Blowfish

cryptographic algorithm

implemented on the hp49g+ pocket calculator

by Michael Kuyumcu *(info@noemanetz.de)*, July 2007

version 0.6

Introduction

Blowfish is a patent-free and royalty-free encryption and decryption scheme. It was invented by Bruce Schneier in 1994 and has not been broken yet (at least successful attempts have not been published). If you want in-depth information on the algorithm, please visit *http://www.schneier.com/blowfish.html*

Blowfish is a symmetrical cipher, meaning that you use the same key for encryption and decryption. The key can have up to 56 extended ASCII characters (using the full ASCII range of codes 0 … 255). The fact that Blowfish is symmetrical means that you have to guard closely the key you are using for decryption and encryption.

This library for the hp49g+ pocket calculator implements Blowfish key **initialization**, the **encoding** of any amount of text (in a string) into a list of codes, and the **decoding** of such a list back into the original plain text. In addition, one routine is supplied that, when called, **resets** the subkey lists to some arbitrary initial value (the digits of π) so that no-one can use your subkeys to encode or decode messages. *It is important that you call the reset procedure when you're done with encoding and decoding!* I don't know whether the library works on hp pocket calculators other than the hp49g+.

The initialization (the core of which is written in SysRPL) takes quite a bit of time (around 6 minutes) but needs to be called only when you change your key. After that, you can encode text and decode relatively quickly using the precalculated subkey lists.

Installation

Transfer the binary directory file BF06.hp (indicating version 0.5) to your hp49g+ calculator to into any directory you please. If you don't have the BF06.hp file, you can extract two versions (one binary and one text) from inside this pdf using the Adobe Acrobat application. Now, inside the freshly transferred directory, you will find, amongst files named »Sbox«, »S0«, »S1«, »S2«, »S3«, and »Pbox«, the following six files:

Usage

»**BFini**« is the first thing you execute in order to initialize Blowfish for use with a particular key. Put your key as a string on the stack (it may be as long as you like although only the first 56 characters will be used as to not weaken the algorithm), press »BFini«, wait… wait… and finally the subkeys have been prepared for encoding and decoding. While the current initialization (6 minutes), written in SysRPL, features a great increase in speed over the original UserRPL version (29.5 minutes), it definitely could use more speed. I hope to be able to provide a faster version in the near future. *Example*: "MIchAEl" BFini

»**BFenc**« is the main encoding routine. Put the text you want to encode on the stack (level 1) and press »BFenc«. After a few seconds, the stack will show a list of hex string values. This represents the encoded message. You can transmit this sequence to another hp Blowfish user (who must know the key), and s/he can then decode the original message. Provided you have not changed the subkeys in the meantime, to decode, put the list of codes on the stack and press »**BFdec**«. Should you have pressed »Reset« in the meantime or encoded messages with other keys, you must re-initialize with the key and »BFini«, then put the codelist on level 1 and press »BFdec«. Returned will be the original message plus possible space padding bytes (which may have been appended in order to pad the string up to a multiple of 8 bytes).

Press »**Reset**« to prevent anyone from using your subkeys for encoding or decoding your messages. After Reset, when you want to encode or decode more messages, you must re-initialize with a key and »BFini«.

The »**BFe6s**« command may be used by more experienced hp users. It expects two hex strings (of at most 32 bits each) on the stack and encodes them using the Blowfish algorithm into another pair of hex strings on the stack. »**BFd6s**« decodes such a pair, using the same subkeys. Please note: after encoding, you must not execute »Reset« because that would scramble your subkeys and make decoding impossible. If you should happen to press »Reset« before decoding, you must re-initialize the subkeys again with »BFini« and the original key.

Following are the source listings for the mentioned modules and for the function f2 used in the algorithm.

Blowfish on the hp49g+

Description: this is the UserRPL function ***BFini***. It scrambles the subkeys using the entries from the original S-boxes (the digits of Pi minus the leading »3«) and the P-box entries using the key, too. Call it to make the current key the working key.

