# DFC Update

Olivier Baudron[1], Henri Gilbert[2], Louis Granboulan[1], Helena Handschuh[3], Robert Harley[4], Antoine Joux[5], Phong Nguyen[1], Fabrice Noilhan[6], David Pointcheval[1], Thomas Pornin[1], Guillaume Poupard[1], Jacques Stern[1], and Serge Vaudenay[1]

[1] Ecole Normale Supérieure – CNRS
[2] France Telecom
[3] Gemplus – ENST
[4] INRIA
[5] SCSSI
[6] Université d'Orsay – LRI

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

**Abstract** This document reports an update of DFC. We answer to a question about the rationale for the CP Confusion Permutation. We give new implementation results for DFC. In particular we present an impressively fast implementation which takes 323 cycles on Compaq's 21164 Alpha microprocessor. On the new 21264 we expect to reach software encryption rates over 500 Mbps. We also discuss making DFC scalable to allow the block size or the number of rounds, in the encryption or key schedule, to be varied. Finally, we describe how DFC may be subject to slight change in its key schedule in order to fix a minor drawback noticed by Coppersmith.

Since DFC was proposed in [5, 6], several issues were raised and several advances made. The present report addresses the following.

1. The rationale for CP was needed.
2. New implementation results.
3. Criticisms raised on the number of rounds.
4. Weak keys were identified.

## 1 Rationale of the Design of CP

During the first AES workshop, the question of the rationale for the CP Confusion Permutation was raised. The somewhat provocative answer given was that CP could be replaced by anything else (even the identity function) as far as the decorrelation analysis is concerned but, as discussed in the next section. This is not enough to guarantee real security though, and CP actually plays some role for the security. Decorrelation provides provable security against some classes of attacks, and the frontier between these attacks and other potential ones which might be covered by this theory is quite sharp. Conservative designs use heuristic security for which the frontier is usually smooth. We believe that we should use both approaches: combining decorrelation designs which provably protect against some classes of attacks, with conservative design which increases the difficulty of other attacks. This is the purpose of the CP function.

In the DFC design we wanted to mix several simple arithmetic operations over mixed algebraic structures. We chose a CP which combines XOR and addition (as is proposed in — for example — RC5 [7]). We also introduced some non-algebraic randomness by means of a look-up table, and we wanted that table to be limited to 256 bytes in order to minimize memory requirements (for smart cards). We did not want to introduce rotations which are painful on the 6805 as well as on Alpha or (Ultra)Sparc. We also used random translations by constants. The original report [6] gives some rationale for the choice of the constants.

## 2 New Implementations

We have been optimizing our implementations provided in the AES CD-ROM2. Using programming tricks to help the compilers to produce optimized code and a fast carry scheme, we basically got a 30% speed improvement for our 64-bit C implementation on Pentium Pro (1262 cycles), UltraSparc (910 cycles) and Alpha processors (565 cycles). Harley made an implementation of DFC on ARM which encrypts within 710 cycles (C language plus asm opcode) or 560 cycles (assembly code).

We have optimized the Java implementation as well, using the JDK-2 and "just-in-time" compilation.

Harley wrote an impressive implementation of DFC on the Alpha architecture. This implementation is in ANSI-C but requires that `long` types be 64-bit integers. It uses the Alpha assembly code instruction `umulh` if it is available (this instruction returns the 64 most significant bits of a 64x64 bit unsigned multiplication) and otherwise falls back to generic replacement code for the multiplication.

On 21164a microprocessors, we got an encryption within 323 cycles for $m = 128$ and $r = 8$. We measured 232 cycles on a prototype for the new generation 21264. A pure C implementation (not using `umulh`) encrypts one block within 526 cycles on the 21164a. This implementation is given in Appendix.

All these results are reported in Table 1.

## 3 Possible Variations on DFC

In order to address several issues on DFC (namely, the low number of rounds and key scheduling issues), we discuss possible adjustments to DFC. The present report does not aim to propose a specific variant but to show that the known problems could easily be fixed.

| platform | language | compiler | programmer | enc. time |
|---|---|---|---|---|
| Alpha 21164a | C+asm | cc.alt | Harley | 323 |
| Alpha 21164a | ANSI C | cc.alt | Harley | 526 |
| Alpha 21264 | C+asm | cc.alt | Harley | 232 |
| Pentium Pro | asm | nasm | Behr | 550 |
| Pentium Pro | asm | masm | McGougan | 482 |
| Pentium Pro | ANSI C | gcc | Noilhan | 1262 |
| UltraSparc | C+asm | SWC 5.0 | Noilhan | 910 |
| UltraSparc | Java | JDK 1.2 | Noilhan | 11350 |
| ARM | C+asm | gcc | Harley | 710 |
| ARM | asm | gas | Harley | 560 |
| 6805 <200B RAM | asm | | Poupard | 35000 |
| 6805 <100B RAM | asm | | Poupard | 200000 |

**Table1.** Best known implementations of DFC on various platforms. The timing are given in cycles for one block encryption. The Java implementations used just-in-time compilers.

