus \mathrm{RF}_{\mathrm{RK}_4}(f(x)).
\end{aligned}$$

The main observation (but for the first attack) is that $\mathrm{RF}_{\mathrm{RK}_2}$ has hardly any collisions. Namely, for any 128-bit round key RK_2, there exist up to 13 pairs $\{y, y'\}$ such that $\mathrm{RF}_{\mathrm{RK}_2}(y) = \mathrm{RF}_{\mathrm{RK}_2}(y')$.

Differential Collision-Attacks on f. In this attack we try about 2^{32} possible x values until we get a collision $f(x_1) = f(x_2)$. We obtain that there exists a pair $\{y_1.y_2\}$ such that

$$y_1 \oplus y_2 = x_1 \oplus x_2$$
$$g(y_1) \oplus g(y_2) = h(x_1) \oplus h(x_2).$$

Since we know an input difference and an output difference for g, we can further attack on RK_2.

Collision Attack on $\mathrm{RF}_{\mathrm{RK}_2}$. In this attack we try about all of the 2^{64} possible x values until we get a collision $x_1 \oplus f(x_1) = x_2 \oplus f(x_2)$. We assume that this comes from a collision on $g = \mathrm{RF}_{\mathrm{RK}_2}$. (It can comes from a collision on $\mathrm{RF}_{\mathrm{RK}_3}$ as well.) For each of the 2^{64} possible values of $\mathrm{RF}_{\mathrm{RK}_1}(c)$ we can compute y_1 and y_2. Since we known this leads to a collision on g, we must have

$$\mathrm{ARK}_2 \times (y_1 - y_2) \equiv \pm 2^{64} \pmod{2^{64}}.$$

Thus we obtain two possible values for ARK_2, and for each of it we obtain 13 possible values for BRK_2. To each possible value of $\mathrm{RF}_{\mathrm{RK}_1}(c)$ we can therefore compute 26 possible candidates for RK_2. The $(c, \mathrm{RF}_{\mathrm{RK}_1}(c))$ pair enables next to recover the whole key within an attack of complexity close to 2^{64}.

Distinguishing Collision Attack on $\mathrm{RF}_{\mathrm{RK}_2}$. In this attack, we aim to distinguish the encryption oracle from a truly random permutation oracle. We try 2^{32} possible x values. If there is a collision $x_1 \oplus f(x_1) = x_2 \oplus f(x_2)$, we output 0, otherwise, we output 1. For the four-round DFC oracle, the probability that the answer is 0 is about $2^{-27} \simeq 0$. For a truly random permutation oracle, the probability that the answer is 1 is about 0.63. The advantage is thus 0.37.

We conclude that the DFC algorithm is weak when reduced to four rounds. We believe that eight attacks are enough for security. (The above attacks are at least not applicable.)

6 Conclusion

We have proposed a dedicated block cipher algorithm which is faster than DES and hopefully more secure than triple-DES. In addition we provided proofs of security against some classes of general simple attacks which includes differential and linear cryptanalysis. This result is based on the decorrelation theory. We believe that this cipher is also "naturally" secure against more complicated attacks since our design introduced no special algebraic property. To summarize, our design is guaranteed to be vulnerable against neither differential nor linear cryptanalysis with complexity less than 2^{81} encryptions. We believe that the best attack is still exhaustive search which is limited by the implementation speed (decreased by a factor of 5 due to the key scheduling algorithm). We (very pessimistically) forecast that one need at least several decades to search a 80-bit key, which makes it safe until the Advanced Encryption Standard expires. Extrapolation estimates that 128-bit key is safe for 93 years, 192-bit key is safe for 126 years, and 256-bit is safe for 190 years.

Another theoretical result claims that if we admit that no key will be used more than 2^{43} times, then the cipher is guaranteed to resist to any iterated known plaintext attack of order 1.

