Fast & Portable DES encryption & decryption

> **Note**
>
> Below is the original README file from the descore.shar package, converted to ReST format.

---

des - fast & portable DES encryption & decryption.

Author's address: how@isl.stanford.edu

==>> To compile after untarring/unsharring, just `make` <<==

This package was designed with the following goals:

1. Highest possible encryption/decryption PERFORMANCE.
2. PORTABILITY to any byte-addressable host with a 32bit unsigned C type
3. Plug-compatible replacement for KERBEROS's low-level routines.

This second release includes a number of performance enhancements for register-starved machines. My discussions with Richard Outerbridge, 71755.204@compuserve.com, sparked a number of these enhancements.

To more rapidly understand the code in this package, inspect desSmallFips.i (created by typing `make`) BEFORE you tackle desCode.h. The latter is set up in a parameterized fashion so it can easily be modified by speed-daemon hackers in pursuit of that last microsecond. You will find it more illuminating to inspect one specific implementation, and then move on to the common abstract skeleton with this one in mind.

performance comparison to other available des code which i could compile on a SPARCStation 1 (cc -O4, gcc -O2):

this code (byte-order independent):

- 30us per encryption (options: 64k tables, no IP/FP)
- 33us per encryption (options: 64k tables, FIPS standard bit ordering)
- 45us per encryption (options: 2k tables, no IP/FP)
- 48us per encryption (options: 2k tables, FIPS standard bit ordering)
- 275us to set a new key (uses 1k of key tables)

> this has the quickest encryption/decryption routines i've seen. since i was interested in fast des filters rather than crypt(3) and password cracking, i haven't really bothered yet to speed up the key setting routine. also, i have no interest in re-implementing all the other junk in the mit kerberos des library, so i've just provided my routines with little stub interfaces so they can be used as drop-in replacements with mit's code or any of the mit- compatible packages below. (note that the first two timings above are highly variable because of cache effects).

kerberos des replacement from australia (version 1.95):

- 53us per encryption (uses 2k of tables)
- 96us to set a new key (uses 2.25k of key tables)

> so despite the author's inclusion of some of the performance improvements i had suggested to him, this package's encryption/decryption is still slower on the sparc and 68000. more specifically, 19-40% slower on the 68020 and 11-35% slower on the sparc, depending on the compiler; in full gory detail (ALT_ECB is a libdes variant):

compiler	machine	desCore libdes	ALT_ECB slower by
gcc 2.1 -O2	Sun 3/110	304 uS 369.5uS	461.8uS 22%
cc -O1	Sun 3/110	336 uS 436.6uS	399.3uS 19%

compiler	machine	desCore libdes	ALT_ECB slower by
cc -O2	Sun 3/110	360 uS 532.4uS	505.1uS 40%
cc -O4	Sun 3/110	365 uS 532.3uS	505.3uS 38%
gcc 2.1 -O2	Sun 4/50	48 uS 53.4uS	57.5uS 11%
cc -O2	Sun 4/50	48 uS 64.6uS	64.7uS 35%
cc -O4	Sun 4/50	48 uS 64.7uS	64.9uS 35%

(my time measurements are not as accurate as his).

the comments in my first release of desCore on version 1.92:

- 68us per encryption (uses 2k of tables)
- 96us to set a new key (uses 2.25k of key tables)

this is a very nice package which implements the most important of the optimizations which i did in my encryption routines. it's a bit weak on common low-level optimizations which is why it's 39%-106% slower. because he was interested in fast crypt(3) and password-cracking applications, he also used the same ideas to speed up the key-setting routines with impressive results. (at some point i may do the same in my package). he also implements the rest of the mit des library.

(code from eay@psych.psy.uq.oz.au via comp.sources.misc)

fast crypt(3) package from denmark:

the des routine here is buried inside a loop to do the crypt function and i didn't feel like ripping it out and measuring performance. his code takes 26 sparc instructions to compute one des iteration; above, Quick (64k) takes 21 and Small (2k) takes 37. he claims to use 280k of tables but the iteration calculation seems to use only 128k. his tables and code are machine independent.

(code from glad@daimi.aau.dk via alt.sources or comp.sources.misc)

swedish reimplementation of Kerberos des library

- 108us per encryption (uses 34k worth of tables)
- 134us to set a new key (uses 32k of key tables to get this speed!)

the tables used seem to be machine-independent; he seems to have included a lot of special case code so that, e.g., `long` loads can be used instead of 4 `char` loads when the machine's architecture allows it.

(code obtained from chalmers.se:pub/des)

crack 3.3c package from england:

as in crypt above, the des routine is buried in a loop. it's also very modified for crypt. his iteration code uses 16k of tables and appears to be slow.