### 3.1 New Parameters

Several of our colleagues criticized DFC because of its low number of rounds. Actually, our paradigm was to trust our security results and commit to them by not adding too many extra rounds. We actually believe that due to the strength provided by the decorrelation approach, the security increases with the number of rounds faster than with regular designs. We are however concerned that DFC should gain the confidence of a general audience not familiar with the subtleties of decorrelation theory. For this we suggest allowing the number of rounds $r$ to be increased, but still consider $r = 8$ as our nominal choice. Biham [3] suggested using $r = 9$ in order to compare DFC to other ciphers with similar margins of security. In Table 2 we reported Biham's suggested number of rounds for the AES candidates and the best known implementation timing on Alpha 21164a scaled accordingly. Mars becomes the fastest algorithm, followed by DFC.

| cipher | # rounds | timing | suggested | new timing | cipher | # rounds | timing | suggested | new timing |
|---|---|---|---|---|---|---|---|---|---|
| Cast256 | 48 | 749 | 40 | 624 | Magenta | 6 | 5074 | 10 | 8457 |
| Crypton | 12 | 499 | 11 | 457 | Mars | 32 | 507 | 20 | 317 |
| Deal | 6 | 2752 | 9 | 4128 | RC6 | 20 | 559 | 20 | 559 |
| DFC | 8 | 323 | 9 | 363 | Rijndael | 10 | 490 | 8 | 392 |
| E2 | 12 | 587 | 10 | 489 | Safer+ | 8 | 1502 | 7 | 1314 |
| Frog | 8 | 2752 | ? | | Serpent | 32 | 998 | 17 | 530 |
| HPC | 8 | 402 | ? | | Twofish | 16 | 490 | 12 | 368 |
| Loki97 | 16 | 2356 | 36 | 5301 | | | | | |

**Table2.** Biham's suggested round numbers given at the Asiacrypt'98 Conference (taken from [3]). We added the encryption time (in clock cycles) of the best software implementation on Alpha 21164a (see [2]) and the new timing if we change the algorithm accordingly.

In this report, we also propose making DFC scalable, which may be useful for some applications. We thus provide a scalable size $m$ of the message blocks. The AES requires a block-size of $m = 128$. If one wishes to reduce to reduce $m$, it is necessary to change the prime number and the CP Confusion Permutation. We simply recommend using the smallest prime number $p$ which is greater than $2^{\frac{m}{2}}$.

| $m$ | $p$ |
|---|---|
| 32 | $2^{16}+1$ |
| 64 | $2^{32}+15$ |
| 96 | $2^{48}+21$ |
| 128 | $2^{64}+13$ |

Concerning the RT Round Table, our design criterion was to limit the look-up table to 256 bytes to fit into the ROM of smart cards. This is why we chose a 6-bit input and a 32-bit output. In general, we propose using a 6-bit input and a $\frac{m}{4}$-bit output. (We thus use an amount of randomness proportional to $m$ and limited to 256 bytes when $m = 128$.) This way, we can define a version of DFC dedicated to 32-bit microprocessors (but with 64-bit blocks). We can also propose a toy cipher with a block-size of 32 bits.

We shall refer to $k$ as the key length. The AES requires $k = 128$, $k = 192$ and $k = 256$. Our original design already tolerated any $k$ within the range $[0, 2m]$.

We add an extra parameter $s$ which consists of the number of rounds used in the key scheduling algorithm per round in the encryption. Our nominal choice consists of $s = 4$. The key setup over encryption time ratio will be equal to $s$, which is thus scalable.

Introducing these parameters does not affect the original design: choosing the corresponding parameters makes it totally compatible.