Our algorithm accepts 128-bit message blocks and any key size from 0 to 256. It can be adapted into a 64-bit variant (with a key size up to 128). We believe that it can be adapted to any other cryptographic primitive such as stream cipher, hash function, MAC algorithm although we did not deeply investigate this issue.

Our algorithm can be implemented on traditional personal computers, as well as on cheap smart cards. We believe that it can be implemented in any other digital environment.

In conclusion we recommend this encryption algorithm as a candidate to the Advanced Encryption Standard process.

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. *Information Technology — Security Techniques — Hash-Functions.* ISO/IEC 10118, 1994.
4. 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.

5. M. Bellare, J. Kilian, P. Rogaway. The security of cipher block chaining. In *Advances in Cryptology CRYPTO'94*, Santa Barbara, California, U.S.A., Lectures Notes in Computer Science 839, pp. 341–358, Springer-Verlag, 1994.
6. 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.
7. E. Biham, A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.
8. H. Feistel. Cryptography and computer privacy. *Scientific American*, vol. 228, pp. 15–23, 1973.
9. T. Jakobsen, L. R. Knudsen. The interpolation attack on block ciphers. In *Fast Software Encryption*, Haifa, Israel, Lectures Notes in Computer Science 1267, pp. 28–40, Springer-Verlag, 1997.
10. P. Kocher. Timing attacks in implementations of Diffie-Hellman, RSA, DSS and other systems. In *Advances in Cryptology CRYPTO'96*, Santa Barbara, California, U.S.A., Lectures Notes in Computer Science 1109, pp. 104–113, Springer-Verlag, 1996.
11. L. R. Knudsen, B. Preneel. Fast and secure hashing based on codes. In *Advances in Cryptology CRYPTO'97*, Santa Barbara, California, U.S.A., Lectures Notes in Computer Science 1294, pp. 485–498, Springer-Verlag, 1997.
12. M. Luby, C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, vol. 17, pp. 373–386, 1988.
13. 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.
14. 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.
15. 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.
16. K. Nyberg, L. R. Knudsen. Provable security against a differential cryptanalysis. *Journal of Cryptology*, vol. 8, pp. 27–37, 1995.
17. C. E. Shannon. Communication theory of secrecy systems. *Bell system technical journal*, vol. 28, pp. 656–715, 1949.
18. A. Shamir. Visual cryptanalysis. In *Advances in Cryptology EUROCRYPT'98*, Espoo, Finland, Lectures Notes in Computer Science 1403, pp. 201–209, Springer-Verlag, 1998.
19. S. Vaudenay. Provable security for block ciphers by decorrelation. In *STACS 98*, Paris, France, Lectures Notes in Computer Science 1373, pp. 249–275, Springer-Verlag, 1998.
20. S. Vaudenay. Provable security for block ciphers by decorrelation. (Journal Version.) Submitted.
21. S. Vaudenay. The decorrelation technique home-page.
URL:`http://www.dmi.ens.fr/~vaudenay/decorrelation.html`
22. G. S. Vernam. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Journal of the American Institute of Electrical Engineers*, vol. 45, pp. 109–115, 1926.

A Basic Definitions in Decorrelation Theory

We briefly recall the basic definitions used in the decorrelation theory and results taken from [19–21]. Firstly, let us recall the notion of d-wise distribution matrix associated to a random function. This matrix characterizes its d-wise decorrelation.

Definition 3. *Given a random function F from a given set $\mathcal{A}$ to a given set $\mathcal{B}$ and an integer d, we define the d-wise distribution matrix $[F]^d$ of F as a $\mathcal{A}^d \times \mathcal{B}^d$-matrix where the (x, y)-entry of $[F]^d$ corresponding to the multi-points $x = (x_1, \ldots, x_d) \in \mathcal{A}^d$ and $y = (y_1, \ldots, y_d) \in \mathcal{B}^d$ is defined as the probability that we simultaneously have $F(x_i) = y_i$ for $i = 1, \ldots, d$.*

Secondly, we recall the definition of the $|||.|||_\infty$ matrix norm[1] which will be denoted $||.||$ throughout this report since it is the only one considered here.