Stack: (key —)

```
%%HP: T(3)A(R)F(.);                          hp prologue, flag settings
\<< 0 0 \-> K Z MA                           the key is stored in 'K'
  \<< HEX                                    set hexadecimal mode
      Pbox 'P' STO                           Reinstate the original P-box entries
      Sbox 1 GET 'S0' STO                    … and the original 4 S-boxes
      Sbox 2 GET 'S1' STO
      Sbox 3 GET 'S2' STO
      Sbox 4 GET 'S3' STO

      1 'Z' STO
      K SIZE 56 MIN 'MA' STO

      1 18                                   There are 18 P-box entries to be
      FOR I                                  scrambled by logical ORing with
        # 0d                                 characters from the user key.

        0 3
        FOR R
          SLB
          # 4294967295d AND
          K Z Z SUB
          NUM R\->B
          OR
          Z 1 +
          DUP
          MA > IF THEN DROP 1 END
          'Z' STO
        NEXT
        P I GET XOR
        P SWAP I SWAP PUT 'P' STO
      NEXT
                                             The first #0d is the left 32 bits
      # 0d # 0d                              and the second the right 32 bits of
                                             data

      1 17                                   Now the P-box gets Blowfish-encoded
      FOR I
        BFe6s DUP2 P SWAP I 1 + SWAP PUT
        SWAP I SWAP PUT 'P' STO
      2 STEP

      1 255                                  And now the S-boxes
      FOR J
        BFe6s DUP2 S0 SWAP J 1 + SWAP PUT
        SWAP J SWAP PUT 'S0' STO
      2 STEP

      1 255
      FOR J
        BFe6s DUP2 S1 SWAP J 1 + SWAP PUT
        SWAP J SWAP PUT 'S1' STO
      2 STEP

      1 255
      FOR J
        BFe6s DUP2 S2 SWAP J 1 + SWAP PUT
        SWAP J SWAP PUT 'S2' STO
      2 STEP

      1 255
      FOR J
        BFe6s DUP2 S3 SWAP J 1 + SWAP PUT
        SWAP J SWAP PUT 'S3' STO
      2 STEP

      DEC                                    Restore decimal operating mode

      DROP2                                  and drop the intermediate data
  \>>
\>>
```

Blowfish on the hp49g+

Description: this is the UserRPL function ***BFenc***.
Using the subkeys in the four »s-boxes«, any string is encoded using the Blowfish algorithm.

Stack: (text as string — list of hex-string codes)

```
%%HP: T(3)A(R)F(.);                          hp prologue, flag settings
\<< 0 \-> T RO                               the plain text is stored in 'T'
  \<< HEX                                    set hexadecimal mode
      T SIZE 8 IDIV2                         Pad plaintext T if necessary
      DUP                                    so it has a length of a multiple of
                                             8 bytes afterwards. The 8 bytes are
      0 ==                                   necessary as the basic unit of data
      IF THEN DROP                           encryption by the Blowfish algorithm
      ELSE T SWAP 8 SWAP - 1 SWAP            as it encodes a pair of 32-bit values
           FOR I " " + NEXT                  simultaneously.
      'T' STO                                Save modified and padded text
      1 +
      END

      'RO' STO                               'RO' means encryption ROunds

      { }                                    this empty list collects the codes

      1 RO                                   Do 'RO' rounds of encryption
      FOR I
        T DUP HEAD NUM 16777216 *            Calculate a code for the next
        SWAP TAIL DUP HEAD NUM 65536 * ROT   4 bytes of the plaintext
        +                                    (using their ASCII values)
        SWAP TAIL DUP HEAD NUM 256 * ROT
        +
        SWAP TAIL DUP HEAD NUM
        ROT                                  and access the next 4 characters
        +                                    and do the same for them
        R\->B
        SWAP TAIL DUP HEAD NUM 16777216 *
        SWAP TAIL DUP HEAD NUM 65536 * ROT
        +
        SWAP TAIL DUP HEAD NUM 256 * ROT
        +
        SWAP TAIL DUP HEAD NUM ROT
        +

        SWAP                                 get the order right: left, right

        TAIL 'T' STO                         store the rest of the characters
        R\->B                                for later processing

        BFe6s                                call the 64-bit encryption routine

        2                                    make a 2-element mini-list from them
        \->LIST
        +                                    and put it into the collection list.
      NEXT
  \>>
\>>
```

Blowfish on the hp49g+

Description: this is the UserRPL function ***BFdec***.

Using the subkeys in the four »s-boxes«, any list of hex string codes is decrypted using the Blowfish algorithm.