### 3.2 A New Key Schedule

The key schedule of DFC has two (minor) drawbacks. First of all, Coppersmith noted that if the internal $\mathrm{RK}_2$ Round Key happens to be zero (which holds with probability $2^{-128}$), then the symmetries in the key schedule make the whole encryption scheme become the identity function (in the sense that the encryption of any message $x$ will be $x$ itself)! Second the first round key, $\mathrm{RK}_1$, depends on only half of the secret key which may lead to an exhaustive key search attack on the first round key. If desired, these drawbacks could easily be fixed with minor changes to the key schedule, *e.g.* by changing $\mathrm{EF}_i(K)$ from one round to the other, and by making $\mathrm{RK}_0$ depend on $K$ instead of being 0. We do not propose making such an adjustment at this stage but, if requested, could do so during the second round of the AES process.

## 4 Security Results

We state the security results in terms of the new parameters $(m, k, r, s)$ and the prime number $p$. We recall that the security results consist of, first, theoretical results for an ideal DFC in which the $\mathrm{RK}_i$ sequence is uniformly distributed (we will call $\mathrm{DFC}^*(m,r)$ this ideal algorithm which does not depend on $k$ or $s$) and second, some practical results obtained by relating the theoretical results back to the real DFC algorithm.

**Theorem 1.** *The permutation* $\mathrm{DFC}^*(m,r)$ *admits a decorrelation bias of order two which is such that*

$$\mathrm{DecP}^2(\mathrm{DFC}^*(m,r)) \leq \left(\epsilon + 3\epsilon^2 + \epsilon^3 + 2^{3-\frac{m}{2}}\right)^{\lfloor\frac{r}{3}\rfloor} \tag{1}$$

*where* $\epsilon = \mathrm{DecF}^2(\mathrm{RF})$ *is the pairwise decorrelation bias of the round function which is such that*

$$\mathrm{DecF}^2(\mathrm{RF}) \leq 2\left(\left(\frac{p}{2^{\frac{m}{2}}}\right)^2 - 1\right) \tag{2}$$

*where* $p$ *is the smallest prime number greater than* $2^{\frac{m}{2}}$.

(We consider here the decorrelation with the $|||.|||_\infty$ norm as explained in [8–11].) Thus if we let $p = 2^{\frac{m}{2}}(1+\delta)$ we can approximate the decorrelation bias upper bound by

$$\left(4\delta + 2^{3-\frac{m}{2}}\right)^{\lfloor\frac{r}{3}\rfloor}. \tag{3}$$

This shows that the pairwise decorrelation bias is negligible compared to $2^{-m}$ if $r \geq 9$. We believe that $r = 8$ is sufficient. For $m = 128$, we have $\delta = 13 \times 2^{-64}$ and we get back the original design of DFC:

$$\mathrm{DecP}^2(\mathrm{DFC}^*(128,r)) \leq 2^{-58\lfloor\frac{r}{3}\rfloor}. \tag{4}$$

From decorrelation theory we know that the average complexity of differential cryptanalysis (over the distribution of the keys) needs at least to be on the order of $1/\mathrm{DecP}^2$, as for linear cryptanalysis (from an asymptotic bound). Similarly, the average complexity of any known plaintext which comes from an iterated attack of order one needs to be at least on the order of $1/\sqrt{\mathrm{DecP}^2}$. (In these results, the phrase "on the order of" means equality to within a constant factor depending only on the expected probability of success. For a probability of 50%, these constants are greater than 1/10.) More precisely we recall the following results taken from [8–11].

**Theorem 2.** *For any differential distinguisher with complexity* $n$ *against a permutation over a space of* $2^m$ *elements and with a pairwise decorrelation bias of* $\mathrm{DecP}^2$, *the advantage* Adv *is such that*

$$\mathrm{Adv} \leq \frac{n}{2}\mathrm{DecP}^2 + \frac{n}{2^m - 1}. \tag{5}$$

*Similarly, for any linear distinguisher we have*

$$\lim_{n\to+\infty} \frac{\mathrm{Adv}}{n^{\frac{1}{3}}} \leq 9.3\left(2\mathrm{DecP}^2 + \frac{1}{2^m - 1}\right)^{\frac{1}{3}}. \tag{6}$$

*For any known plaintext iterated distinguisher with order 1 we have*

$$\mathrm{Adv} \leq 3\left(\left(\frac{9}{2}2^{-m}+\frac{3}{2}\mathrm{DecP}^2\right)n^2\right)^{\frac{1}{3}}+\frac{n}{2}\mathrm{DecP}^2. \qquad (7)$$

(For an accurate formalization of differential distinguishers, linear distinguishers and iterated distinguishers, see [9–11].) When $\mathrm{DecP}^2$ is negligible compared to $2^{-m}$, decorrelation does not provide any more security with these bounds. It is thus useless (when considering these results) to add too many rounds after $r \geq 8$.