Definition 4. *Given a matrix A, we define*

$$||A|| = \max_x \sum_y |A_{x,y}|$$

where the sums run over all the (x, y)-entries of the matrix A.

We recall that $||.||$ is a matrix norm (*i.e.* that the norm of any matrix-product $A \times B$ is at most the product of the norms of A and B). Finally, the definition of the general d-wise decorrelation distance between two random functions is as follows.

Definition 5. *Given two random functions F and G from a given set $\mathcal{A}$ to a given set $\mathcal{B}$, an integer d and a matrix norm $||.||$ over the vector space $\mathbf{R}^{\mathcal{A}^d \times \mathcal{B}^d}$, we call $||[F]^d - [G]^d||$ the d-wise decorrelation distance between F and G.*

We consider a block ciphers on a message-block space $\mathcal{M} = \{0, 1\}^m$ with a key represented by a random variable K and a random permutation C defined by K over $\mathcal{M}$. Ideally, we consider the Perfect

[1] The strange $|||.|||_\infty$ notation used in [19] comes from the fact that this norm is associated to the usual $||.||_\infty$ norm over the vectors defined by $||V||_\infty = \max_x |V_x|$ by

$$|||A|||_\infty = \max_{||V||_\infty = 1} ||AV||_\infty.$$

Cipher C^* for which the distribution of C^* is uniform over the set of all permutations over $\mathcal{M}$. Hence for any multi-points $x = (x_1, \ldots, x_d)$ and $y = (y_1, \ldots, y_d)$ with pairwise different entries, we have

$$[C^*]^d_{x,y} = \Pr[C^*(x_i) = y_i; i = 1, \ldots, d] = \frac{1}{2^m \ldots (2^m - d + 1)}.$$

The following Lemma makes the decorrelation distance a friendly measurement thanks to the matrix norm property.

Lemma 6. *Let $||.||$ be a matrix norm over $\mathbf{R}^{\mathcal{M}^d \times \mathcal{M}^d}$. For any independent random ciphers C_1, C_2, C^* where C^* is a Perfect Cipher, we have*

$$||[C_1 \circ C_2]^d - [C^*]^d|| \leq ||[C_1]^d - [C^*]^d||.||[C_2]^d - [C^*]^d||. \qquad (33)$$

This property comes from the easy facts $[C_1 \circ C_2]^d = [C_2]^d \times [C_1]^d$ and $[C^*]^d \times [C_1]^d = [C^*]^d$.

We are mostly interested into Feistel Ciphers over $\mathcal{M} = \mathcal{M}_0^2$ (where $\mathcal{M}_0$ is a given group) which are defined with random round functions $F_1, \ldots, F_r$ on $\mathcal{M}_0$. (In most of practical examples we have $\mathcal{M}_0 = \mathbf{Z}_2{}^{\frac{m}{2}}$.) We denote $C = \Psi(F_1, \ldots, F_r)$ the cipher defined by $C(x^l, x^r) = (y^l, y^r)$ where we iteratively compute a sequence (x_i^l, x_i^r) such that

$$\begin{aligned} x_0^l &= x^l \text{ and } x_0^r = x^r \\ x_i^l &= x_{i-1}^r \text{ and } x_i^r = x_{i-1}^l + F_i(x_{i-1}^r) \\ y^l &= x_r^r \text{ and } y^r = x_r^l \end{aligned}$$

where $+$ is the group operation (see Feistel [8]).

B Security Results in Decorrelation Theory

The following security result taken from [20], shows how d-wise decorrelation distance zero between C and C^* implies unconditional security of C.

Theorem 7. *Let C be a cipher which has a d-wise decorrelation zero to the Perfect Cipher. For any sequence of messages $x_1, \ldots, x_{d-1}$, if*

X denotes a random variable whose values are in $\mathcal{M}\backslash\{x_1, \ldots, x_{d-1}\}$, we have

$$H(X/C(x_1), \ldots, C(x_{d-1}), C(X)) = H(X)$$

where H denotes Shannon's entropy of random variables.