(code obtained from aem@aber.ac.uk via alt.sources or comp.sources.misc)

`highly optimized` and tweaked Kerberos/Athena code (byte-order dependent):

- 165us per encryption (uses 6k worth of tables)
- 478us to set a new key (uses <1k of key tables)

so despite the comments in this code, it was possible to get faster code AND smaller tables, as well as making the tables machine-independent. (code obtained from prep.ai.mit.edu)

UC Berkeley code (depends on machine-endedness):

- 226us per encryption
- 10848us to set a new key

table sizes are unclear, but they don't look very small (code obtained from wuarchive.wustl.edu)

motivation and history

a while ago i wanted some des routines and the routines documented on sun's man pages either didn't exist or dumped core. i had heard of kerberos, and knew that it used des, so i figured i'd use its routines. but once i got it and looked at the code, it really set off a lot of pet peeves - it was too convoluted, the code had been written without taking advantage of the regular structure of operations such as IP, E, and FP (i.e. the author didn't sit down and think before coding), it was excessively slow, the author had attempted to clarify the code by adding MORE statements to make the data movement more `consistent` instead of simplifying his implementation and cutting down on all data movement (in particular, his use of L1, R1, L2, R2), and it was full of idiotic `tweaks` for particular machines which failed to deliver significant speedups but which did obfuscate everything. so i took the test data from his verification program and rewrote everything else.

a while later i ran across the great crypt(3) package mentioned above. the fact that this guy was computing 2 sboxes per table lookup rather than one (and using a MUCH larger table in the process) emboldened me to do the same - it was a trivial change from which i had been scared away by the larger table size. in his case he didn't realize you don't need to keep the working data in TWO forms, one for easy use of half the sboxes in indexing, the other for easy use of the other half; instead you can keep it in the form for the first half and use a simple rotate to get the other half. this means i have (almost) half the data manipulation and half the table size. in fairness though he might be encoding something particular to crypt(3) in his tables - i didn't check.

i'm glad that i implemented it the way i did, because this C version is portable (the ifdef's are performance enhancements) and it is faster than versions hand-written in assembly for the sparc!

porting notes

one thing i did not want to do was write an enormous mess which depended on endedness and other machine quirks, and which necessarily produced different code and different lookup tables for different machines. see the kerberos code for an example of what i didn't want to do; all their endedness-specific `optimizations` obfuscate the code and in the end were slower than a simpler machine independent approach. however, there are always some portability considerations of some kind, and i have included some options for varying numbers of register variables. perhaps some will still regard the result as a mess!

1. i assume everything is byte addressable, although i don't actually depend on the byte order, and that bytes are 8 bits. i assume word pointers can be freely cast to and from char pointers. note that 99% of C programs make these assumptions. i always use unsigned char's if the high bit could be set.
2. the typedef `word` means a 32 bit unsigned integral type. if `unsigned long` is not 32 bits, change the typedef in desCore.h. i assume sizeof(word) == 4 EVERYWHERE.

the (worst-case) cost of my NOT doing endedness-specific optimizations in the data loading and storing code surrounding the key iterations is less than 12%. also, there is the added benefit that the input and output work areas do not need to be word-aligned.

OPTIONAL performance optimizations

1. you should define one of `i386`, `vax`, `mc68000`, or `sparc`, whichever one is closest to the capabilities of your machine. see the start of desCode.h to see exactly what this selection implies. note that if you select the wrong one, the des code will still work; these are just performance tweaks.
2. for those with functional `asm` keywords: you should change the ROR and ROL macros to use machine rotate instructions if you have them. this will save 2 instructions and a temporary per use, or about 32 to 40 instructions per en/decryption.

   note that gcc is smart enough to translate the ROL/R macros into machine rotates!

these optimizations are all rather persnickety, yet with them you should be able to get performance equal to assembly-coding, except that:

1. with the lack of a bit rotate operator in C, rotates have to be synthesized from shifts. so access to `asm` will speed things up if your machine has rotates, as explained above in (3) (not necessary if you use gcc).
2. if your machine has less than 12 32-bit registers i doubt your compiler will generate good code.