For example with $m = 128$ and $r = 6$ (the nominal choice of DFC reduced to 6 rounds instead of 8) we get $\mathrm{Adv} \leq n.2^{-117}$, $\mathrm{Adv} \leq (n.2^{-105})^{\frac{1}{3}}$ and $\mathrm{Adv} \leq (n^2.2^{-110})^{\frac{1}{3}}$ for differential distinguishers, linear distinguishers and iterated attacks of order 1 against $\mathrm{DFC}^*(128,6)$ respectively. This formally proves that DFC* is immune against these attacks if we use it less than $2^{55}$ times with the same key.

Since DFC has a key scheduling algorithm, we need some extra result to transform the security results on DFC* to DFC. We let $\mathcal{D}(m,k,r,s)$ denote the distribution of $(\mathrm{RK}_1,\ldots,\mathrm{RK}_r)$ spanned by the key scheduling algorithm of $\mathrm{DFC}(m,k,r,s)$ when $K$ is a uniformly distributed $k$-bit key, and we let $\mathcal{D}^*$ denote the uniform distribution over $rm$-bit sequences. DFC* relies on the $\mathcal{D}^*$ distribution, but DFC uses the $\mathcal{D}$ one.

**Definition 3.** *We consider a probabilistic Turing machine $\mathcal{A}$ limited to $t$ instructions and which is fed by a $rm$-bit random string of distribution $D$ and must output one bit. We consider the advantage of distinguishing $\mathcal{D}(m,k,r,s)$ from $\mathcal{D}^*$ defined by*

$$\mathrm{Adv}_t^{\mathcal{A}}(m,k,r,s) = |\Pr[\mathcal{A}(\mathcal{D}(m,k,r,s))=1]-\Pr[\mathcal{A}(\mathcal{D}^*)=1]|.$$

*We let $H_t(m,k,r,s)$ denote the maximal possible advantage with these parameters.*

This function $H_t$ enables us to state the following "meta-theorem".

**Theorem 4.** *Let $t_k$ be the minimal complexity of the key scheduling algorithm and $t_e$ be the minimal complexity of the encryption algorithm. If for some class $\mathcal{C}$ of distinguishers the advantage of distinguishing $\mathrm{DFC}^*(m,r)$ from a random permutation is limited to $f(t,n)$ where $t$ is the complexity and $n$ the number of oracle calls, then the advantage of distinguishing $\mathrm{DFC}(m,k,r,s)$ from a truly random permutation is limited to $H_{t+nt_e+t_k}(m,k,r,s)+f(t,n)$ in this class of attacks.*

When considering the structure of DFC, we have $t_k \approx st_e$. Therefore, assuming that we cannot efficiently distinguish $\mathcal{D}(m,k,r,s)$ from $\mathcal{D}^*$ within a complexity limited to $xt_e$ (*i.e.* that $H_{xt_e}(m,k,r,s)$ is negligible), all results for distinguishers on DFC* with a limited number of oracle calls of $n = \frac{x}{2} - s$ and a limited complexity of $t = \frac{x}{2}t_e$ are applicable on DFC.

**Corollary 5.** *With the above notations, let* $\epsilon = H_{xt_e}(128, 128, 8, 4)$. *With the approximation that* $t_k \approx 4t_e$, *we have the following results.*

1. $\mathrm{Adv} \leq x.2^{-118} + \epsilon$ *for any differential distinguisher,*
2. $\mathrm{Adv} \leq (x.2^{-106})^{\frac{1}{3}} + \epsilon$ *for any linear distinguisher,*
3. $\mathrm{Adv} \leq (x^2.2^{-112})^{\frac{1}{3}} + \epsilon$ *for any iterated attack of order 1*

*for a number of queries of* $\frac{x}{2} - 4$ *against DFC reduced to 6 rounds instead of 8.*

Hence if $\epsilon$ is negligible, $\mathrm{DFC}(128, 128, 8, 4)$ is provably secure against these attacks for a number of uses which is less than $2^{55}$.