This means that if an adversary knows $d-1$ pairs $(x_i, C(x_i))$, then, for any y_d which is different from all $C(x_i)$s, his knowledge of $C^{-1}(y_d)$ is *exactly* that it is different from all x_is.

Decorrelation distance also enables to quantify the security of ciphers. Here we consider the security in the Luby-Rackoff model [12]. We consider an opponent as an infinitely powerful Turing machine which has a limited access to an encryption oracle device and whose aim is to distinguish whether the device implements a given practical cipher $C_1 = C$ or a given ideal cipher C_2 which is usually $C_2 = C^*$. When fed with an oracle c, the Turing machine $\mathcal{T}^c$ returns either 0 or 1 (which can be probabilistic). More precisely, a distinguisher with d oracle calls is represented by any Turing machine which complies with the following model:

Input: an oracle c

1. calculate a message X_1 and get $Y_1 = c(X_1)$
2. calculate a message X_2 and get $Y_2 = c(X_2)$
3. ...
4. calculate a message X_d and get $Y_d = c(X_d)$
5. depending on $X = (X_1, \ldots, X_d)$ and $Y = (Y_1, \ldots, Y_d)$, output 0 or 1

If we want to distinguish a random cipher C from C^*, we let p (resp. p^*) denotes $\Pr[\mathcal{T}^C = 1]$ (resp. $\Pr[\mathcal{T}^{C^*} = 1]$). We say that the attack is successful if $|p - p^*|$ is large. We call $|p - p^*|$ the *advantage* of the distinguisher. Conversely, we say that the cipher C resists the attack if we have $|p - p^*| \leq \epsilon$ for some small ϵ. This model is quite powerful: we prove that a cipher C cannot be distinguished from the Perfect Cipher C^*, then any attempt to decrypt a ciphertext provided by C will also be applicable against the cipher C^* for which we know the security. (For more motivation on this security model, see Luby-Rackoff [12].)

We call *non-adaptive* a d-limited distinguisher in the following model:

Input: an oracle c

1. calculate some messages $X = (X_1, \ldots, X_d)$

2. get $Y = (c(X_1), \ldots, c(X_d))$
3. depending on X and Y, output 0 or 1

From [19] we know that the d-wise decorrelation distance between C and C^* quantifies the security against any non-adaptive attack.

Theorem 8. *Let C be a cipher; let d be an integer and ϵ be the d-wise decorrelation distance between C and the Perfect Cipher C^*. The greatest advantage for any d-limited non-adaptive distinguisher is equal to $\epsilon/2$.*

We call *differential distinguisher* with the characteristic (a, b) and complexity n the following algorithm:

Input: a cipher c, a complexity n, a characteristic (a, b)

1. for i from 1 to n do
   (a) pick uniformly a random X and query for $c(X)$ and $c(X \oplus a)$
   (b) if $c(X \oplus a) = C(X) \oplus b$, stop and output 1
2. output 0

Theorem 9. *Let C be a cipher on a space $\mathcal{M}$ of size 2^m, and let C^* be the Perfect Cipher. For any differential distinguisher between C and the Perfect Cipher C^* with complexity n, the advantage is at most $\frac{n}{2^m-1} + \frac{n}{2}||[C]^2 - [C^*]^2||$.*

Similarly, we call *linear distinguisher* with the characteristic (a, b) and complexity n the following algorithm:

Input: a cipher c, a complexity n, a characteristic (a, b), a set A

1. initialize the counter value t to zero
2. for i from 1 to n do
   (a) pick a random X with a uniform distribution and query for $c(X)$
   (b) if $X \cdot a = c(X) \cdot b$, increment the counter t
3. if $t \in A$, output 1, otherwise output 0

Theorem 10. *Let C be a cipher on a space $\mathcal{M}$ of size 2^m, and let C^* be the Perfect Cipher. For any linear distinguisher between C and the Perfect Cipher C^* with complexity n, the advantage $|p - p^*|$ is such that*

$$\lim_{n \to +\infty} \frac{|p - p^*|}{n^{\frac{1}{3}}} \leq 9.3 \left(\frac{1}{2^m - 1} + 2\epsilon \right)^{\frac{1}{3}}$$

where $\epsilon = ||[C]^2 - [C^*]^2||$.

