```c
/*
 * The MIT License (MIT)
 *
 * Copyright (c) 2018 Zhou Zhi Gang
 * Email: keorapetse.finger@yahoo.com
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */
#include <stdio.h>
#include <malloc.h>
#include "twofish.h"
#include "tables.h"

#define xor(g,r)    (g^r)                   /* Xor operation */
#define ror(g,n)    ((g>>n)|(g<<(32-n)))    /* Rotate right  */
#define rol(g,n)    ((g<<n)|(g>>(32-n)))    /* Rotate left   */
#define nxt(g,r)    (*(g+r))                /* Get next byte */
#define LITTILE_ENDIAN
#ifdef  LITTILE_ENDIAN
#define unpack(g,r) ((g>>(r*8))&0xff)                                   /* Extracts a byte from a word.  */
#define pack(g)     ((*(g))|(*(g+1)<<8)|(*(g+2)<<16)|(*(g+3)<<24))  /* Converts four byte to a word. */
#endif
#define pad_factor(g) \
        if (g<16)       n = 16; \
        else if (g<24)  n = 24; \
        else if (g<32)  n = 32;
#define rsm(i,a,b,c,d,e,f,g,h)  \
        gf(nxt(tf_key->k,r*8),a,0x14d)^gf(nxt(tf_key->k,r*8+1),b,0x14d)^\
        gf(nxt(tf_key->k,r*8+2),c,0x14d)^gf(nxt(tf_key->k,r*8+3),d,0x14d)^\
        gf(nxt(tf_key->k,r*8+4),e,0x14d)^gf(nxt(tf_key->k,r*8+5),f,0x14d)^\
        gf(nxt(tf_key->k,r*8+6),g,0x14d)^gf(nxt(tf_key->k,r*8+7),h,0x14d)
#define u(x,a)\
        x[0] = unpack(a,0); \
        x[1] = unpack(a,1); \
        x[2] = unpack(a,2); \
        x[3] = unpack(a,3);
#define release(a,b,c)  { free(a); free(b);free(c); }
#ifdef  TWOFISH
typedef struct key_t
{
    uint8_t len;
    uint8_t *k;
}key_t;
typedef struct subkey_t
{
    uint8_t len;
    uint8_t s[4][4];
    uint8_t me[4][4];
    uint8_t mo[4][4];
}subkey_t;
```

```
 67 #endif
 68 /*
 69  * Twofish Expand Key Function
 70  *
 71  * Description:
 72  *
 73  * @param   s
 74  * @param   len
 75  * @usage
 76  * {@code}
 77  */
 78 key_t* Twofish_expand_key(uint8_t *s, uint32_t len);
 79 /*
 80  * Twofish Galois Field Multiplication Function
 81  *
 82  * Description:
 83  *
 84  * @param   x
 85  * @param   y
 86  * @param   m
 87  * @usage
 88  * {@code}
 89  */
 90 uint8_t gf(uint8_t x, uint8_t y, uint16_t m);
 91 /*
 92  * Twofish Generate Subkeys Function
 93  *
 94  * Description:
 95  *
 96  * @param   tf_key
 97  * @usage
 98  * {@code}
 99  */
100 subkey_t* Twofish_generate_subkey(key_t* tf_key);
101 /*
102  * Twofish Generate Subkeys Function
103  *
104  * Description:
105  *
106  * @param   x[]
107  * @param   y[]
108  * @param   s
109  * @param   stage
110  * @usage
111  * {@code}
112  */
113 void Twofish_h(uint8_t x[],  uint8_t y[], uint8_t s[][4], int stage);
114 /*
115  * Twofish MDS Multiply Function
116  *
117  * Description:
118  *
119  * @param   y[]
120  * @param   out[]
121  * @usage
122  * {@code}
123  */
124 void Twofish_mds_mul(uint8_t y[],  uint8_t out[]);
125 /*
126  * Twofish MDS Multiply Function
127  *
128  * Description:
129  *
130  * @param   tf_twofish
131  * @param   tf_subkey
132  * @param   p