We can thus propose two kinds of challenge to the research community:

1. distinguishing one $\mathrm{RK}_1, \ldots, \mathrm{RK}_r$ sequence generated by $\mathrm{DFC}(128, 128, 8, 4)$ from a uniformly distributed sequence with a complexity negligible compared to $2^{55}$ DFC encryptions,
2. distinguishing $\mathrm{DFC}^*(128, 6)$ from a random permutation with a complexity negligible compared to $2^{55}$ DFC encryptions and a number of oracle calls negligible compared to $2^{55}$.

We proved that no differential, linear or iterated attack of order 1 can meet these challenges. We conjecture that $m \geq 128$, $k \geq 128$, $r \geq 8$ and $s \geq 4$ are safe in the sense of these challenges with the technology of the AES application period.


## Acknowledgements

We wish to thank Don Coppersmith for mentioning the weak keys of DFC. We also thank Brian Gladman, Dominik Behr and Danjel McGougan for their impressive implementations of DFC as well as Eli Biham for valuable discussions, Kazumaro Aoki and Doug Whiting for their comments and the NIST for organizing this exciting work.


## References


1. K. Almquist. AES Candidate Performance on the Alpha 21164 Processor (version 1). Published in the `sci.crypt` Usenet Newsgroup. 23rd of December, 1998.
2. O. Baudron, H. Gilbert, L. Granboulan, H. Handschuh, A. Joux, P. Nguyen, F. Noilhan, D. Pointcheval, T. Pornin, G. Poupard, J. Stern, S. Vaudenay. Report on the AES Candidates. Submitted.
3. E. Biham. Invited talk given at the Asiacrypt'98 Conference. Slides available on `http://www.cs.technion.ac.il/~biham/`
4. H. Feistel. Cryptography and Computer Privacy. *Scientific American*, vol. 228, pp. 15–23, 1973.
5. H. Gilbert, M. Girault, P. Hoogvorst, F. Noilhan, T. Pornin, G. Poupard, J. Stern, S. Vaudenay. Decorrelated Fast Cipher: an AES Candidate. (Extended Abstract.) In *Proceedings from the First Advanced Encryption Standard Candidate Conference*, National Institute of Standards and Technology (NIST), August 1998.

6. H. Gilbert, M. Girault, P. Hoogvorst, F. Noilhan, T. Pornin, G. Poupard, J. Stern, S. Vaudenay. Decorrelated Fast Cipher: an AES Candidate. Submitted to the Advanced Encryption Standard process. In *CD-ROM "AES CD-1: Documentation"*, National Institute of Standards and Technology (NIST), August 1998.
7. R. L. Rivest. The RC5 Encryption Algorithm. In *Fast Software Encryption*, Leuven, Belgium, Lectures Notes in Computer Science 1008, pp. 86–96, Springer-Verlag, 1995.
8. 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.
9. S. Vaudenay. Feistel Ciphers with $L_2$-Decorrelation. To appear in SAC'98, LNCS.
10. S. Vaudenay. The Decorrelation Technique Home-Page. URL:`http://www.dmi.ens.fr/~vaudenay/decorrelation.html`
11. S. Vaudenay *Vers une Théorie du Chiffrement Symétrique*, Dissertation for the diploma of "habilitation to supervise research" from the University of Paris 7, Technical Report LIENS-98-15 of the Laboratoire d'Informatique de l'Ecole Normale Supérieure, 1998.