Theorem 9 and 10 mean that if $\epsilon = ||[C]^2 - [C^*]^2|| >> 2^{-m}$, then we need a complexity $n = \Omega(1/\epsilon)$ to attack C in either way. They also mean that C is secure against any differential or linear distinguisher if $||[C]^2 - [C^*]^2|| << 2^{-m}$.

There is another result which is a weaker but more general. We consider a general iterated attack of order d *i.e.* an attack in the following model.

Input: a cipher c, a complexity n, a distribution on X, a test $\mathcal{T}$, an acceptance set $\mathcal{A}$

1. for i from 1 to n do
   (a) get a new $X = (X_1, \ldots, X_d)$
   (b) get $Y = (c(X_1), \ldots, c(X_d))$
   (c) set $T_i = 0$ or 1 with an expected value $\mathcal{T}(X, Y)$
2. if $(T_1, \ldots, T_n) \in \mathcal{A}$ output 1 otherwise output 0

Here is the corresponding security result.

Theorem 11. *Let C be a cipher on a message space of size M such that, if C^* is the perfect cipher with uniform distribution, then $||[C]^{2d} - [C^*]^{2d}|| \leq \epsilon$ for a given $d \leq M/2$. For any iterated attack of order d between C and C^* such that the obtained plaintexts are independent, we have*

$$|p - p^*| \leq 3\left(\left(2\delta + \frac{5d^2}{2M} + \frac{3\epsilon}{2}\right)n^2\right)^{\frac{1}{3}} + \frac{n\epsilon}{2}$$

where δ is the probability that for two independent X and X' there exists i and j such that $X_i = X'_j$.

In particular, for any known plaintext attack in which the X_is are independent and uniformly distributed, we have

$$|p - p^*| \leq 3\left(\left(\frac{7d^2}{2M} + \frac{3\epsilon}{2}\right)n^2\right)^{\frac{1}{3}} + \frac{n\epsilon}{2}.$$

C Decorrelation of the PEANUT Family

The DFC cipher is in the PEANUT Cipher Family [19]. PEANUT Ciphers are characterized by some parameters (m, r, d, p). They are Feistel Ciphers with block length of m bits (m even), r rounds. The parameter d is the order of partial decorrelation that the cipher achieves, and p must be a prime number greater than $2^{\frac{m}{2}}$. For DFC, we have $m = 128$, $r = 8$, $d = 2$ and $p = 2^{64} + 13$.

The cipher is defined by a key of $\frac{mrd}{2}$ bits which consists of a sequence of r lists of d $\frac{m}{2}$-bit strings $k_0, \ldots, k_{d-1}$, one for each round. In each round, the F function has the form

$$F(x) = g\left(\left|\sum_{i=0}^{d-1} \bar{k}_i.\bar{x}^i \bmod p \bmod 2^{\frac{m}{2}}\right|_{\frac{m}{2}}\right)$$

where g is any permutation on the set of all $\frac{m}{2}$-bit strings.

We have the following result which enables to get an upper bound on the decorrelation distance.