133  * @param   k
134  * @usage
135  * {@code}
136  */
```

```
137 twofish_t* Twofish_generate_ext_k_keys(twofish_t* tf_twofish, subkey_t *
    tf_subkey,uint32_t p, uint8_t k);
138 /*
139  * Twofish MDS Multiply Function
140  *
141  * Description:
142  *
143  * @param   tf_twofish
144  * @param   tf_subkey
145  * @param   k
146  * @usage
147  * {@code}
148  */
149 twofish_t* Twofish_generate_ext_s_keys(twofish_t* tf_twofish, subkey_t *
    tf_subkey, uint8_t k);
150 /*
151  * Twofish f Function
152  *
153  * Description:
154  *
155  * @param   tf_twofish
156  * @param   r
157  * @param   r0, r1
158  * @param   f0, f1
159  * @usage
160  * {@code}
161  */
162 void Twofish_f(twofish_t* tf_twofish, uint8_t r,uint32_t r0, uint32_t r1,
    uint32_t* f0, uint32_t* f1);
163 /*
164  * Twofish g Function
165  *
166  * Description:
167  *
168  * @param   tf_twofish
169  * @param   x
170  * @usage
171  * {@code}
172  */
173 uint32_t Twofish_g(twofish_t* tf_twofish, uint32_t x);
174
175 twofish_t* Twofish_setup(uint8_t *s, uint32_t len)
176 {
177     /* Expand the key if necessary. */
178     key_t* tf_key = Twofish_expand_key(s, len);
179
180     /* Generate subkeys: s and k */
181     subkey_t *tf_subkey = Twofish_generate_subkey(tf_key);
182
183     /* Generate 40 K keys */
184     twofish_t* tf_twofish = (twofish_t*)malloc(sizeof(twofish_t));
185     tf_twofish = Twofish_generate_ext_k_keys(tf_twofish,tf_subkey,0x01010101,(
    tf_key->len/8));
186      /* Generate 4x256 S keys */
187     tf_twofish = Twofish_generate_ext_s_keys(tf_twofish,tf_subkey,(tf_key->len
    /8));
188
189     /* Free memory */
190     release(tf_key->k, tf_key, tf_subkey);
191
192     return tf_twofish;
193 }
194
195 void Twofish_encryt(twofish_t* tf_twofish, uint8_t *data, uint8_t *cypher)
196 {
197     uint32_t r0, r1, r2, r3, f0, f1, c2,c3;
198     /* Input Whitenening */
199     r0 = tf_twofish->k[0]^pack(data);
200     r1 = tf_twofish->k[1]^pack(data+4);
201     r2 = tf_twofish->k[2]^pack(data+8);
```

```
202     r3 = tf_twofish->k[3]^pack(data+12);
203
204     /* The black box */
205     for (int i=0; i<16;++i)
206     {
207         Twofish_f(tf_twofish, i, r0, r1, &f0, &f1);
208         c2 = ror((f0 ^r2), 1);
209         c3 = (f1^rol(r3,1));
210         /* swap */
211         r2 = r0;
212         r3 = r1;
213         r0 = c2;
214         r1 = c3;
215     }
216
217     /* Output Whitening */
218     c2 = r0;
219     c3 = r1;
220     r0 = tf_twofish->k[4]^r2;
221     r1 = tf_twofish->k[5]^r3;
222     r2 = tf_twofish->k[6]^c2;
223     r3 = tf_twofish->k[7]^c3;
224
225     for (int i=0;i<4;++i)
226     {
227         cypher[i]   = unpack(r0,i);
228         cypher[i+4] = unpack(r1,i);
229         cypher[i+8] = unpack(r2,i);
230         cypher[i+12]= unpack(r3,i);
231     }
232 }
233
234 void Twofish_decryt(twofish_t* tf_twofish, uint8_t *cypher, uint8_t *data)
235 {
236     uint32_t r0, r1, r2, r3, f0, f1, c2,c3;