## A Implementations of DFC on Alpha

The following implementation of DFC in ANSI-C detects if the Alpha ASM instruction `umulh` is available and uses it if possible.

```
/* > dfc.c
 * Purpose: Implement "Decorrelated Fast Cipher" from ENS for AES.
 * Copyright: (c) Robert J. Harley, 16-July-1998
 * Contact: Robert.Harley@inria.fr
 * Legalese: This code is subject to the GNU General Public License (v2).
 */

/* Compile with an incantation like one of these:
 *   egcs -O3 -freg-struct-return -Wall dfc.c -o dfc
 *   cc.alt -O5 -tune host -std1 dfc.c -inline all -o dfc
 */

#ifndef __alpha
#error Take out the rpcc stuff to use this on 64-bit chips other than Alpha.
#endif


/*== #includes ==============================================================*/

/* ANSI includes */
#include <stdio.h>
#include <assert.h>

/* System-specific includes */
#if defined(__DECC) && defined(__alpha)
#include <c_asm.h>
#endif


/*== Types ==================================================================*/

typedef unsigned int u32;
typedef signed long s64;
typedef unsigned long u64;

typedef struct { u64 hi, lo; } u128;
```

```
/*== #defines ==============================================================*/

#ifdef __GNUC__
#define INLINE inline
#elif defined(__DECC)
#define INLINE __inline
#else
#define INLINE
#endif


/*== Function declarations =================================================*/

static INLINE u64 rf(u128 rk, u64 x);
static u128 dfc(u128 pt, const u128 *ek);

static INLINE u64 rpcc(void);

int main(int argc, char *argv[]);


/*== Function definitions ==================================================*/

/*-- rf --------------------------------------------------------------------*/

/* Round function. */
static INLINE u64 rf(u128 rk, u64 x) {
  u64 a, b, h, l, t, v;
  const u64 kc = 0xEB64749AUL, kd = 0x86D1BF275B9B241DUL;
  static const u32 rt[64] =
  { 0xB7E15162U, 0x8AED2A6AU, 0xBF715880U, 0x9CF4F3C7U
  , 0x62E7160FU, 0x38B4DA56U, 0xA784D904U, 0x5190CFEFU
  , 0x324E7738U, 0x926CFBE5U, 0xF4BF8D8DU, 0x8C31D763U
  , 0xDA06C80AU, 0xBB1185EBU, 0x4F7C7B57U, 0x57F59584U
  , 0x90CFD47DU, 0x7C19BB42U, 0x158D9554U, 0xF7B46BCEU
  , 0xD55C4D79U, 0xFD5F24D6U, 0x613C31C3U, 0x839A2DDFU
  , 0x8A9A276BU, 0xCFBFA1C8U, 0x77C56284U, 0xDAB79CD4U
  , 0xC2B3293DU, 0x20E9E5EAU, 0xF02AC60AU, 0xCC93ED87U
  , 0x4422A52EU, 0xCB238FEEU, 0xE5AB6ADDU, 0x835FD1A0U
  , 0x753D0A8FU, 0x78E537D2U, 0xB95BB79DU, 0x8DCAEC64U
  , 0x2C1E9F23U, 0xB829B5C2U, 0x780BF387U, 0x37DF8BB3U
  , 0x00D01334U, 0xA0D0BD86U, 0x45CBFA73U, 0xA6160FFEU
  , 0x393C48CBU, 0xBBCA060FU, 0x0FF8EC6DU, 0x31BEB5CCU
  , 0xEED7F2F0U, 0xBB088017U, 0x163BC60DU, 0xF45A0ECBU
  , 0x1BCD289BU, 0x06CBBFEAU, 0x21AD08E1U, 0x847F3F73U
  , 0x78D56CEDU, 0x94640D6EU, 0xF0D3D37BU, 0xE67008E1U
  };

  a = rk.hi;


  /** Multiply x by half of rk: 64 x 64 -> 128 bits. **/

#if defined(__GNUC__) && defined(__alpha)

  asm volatile ("umulh %1,%2,%0" : "=r" (h) : "%rJ" (a), "rI" (x));
  l = a*x;

#elif defined(__DECC) && defined(__alpha)

  h = asm("umulh %a0, %a1, %v0", a, x);
  l = a*x;

#else

  /* Do it the hard way! */
```

```
  { u64 ah, al, xh, xl, t, u;
    const u64 mask = 0xFFFFFFFFUL;

    ah = a>>32; al = a & mask;
    xh = x>>32; xl = x & mask;