Theorem 12. *Let F be any round function used in any cipher in the PEANUT family with parameters (m, r, d, p). Let R be a random function with uniform distribution, We have*

$$||[F]^d - [R]^d|| \leq 2\left(p^d 2^{-\frac{md}{2}} - 1\right).$$

Theorem 13. *Let C be a cipher in the PEANUT family with parameters (m, r, d, p) and let C^* be the Perfect Cipher. We have*

$$||[C]^d - [C^*]^d|| \leq \left(\left(1 + 2\left(p^d 2^{-\frac{md}{2}} - 1\right)\right)^3 - 1 + \frac{2d^2}{2^{\frac{m}{2}}}\right)^{\lfloor \frac{r}{3} \rfloor}.$$

When $p = 2^{\frac{m}{2}} + \delta$ with $\delta << 2^{\frac{m}{2}}$, this can be approximated to

$$\left(2d(3\delta + d)2^{-\frac{m}{2}}\right)^{\lfloor \frac{r}{3} \rfloor}.$$

D Security on the Key Scheduling Algorithm

We bound here the security loss introduced by the key scheduling algorithm. For this, we prove that if some attack were possible against the real DFC algorithm, then it would *roughly* be possible against Enc_K with a truly 1024-bit random key K, unless some reasonable assumption is false.

Let S_i, $i = 1, \ldots, 4$ denotes four random 1024-bit strings with different distributions. Let us denote

$$S_i = A_1^i | B_1^i | A_2^i | \ldots | B_4^i$$

where A_j^i and B_j^i are 128-bit strings. The distribution of S_1 is uniform. The distribution of S_2 is defined by

$$A_{j+1}^2 = F(A_j^2) \quad B_{j+1}^2 = G(B_j^2)$$

and $A_0^2 = B_0^2 = |0|_{64}$, F and G being two random permutations uniformly distributed. The distribution of S_3 is defined by

$$A_{j+1}^3 = \mathrm{Enc}_{\mathrm{EF}_1(K)}(A_j^3) \quad B_{j+1}^3 = G(B_j^3)$$

and $A_0^2 = B_0^2 = |0|_{64}$, where K is uniformly distributed. The distribution of S_4 is defined by $S_4 = \mathrm{EF}(K)$ where K is uniformly distributed.

We make the following assumption.

Hypothesis 14. *The best advantage for any distinguisher between* $\mathrm{Enc}_{\mathrm{EF}_i(K)}$ *and the Perfect Cipher which is limited to 4 queries and within a maximum cost of c is given by the function $h(c)$.*

Here the notion of "cost" includes the *real* cost of an attack, *i.e.* the cost for developing it, implementing it, running it, ...

Here is our main result.

Theorem 15. *Let $\mathcal{A}$ be an attack which distinguishes* DFC_K *from the Perfect Cipher within a cost c and an advantage of ϵ. The same attack is able to distinguish* Enc_{S_1} *from the Perfect Cipher with the same cost and advantage greater than $\epsilon - 2h(c + c_0) - 2^{-122}$ where c_0 is a negligible cost (e.g. less than US\$1).*

As an example of application, we know that distinguishing Enc_{S_1} from the Perfect Cipher through a differential attack requires more than 2^{107} chosen plaintexts in order to achieve an advantage of 1%. Assuming that $h(c) \leq 1\%$ for the cost c of the attack, our Theorem proves that there is no differential attack of cost c and less than 2^{107} chosen plaintexts which achieves an advantage of 3% (the c_0 cost and the advantage 2^{-122} are negligible). As a cryptanalysis challenge, we can thus propose to make a distinguisher between $\mathrm{Enc}_{\mathrm{EF}_1(K)}$ and the Perfect Cipher which achieves an advantage greater than 1%, which is limited to 4 chosen plaintexts and with a limited budget of US\$$1,000,000,000$. If this is not possible, then all our results on Enc_K are applicable on DFC_K.

Proof. Let ϵ' be the advantage of $\mathcal{A}$ for distinguishing Enc_{S_1} from the Perfect Cipher. We let $p^{\mathrm{Enc}_{S_i}}$ be the probability that $\mathcal{A}$ outputs 1 when fed with Enc_{S_i}. Obviously, we have

$$\epsilon \leq \epsilon' + |p^{\mathrm{Enc}_{S_4}} - p^{\mathrm{Enc}_{S_3}}| + |p^{\mathrm{Enc}_{S_3}} - p^{\mathrm{Enc}_{S_2}}| + |p^{\mathrm{Enc}_{S_2}} - p^{\mathrm{Enc}_{S_1}}|.$$

The first two differences can be studied in the same way. For instance, let $\mathcal{B}$ be a distinguisher between $\mathrm{Enc}_{\mathrm{EF}_1 K}$ and the Perfect Cipher which runs this way.