237     /* Input Whitenening */
238     r0 = tf_twofish->k[4]^pack(cypher);
239     r1 = tf_twofish->k[5]^pack(cypher+4);
240     r2 = tf_twofish->k[6]^pack(cypher+8);
241     r3 = tf_twofish->k[7]^pack(cypher+12);
242
243     /* The black box */
244     for (int i=15; i >= 0;--i)
245     {
246         Twofish_f(tf_twofish, i, r0, r1, &f0, &f1);
247         c2 = (rol(r2,1)^f0);
248         c3 = ror((f1^r3),1);
249         /* swap */
250         r2 = r0;
251         r3 = r1;
252         r0 = c2;
253         r1 = c3;
254     }
255
256     /* Output Whitening */
257     c2 = r0;
258     c3 = r1;
259     r0 = tf_twofish->k[0]^r2;
260     r1 = tf_twofish->k[1]^r3;
261     r2 = tf_twofish->k[2]^c2;
262     r3 = tf_twofish->k[3]^c3;
263
264     for (int i=0;i<4;++i)
265     {
266         data[i]   = unpack(r0,i);
267         data[i+4] = unpack(r1,i);
268         data[i+8] = unpack(r2,i);
269         data[i+12]= unpack(r3,i);
270     }
271 }
```

```
272
273 void Twofish_f(twofish_t* tf_twofish, uint8_t r,uint32_t r0, uint32_t r1,
    uint32_t* f0, uint32_t* f1)
274 {
275     uint32_t t0, t1, o;
276     t0 = Twofish_g(tf_twofish, r0);
277     t1 = rol(r1, 8);
278     t1 = Twofish_g(tf_twofish, t1);
279     o = 2*r;
280     *f0= (t0 + t1 + tf_twofish->k[o+8]);
281     *f1= (t0 + (2*t1) + tf_twofish->k[o+9]);
282 }
283
284 twofish_t* Twofish_generate_ext_k_keys(twofish_t* tf_twofish, subkey_t *
    tf_subkey,uint32_t p, uint8_t k)
285 {
286     uint32_t a, b, o;
287     uint8_t x[4], y[4], z[4];
288     for(int i=0;i<20;++i)
289     {
290         a = (2*i*p);
291         b = (a+p);
292         u(x,a);
293         Twofish_h(x, y, tf_subkey->me, k);
294         Twofish_mds_mul(y,z);
295         a = pack(z);                        /* Convert four bytes z[4] to a
    word (a). */
296         u(x,b);                             /* Convert a word (b) to four
    bytes x[4]. */
297         Twofish_h(x, y, tf_subkey->mo, k);
298         Twofish_mds_mul(y,z);
299         b = pack(z);
300         b = rol(b, 8);
301         o = 2*i;
302         tf_twofish->k[o] = ((a + b));
303         tf_twofish->k[o+1] = rol(((a + (2*b))),9);
304     }
305     return tf_twofish;
306 }
307
308 twofish_t* Twofish_generate_ext_s_keys(twofish_t* tf_twofish, subkey_t *
    tf_subkey, uint8_t k)
309 {
310     uint8_t x[4], y[4];
311     for(int i=0;i<256;++i)
312     {
313         x[0] = x[1] = x[2] = x[3] = i;
314         Twofish_h(x, y, tf_subkey->s, k);
315         /* Special MDS multiplication */
316         tf_twofish->s[0][i] = (gf(y[0], mds[0][0],0x169) |(gf(y[0],mds[1][0],
    0x169)<< 8)|(gf(y[0], mds[2][0],0x169)<<16) |(gf(y[0], mds[3][0], 0x169) <<24)
    );
317         tf_twofish->s[1][i] = (gf(y[1], mds[0][1],0x169) |(gf(y[1],mds[1][1],
    0x169)<< 8)|(gf(y[1], mds[2][1],0x169)<<16) |(gf(y[1], mds[3][1], 0x169) <<24)
    );
318         tf_twofish->s[2][i] = (gf(y[2], mds[0][2],0x169) |(gf(y[2],mds[1][2],