   `i386` tries to configure the code for a 386 by only declaring 3 registers (it appears that gcc can use ebx, esi and edi to hold register variables). however, if you like assembly coding, the 386 does have 7 32-bit registers, and if you use ALL of them, use `scaled by 8` address modes with displacement and other tricks, you can get reasonable routines for DesQuickCore... with about 250 instructions apiece. For DesSmall... it will help to rearrange des_keymap, i.e., now the sbox # is the high part of the index and the 6 bits of data is the low part; it helps to exchange these.

   since i have no way to conveniently test it i have not provided my shoehorned 386 version. note that with this release of desCore, gcc is able to put everything in registers(!), and generate about 370 instructions apiece for the DesQuickCore... routines!

coding notes

the en/decryption routines each use 6 necessary register variables, with 4 being actively used at once during the inner iterations. if you don't have 4 register variables get a new machine. up to 8 more registers are used to hold constants in some configurations.

i assume that the use of a constant is more expensive than using a register:

a. additionally, i have tried to put the larger constants in registers. registering priority was by the following:

   - anything more than 12 bits (bad for RISC and CISC)
   - greater than 127 in value (can't use movq or byte immediate on CISC)
   - 9-127 (may not be able to use CISC shift immediate or add/sub quick),

    - 1-8 were never registered, being the cheapest constants.

b. the compiler may be too stupid to realize table and table+256 should be assigned to different constant registers and instead repetitively do the arithmetic, so i assign these to explicit `m` register variables when possible and helpful.

i assume that indexing is cheaper or equivalent to auto increment/decrement, where the index is 7 bits unsigned or smaller. this assumption is reversed for 68k and vax.

i assume that addresses can be cheaply formed from two registers, or from a register and a small constant. for the 68000, the `two registers and small offset` form is used sparingly. all index scaling is done explicitly - no hidden shifts by log2(sizeof).

the code is written so that even a dumb compiler should never need more than one hidden temporary, increasing the chance that everything will fit in the registers. KEEP THIS MORE SUBTLE POINT IN MIND IF YOU REWRITE ANYTHING.

(actually, there are some code fragments now which do require two temps, but fixing it would either break the structure of the macros or require declaring another temporary).

special efficient data format

bits are manipulated in this arrangement most of the time (S7 S5 S3 S1):

```
003130292827xxxx242322212019xxxx161514131211xxxx080706050403xxxx
```

(the x bits are still there, i'm just emphasizing where the S boxes are). bits are rotated left 4 when computing S6 S4 S2 S0:

```
282726252423xxxx201918171615xxxx121110090807xxxx040302010031xxxx
```

the rightmost two bits are usually cleared so the lower byte can be used as an index into an sbox mapping table. the next two x'd bits are set to various values to access different parts of the tables.

how to use the routines

datatypes:

pointer to 8 byte area of type DesData used to hold keys and input/output blocks to des.

pointer to 128 byte area of type DesKeys used to hold full 768-bit key. must be long-aligned.

DesQuickInit()

call this before using any other routine with `Quick` in its name. it generates the special 64k table these routines need.

DesQuickDone()

frees this table

DesMethod(m, k)

m points to a 128byte block, k points to an 8 byte des key which must have odd parity (or -1 is returned) and which must not be a (semi-)weak key (or -2 is returned). normally DesMethod() returns 0.

m is filled in from k so that when one of the routines below is called with m, the routine will act like standard des en/decryption with the key k. if you use DesMethod, you supply a standard 56bit key; however, if you fill in m yourself, you will get a 768bit key - but then it won't be standard. it's 768bits not 1024 because the least significant two bits of each byte are not used. note that these two bits will be set to magic constants which speed up the encryption/decryption on some machines. and yes, each byte controls a specific sbox during a specific iteration.

you really shouldn't use the 768bit format directly; i should provide a routine that converts 128 6-bit bytes (specified in S-box mapping order or something) into the right format for you. this would entail some byte concatenation and rotation.

Des{Small|Quick}{Fips|Core}{Encrypt|Decrypt}(d, m, s)

performs des on the 8 bytes at s into the 8 bytes at `d. (d,s: char *).`

uses m as a 768bit key as explained above.

the Encrypt|Decrypt choice is obvious.

Fips|Core determines whether a completely standard FIPS initial and final permutation is done; if not, then the data is loaded and stored in a nonstandard bit order (FIPS w/o IP/FP).

Fips slows down Quick by 10%, Small by 9%.

Small|Quick determines whether you use the normal routine or the crazy quick one which gobbles up 64k more of memory. Small is 50% slower then Quick, but Quick needs 32 times as much memory. Quick is included for programs that do nothing but DES, e.g., encryption filters, etc.

Getting it to compile on your machine

there are no machine-dependencies in the code (see porting), except perhaps the `now()` macro in desTest.c. ALL generated tables are machine independent. you should edit the Makefile with the appropriate optimization flags for your compiler (MAX optimization).

Speeding up kerberos (and/or its des library)

note that i have included a kerberos-compatible interface in desUtil.c through the functions des_key_sched() and des_ecb_encrypt(). to use these with kerberos or kerberos-compatible code put desCore.a ahead of the kerberos-compatible library on your linker's command line. you should not need to #include desCore.h; just include the header file provided with the kerberos library.

Other uses

the macros in desCode.h would be very useful for putting inline des functions in more complicated encryption routines.