1. Construct a sequence $A_1, \ldots, A_4$ by querying the oracle ($A_1 = O(|0|_{64})$ and $A_{i+1} = O(A_i)$).
2. Construct a sequence $B_1, \ldots, B_4$ like in S_2 or S_3.
3. With $S = A_1|\ldots|B_4$ simulate the attack $\mathcal{A}$ on Enc_S.

Its advantage is $|p^{\mathrm{Enc}_{S_3}} - p^{\mathrm{Enc}_{S_2}}|$ and can be upper bounded thanks to Hypothesis 14. Let $c + c_0$ be the cost of this attack. The c_0 overhead is obviously negligible. From our Hypothesis, we have

$$\epsilon \leq \epsilon' + 2h(c + c_0) + |p^{\mathrm{Enc}_{S_2}} - p^{\mathrm{Enc}_{S_1}}|.$$

Let us now prove that for any (unlimited) distinguisher between S_1 and S_2 the advantage is less than 43.2^{-128} which completes the proof.

When fed with a 1024-bit sequence s, such a distinguisher gives a random output which is 1 with probability $f(s)$ for a given function f. The advantage is

$$\mathrm{Adv} = \left| \sum_s f(s) \left(\Pr_{S_1}[s] - \Pr_{S_2}[s] \right) \right| \leq \sum_s \left| 2^{-1024} - \Pr_{S_2}[s] \right|.$$

Let

$$s = a_1|b_1|\ldots|a_4|b_4$$

and $a_0 = b_0 = |0|_{64}$. When $a_i \neq a_{i+j}$ and $b_i \neq b_{i+j}$ for all $0 \leq i < 4$ and $0 < j < 4$, the difference probability is equal to

$$\prod_{j=0}^{3} \left(\frac{1}{2^{128} - j} \right)^2 - 2^{-1024}$$

which is less than $13.2^{-128-1024}$. Thus

$$\mathrm{Adv} \leq 13.2^{-128} + \sum_{\text{other } s} \left(2^{-1024} + \Pr_{S_2}[s] \right).$$

The number of "other s" is less than $24 \times 2^{1024-128}$ so

$$\mathrm{Adv} \leq 37.2^{-128} + \Pr_{S_2}[\text{other } s].$$

Since F and G are permutations, if we have some $a_i \neq a_{i+j}$ or $b_i \neq b_{i+j}$ equation, we must have one with $i = 0$. Let J be the length of the orbit of $|0|_{128}$ with the permutation F. We thus have

$$\mathrm{Adv} \leq 37.2^{-128} + 2 \Pr_{F}[1 \leq J \leq 3].$$

Thus $\mathrm{Adv} \leq 43.2^{-128}$. $\square$

Workshop Compilers$^{\text{TM}}$, SunOS$^{\text{TM}}$ and JAVA$^{\text{TM}}$ are registered trademarks of Sun Microsystems.
UNIX$^{\text{TM}}$ is a registered trademark of Open Group.
Pentium$^{\text{TM}}$ and Intel$^{\text{TM}}$ are registered trademarks of Intel Corporation.
DEC$^{\text{TM}}$ and AXP$^{\text{TM}}$ are registered trademarks of Digital Equipment Corporation.
SPARC$^{\text{TM}}$, Ultrasparc$^{\text{TM}}$ and Solaris$^{\text{TM}}$ are registered trademarks of Sparc International, Inc.
Motorola$^{\text{TM}}$ is a registered trademark of Motorola Inc.
Windows NT$^{\text{TM}}$ and Visual C++$^{\text{TM}}$ are registered trademarks of Microsoft.