    h = ah*xh; l = al*xl; t = ah*xl; u = al*xh;
    t += u;
    h += t>>32; h += (u64)(t < u)<<32;
    t <<= 32; l += t; h += l < t;
  } /* end block */

#endif


  /** Add other half of rk and reduce modulo 2^64+13. **/

  b = rk.lo; b += 182UL; v = b < 182UL;
  h = ~h;
  v += h>>61; t = b+(h<<2); v += t < b;
  v += h>>62; b = t+(h<<3); v += b < t;
  b += h; v += b < h;
  b += l; v += b < l;
  v *= 13UL;
  t = b < v; b -= v;


  /** Confusion permutation. **/

#define CP ((b ^ (u64)rt[b>>58])<<32 | (b>>32 ^ kc)) + kd

  if (!t) return CP;
  b += 13UL; return CP;

#undef CP

} /* end rf */


/*-- dfc ------------------------------------------------------------------*/

/* Decorrelated Fast Cipher. */
static u128 dfc(u128 pt, const u128 *ek) {
  u64 r0, r1;
  u128 res;

  r0 = pt.hi; r1 = pt.lo;

  r0 ^= rf(ek[0], r1);
  r1 ^= rf(ek[1], r0);
  r0 ^= rf(ek[2], r1);
  r1 ^= rf(ek[3], r0);
  r0 ^= rf(ek[4], r1);
  r1 ^= rf(ek[5], r0);
  r0 ^= rf(ek[6], r1);
  r1 ^= rf(ek[7], r0);

  res.lo = r0; res.hi = r1;
  return res;
} /* end dfc */


/*-- rpcc -----------------------------------------------------------------*/

/* Read the Alpha's 64 bit process cycle counter. */
static INLINE u64 rpcc(void) {
  u64 cyc;

#if defined(__GNUC__) && defined(__alpha)
```

```
  asm volatile("rpcc %0" : "=r"(cyc) : );
#elif defined(__DECC) && defined(__alpha)
  cyc = asm("rpcc %v0");
#else
#error You should write an rpcc() here or remove cycle counting altogether.
#endif

  return cyc;
} /* end rpcc */


/*-- main -----------------------------------------------------------------*/

#define ITERATIONS (1000000UL)

#ifndef WHICH
#define WHICH 1
#endif

int main(int argc, char *argv[]) {
  u32 cyc;
  u64 c0, c1, dummy, i, tot;
  u128 t;

  /* Some test vectors. */
#if WHICH == 1
  static const u128 ek[8] = {
      { 0x127433b4c2ee4273UL, 0xe0ff3b8da7fddb11UL }
    , { 0x76a13ccf10836432UL, 0x5a6e7f7a9aa4676aUL }
    , { 0x7032b26204585fdbUL, 0xf293847015f51616UL }
    , { 0xd16f25e08ecb5a2aUL, 0x11457b2c4120e5e9UL }
    , { 0x272ceaa00e0eb86fUL, 0x8238748002bc2febUL }
    , { 0x711f3d8ce9beb919UL, 0x4e6fae2e59021d13UL }
    , { 0x8f06eeeb7c693c90UL, 0x626b4996dca97e9aUL }
    , { 0x84f3600638ea9802UL, 0x3b53d77f60ae2a93UL }
    }
  , pt = { 0x0001020304050607UL, 0x08090a0b0c0d0e0fUL }
  , ct = { 0x630420709e777ff6UL, 0xcb1c65231362c5e3UL }
  ;
#elif WHICH == 2
  static const u128 ek[8] = {
      { 0x496d91990be5df5cUL, 0x1db16891b4d94189UL }
    , { 0xfe338d5835695c9eUL, 0xe6965e1e1aa22ad8UL }
    , { 0x736ccee6099b8943UL, 0x3a2e390cdb84b6ffUL }
    , { 0x979d063435bd234bUL, 0xc0267e81d0868d26UL }
    , { 0xb9ccb771164dfc63UL, 0xa95a41963c2fdb84UL }
    , { 0x73996b00d23b5b92UL, 0x81fa21815c0ad0aeUL }
    , { 0x13af53a7896725f2UL, 0xc339e4ae2d7e16dfUL }
    , { 0xdac0f37721ffcefaUL, 0x37b2f77af0751899UL }
    }
  , pt = { 0x0000000000000000UL, 0x0000000000000000UL }
  , ct = { 0xbb46bb6ac0093c1dUL, 0xf567576616077eefUL }
  ;
#elif WHICH == 3
  static const u128 ek[8] = {
      { 0x123c32a554882c38UL, 0xde7e2e63b7cc7420UL }
    , { 0xa6f556e8ceabf7f2UL, 0x1b6c467267a615b3UL }
    , { 0xa6d0ac27ae9f2ec3UL, 0x0d4e6c5a0e6f9226UL }
    , { 0x828fea1d0dfd4937UL, 0xc89f731ad9963921UL }
    , { 0x039819d387c111f0UL, 0x5093fce0801823fcUL }