Stack: (list of hex string codes — plaintext as string)

```
%%HP: T(3)A(R)F(.);                          hp prologue, flag settings
\<< "" \-> C T                               the code list is stored in 'C'
  \<< HEX                                    the plain text in 'T'.

      1 C SIZE 1 -                           Run through code list C
      FOR I                                  in steps of 2…
        C I GET                              retrieve first 32 bits of code
        C I 1 + GET                          retrieve second 32 bits of code

        BFd6s                                call the 64-bit decoding routine

        B\->R                                Transform the result into real
        SWAP                                 numbers for further arithmetic
        B\->R                                processing

        16777216 IDIV2                       Split up the big numbers into 4
        SWAP CHR T SWAP +                    single bytes, translate the bytes
        'T' STO                              into character codes and store
        65536 IDIV2 SWAP                     the growing plaintext in 'T'.
        CHR T SWAP +
        'T' STO
        256 IDIV2 SWAP
        CHR T SWAP +
        'T' STO
        CHR T SWAP +
        'T' STO

        16777216 IDIV2                       Access the next 4 characters
        SWAP CHR T SWAP +                    and do the same for them
        'T' STO
        65536 IDIV2 SWAP
        CHR T SWAP +
        'T' STO
        256 IDIV2 SWAP
        CHR T SWAP +
        'T' STO
        CHR T SWAP +
        'T' STO
     2                                       now access the next two codes
    STEP

    T                                        leave the plaintext on the stack
  \>>                                        as the result
\>>
```

Description: this is the UserRPL function ***Reset***.
To prevent tampering with the key-dependent subkeys, call this routine to restore the S-boxes and P-box entries to factory settings. DROP the comment afterwards.

Stack: (— "Blowfish reset.")

```
%%HP: T(3)A(R)F(.);                          hp prologue, flag settings
\<< Pbox 'P' STO                             Restore the P-box
    Sbox 1 GET ,S0‘ STO                      and the 4 S-boxes.
    Sbox 2 GET ,S1‘ STO
    Sbox 3 GET ,S2‘ STO
    Sbox 4 GET ,S3‘ STO

    "Blowfish reset."
\>>
```

Blowfish on the hp49g+

Description: this is the system-rpl function ***BFe6s***. This routine encodes a pair of HXS 32-bit values on the stack, data_l and data_r. It returns two scrambled values on the stack, the Blowfish-encoded data_l' and data_r'. data_l are the left 32-bits of a 64-bit datum, data_r the right 32 bits.

Stack: (data_l data_r — data_l' data_r') (all values HXS hex strings)

```
%%HP: T(3)A(R)F(.);                          hp prologue, flag settings
"!RPL                                        This is a System RPL program
!NO CODE                                     Don't produce code prologue for this
::                                           Start secondary
 CK2NOLASTWD                                 Error if no two arguments on stack
 CK&DISPATCH0                                Supported number types on stack are…
   #BB ::                                    HXS in level 1 and level 2

    BINT17 BINT1                             Start a LOOP from 1 to 16
    DO
      SWAP                                   Swap numbers on stack, data_l, data_r
      ID P INDEX@ NTHCOMPDROP                Get P-box element at loop index
      bitXOR                                 XOR with other data element
      SWAP                                   Swap both 32-bit data components
      OVER                                   Process the upper one
      ID F2                                  Call function f
      bitXOR                                 XOR with other data element
      SWAP                                   and swap again
    LOOP

    SWAP                                     Make the last swap undone
    ID P BINT17 NTHCOMPDROP                  access element P[17] of the P-box
    bitXOR                                   XOR 32-bit data element with data el.

    SWAP                                     Swap data_l and data_r once more
    ID P BINT18 NTHCOMPDROP                  access element P[18] of the P-box
    bitXOR                                   XOR 32-bit data element with data el.

    SWAP                                     restore correct order data_l' data_r'
    ;
;
@"
```

Blowfish on the hp49g+

Description: this is the user-rpl function ***BFd6s***. This routine decodes a pair of 32-bit values on the stack, data_l' and data_r'. It returns two cleartext values on the stack, the Blowfish-decoded data_l and data_r. data_l' are the left 32-bits of a 64-bit datum, data_r' the right 32 bits.