    , { 0x9eeb4eb30acf2c10UL, 0xca5f11a78b6cff2cUL }
    , { 0x60bbe5c300ec4139UL, 0x22cc693e17bc25deUL }
    , { 0x21e02b6483dca14dUL, 0x6976dd7e25342eb7UL }
    }
  , pt = { 0x0001020304050607UL, 0x08090a0b0c0d0e0fUL }
  , ct = { 0x7072a955ccedbc49UL, 0x03417213c051f7baUL }
  ;
#elif WHICH == 4
```

```
  static const u128 ek[8] = {
      { 0xd8ade68bd5434eb3UL, 0x968d2c9a73400385UL }
    , { 0x06928796a82e7c3dUL, 0x683369bbee406c13UL }
    , { 0x8db229ecf795a21cUL, 0x8a684e68728a5d54UL }
    , { 0x5cb07984cbb70753UL, 0x17d981e1d781a156UL }
    , { 0xe6daa68c9f24de00UL, 0x9abfabf022724506UL }
    , { 0x0706976f97ddad95UL, 0xbf82eb7945f23b77UL }
    , { 0x928ef13fdf6b6756UL, 0xc36b289335a4683aUL }
    , { 0x830532c56b5d29e9UL, 0x3dbc5b8677cb0abdUL }
    }
  , pt = { 0x0000000000000000UL, 0x0000000000000000UL }
  , ct = { 0x6a6502105c23c62aUL, 0xb01e6b324dcdf664UL }
  ;
#elif WHICH == 5
  static const u128 ek[8] = {
      { 0xf3d66f92e400fdfeUL, 0xa3f015785f1345b5UL }
    , { 0x0715a4f70b90075dUL, 0xd015cea245182fe2UL }
    , { 0x3f7e31f08d676cdbUL, 0xb0c012bfc2c3416fUL }
    , { 0x6c08e2fd09001cccUL, 0x7682c0a9c2e4b2baUL }
    , { 0xf89252dac0ca6e2aUL, 0x0fe2bc91385be2ccUL }
    , { 0xd2e789c0f2d52094UL, 0xbc5f31bbe906de94UL }
    , { 0x38da18720dec3830UL, 0x0eb31504df20812fUL }
    , { 0x6c867cd75af52e02UL, 0xee267a822a44b4cfUL }
    }
  , pt = { 0x0001020304050607UL, 0x08090a0b0c0d0e0fUL }
  , ct = { 0x10fdbad10fa262f6UL, 0x4147a00f4306b11cUL }
  ;
#elif WHICH == 6
  static const u128 ek[8] = {
      { 0xce7ab59921286d96UL, 0x1f9597015e53c8faUL }
    , { 0x4f323cd9e4efbf05UL, 0x1967ca33c87a47d3UL }
    , { 0x05635b387bee64d8UL, 0x5cd9395ad4eeb8d9UL }
    , { 0xbefa386a7c4dfff9UL, 0xa5d00917d4c0fd50UL }
    , { 0xc1179957cddc2973UL, 0x6c5e4524c451c373UL }
    , { 0xdad7ce314826dbf7UL, 0xe10cf75dc9dc6d0cUL }
    , { 0xfdf7dc9f669f9bfcUL, 0x0d6780aa0f425efaUL }
    , { 0x5af8b46027c21aaeUL, 0x57e0104151775e23UL }
    }
  , pt = { 0x0000000000000000UL, 0x0000000000000000UL }
  , ct = { 0x68bf42dcf20d4ec7UL, 0xaa081c03822c6dbaUL }
  ;
#else
#error Please define WHICH = 1, 2, 3, 4, 5 or 6 to select a test vector.
#endif


  /** Check it works! **/
  t = dfc(pt, &ek[0]);
  printf("t = 0x%016lx:%016lx\n", t.hi, t.lo);
  assert(t.hi == ct.hi);
  assert(t.lo == ct.lo);


  /** Do a bunch of iterations for timing. **/
  tot = 0UL;
  dummy = 0UL;
  for (i = 0UL; i < ITERATIONS; ++i) {
    c0 = rpcc();
    t = dfc(t, &ek[0]);
    c1 = rpcc();

    cyc = (u32)(c1-c0+(c1>>32)-(c0>>32)); /* Number of cycles taken. */
    tot += (u64)cyc; /* Total so far. */

    dummy += t.hi; dummy += t.lo; /* Make sure optimiser keeps dfc()! */
  } /* end for */
```

```
  /** Print out stuff. **/
  printf( "Dummy result = %lu\n"
          "Average cycles = %g\n"
         , dummy, (double)tot/(double)ITERATIONS
        );

  /* OS      Compiler   Version Average cycles
   * ========================================
   * D.U.    cc         5.6     334
   * D.U.    cc.alt     5.7     323
   * D.U.    cc.alt2    5.8     363
   * Linux   gcc        2.8.1   373
   * Linux   egcs       1.1.1   382
   */


  return 0;
} /* end main */


/*== end of file dfc.c =====================================================*/
```