Stack: (HXS data_l' HXS data_r' — HXS data_l HXS data_r)

```
%%HP: T(3)A(R)F(.);                              hp prologue, flag settings
!RPL                                             System RPL program
!NO CODE                                         Produce no code prolog
::
 CK2NOLASTWD                                     Two arguments on the stack required
 CK&DISPATCH0                                    They must be of type
   #BB ::                                        Hex string and hex string

    BINT19 BINT3                                 Loop 16 times, from 18 downto 3
    DO
      SWAP                                       exhange data_l' and data_r'
      ID P BINT21 INDEX@  #- NTHCOMPDROP         access P-box element 21-loop index
      bitXOR                                     and XOR P element with data_l'
      SWAP                                       swap XOR(P[i], data_l') with data_r'
      OVER                                       duplicate the XOR result below
      ID F2                                      calculate f2(XOR result)
      bitXOR                                     XOR with data_r'
      SWAP                                       restore stack order: data_l', data_r'
    LOOP

    SWAP                                         undo last swap of data_l' & data_r'
    ID P BINT2 NTHCOMPDROP                       access P[2] of the P-box
    bitXOR                                       XOR 32-bit data element with P[2]

    SWAP                                         and swap once more data_l', data_r'
    ID P BINT1 NTHCOMPDROP                       access P[1] of the P-box
    bitXOR                                       XOR 32-bit data element with P[1]

    SWAP                                         restore correct order data_l data_r
    ;
;
@
```

Blowfish on the hp49g+

Description: this is the system-rpl function ***f2*** at the core of the Blowfish algorithm. From the subkeys in the four »s-boxes«, a kind of hash is calculated which is later used to scramble the left 32 bit of the data to be encoded and the right 32 bits. This version is over 3 x as fast as the User RPL version.

Stack: (32-bit HXS z — HXS f(z))

```
%%HP: T(3)A(R)F(.);                    hp prologue, flag settings
!RPL                                   system rpl language
!NO CODE                               not a code object
::
  CK1NOLASTWD                          Is there any argument?
  DUP                                  duplicate for successive splitting
  HXS 2 FF                             keep only the lowest 8 bits
  bitAND                               by masking out any other bits: a

  SWAP                                 get next-highest byte
  bitSRB                               extract the lowest byte only
  DUP                                  duplicate for successive splitting
  HXS 2 FF                             keep only the lowest 8 bits
  bitAND                               by masking out any other bits: b

  SWAP                                 get next-highest byte
  bitSRB                               extract the lowest byte only
  DUP                                  duplicate for successive splitting
  HXS 2 FF                             keep only the lowest 8 bits
  bitAND                               by masking out any other bits: c

  SWAP                                 get next-highest byte
  bitSRB                               extract the lowest byte only
  HXS 2 FF                             keep only the lowest 8 bits
  bitAND                               by masking out any other bits: d

  HXS>#                                make the hex string into a bint
  #1+                                  add 1 to the index a (list 1-based)
  ID S0                                ready for access the first s-box
  SWAP                                 get access index into place
  NTHCOMPDROP                          and retrieve s-box element

  SWAP                                 get the next index (b)
  HXS>#                                make the hex string into a bint
  #1+                                  add 1 to the index (list 1-based)
  ID S1                                ready for access the first s-box
  SWAP                                 get access index into place
  NTHCOMPDROP                          and retrieve s-box element

  bit+                                 add: stack now holds S0[a]+S1[b]
  HXS 8 FFFFFFFF                       keep only the lowest 32 bits
  bitAND                               by masking out any other bits

  SWAP                                 get the next index (c)
  HXS>#                                make it a bint number for indexing
  #1+                                  add 1 to the index (list 1-based)
  ID S2                                ready for access the first s-box
  SWAP                                 get access index into place
  NTHCOMPDROP                          and retrieve s-box element

  bitXOR                               XOR: stack now (S0[a]+S1[b])^S2[c]

  SWAP                                 get the last index (d)
  HXS>#                                make it a bint number for indexing
  #1+                                  add 1 to the index (list 1-based)
  ID S3                                ready for access the first s-box
  SWAP                                 get access index into place
  NTHCOMPDROP                          and retrieve s-box element

  bit+                                 add: (S0[a]+S1[b])^S2[c] + S3[d]
  HXS 8 FFFFFFFF                       keep only the lowest 32 bits
  bitAND                               by masking out any other bits
  ;
@
```