    0x169)<< 8)|(gf(y[2], mds[2][2],0x169)<<16) |(gf(y[2], mds[3][2], 0x169) <<24)
    );
319         tf_twofish->s[3][i] = (gf(y[3], mds[0][3],0x169) |(gf(y[3],mds[1][3],
    0x169)<< 8)|(gf(y[3], mds[2][3],0x169)<<16) |(gf(y[3], mds[3][3], 0x169) <<24)
    );
320     }
321     return tf_twofish;
322 }
323
324 void Twofish_mds_mul(uint8_t y[],  uint8_t out[])
325 {
326     out[0] = (gf(y[0], mds[0][0], 0x169)^gf(y[1], mds[0][1], 0x169)^gf(y[2],
    mds[0][2], 0x169)^gf(y[3], mds[0][3], 0x169) );
327     out[1] = (gf(y[0], mds[1][0], 0x169)^gf(y[1], mds[1][1], 0x169)^gf(y[2],
```

```
327 mds[1][2], 0x169)^gf(y[3], mds[1][3], 0x169) );
328     out[2] = (gf(y[0], mds[2][0], 0x169)^gf(y[1], mds[2][1], 0x169)^gf(y[2],
    mds[2][2], 0x169)^gf(y[3], mds[2][3], 0x169) );
329     out[3] = (gf(y[0], mds[3][0], 0x169)^gf(y[1], mds[3][1], 0x169)^gf(y[2],
    mds[3][2], 0x169)^gf(y[3], mds[3][3], 0x169) );
330 }
331
332 uint32_t Twofish_g(twofish_t* tf_twofish, uint32_t x)
333 {
334     return (tf_twofish->s[0][unpack(x, 0)]^tf_twofish->s[1][unpack(x, 1)]^
    tf_twofish->s[2][unpack(x, 2)]^tf_twofish->s[3][unpack(x, 3)]);
335 }
336
337 void Twofish_h(uint8_t x[],  uint8_t out[], uint8_t s[][4], int stage)
338 {
339     uint8_t y[4];
340     for (int j=0; j<4;++j)
341     {
342         y[j] = x[j];
343     }
344
345     if (stage == 4)
346     {
347         y[0] = q[1][y[0]] ^ (s[3][0]);
348         y[1] = q[0][y[1]] ^ (s[3][1]);
349         y[2] = q[0][y[2]] ^ (s[3][2]);
350         y[3] = q[1][y[3]] ^ (s[3][3]);
351     }
352     if (stage > 2)
353     {
354         y[0] = q[1][y[0]] ^ (s[2][0]);
355         y[1] = q[1][y[1]] ^ (s[2][1]);
356         y[2] = q[0][y[2]] ^ (s[2][2]);
357         y[3] = q[0][y[3]] ^ (s[2][3]);
358     }
359
360     out[0] = q[1][q[0][ q[0][y[0]] ^ (s[1][0])] ^ (s[0][0])];
361     out[1] = q[0][q[0][ q[1][y[1]] ^ (s[1][1])] ^ (s[0][1])];
362     out[2] = q[1][q[1][ q[0][y[2]] ^ (s[1][2])] ^ (s[0][2])];
363     out[3] = q[0][q[1][ q[1][y[3]] ^ (s[1][3])] ^ (s[0][3])];
364 }
365
366 subkey_t* Twofish_generate_subkey(key_t* tf_key)
367 {
368     int k, r, g;
369     subkey_t *tf_subkey = (subkey_t*)malloc(sizeof(subkey_t));
370     k = tf_key->len/8;                                  /* k=N/64 */
371     for(r=0; r<k;++r)
372     {
373         /* Generate subkeys Me and Mo */
374         tf_subkey->me[r][0] = nxt(tf_key->k, r*8 + 0);
375         tf_subkey->me[r][1] = nxt(tf_key->k, r*8 + 1);
376         tf_subkey->me[r][2] = nxt(tf_key->k, r*8 + 2);
377         tf_subkey->me[r][3] = nxt(tf_key->k, r*8 + 3);
378         tf_subkey->mo[r][0] = nxt(tf_key->k, r*8 + 4);
379         tf_subkey->mo[r][1] = nxt(tf_key->k, r*8 + 5);
380         tf_subkey->mo[r][2] = nxt(tf_key->k, r*8 + 6);
381         tf_subkey->mo[r][3] = nxt(tf_key->k, r*8 + 7);
382
383         g=k-r-1;                                        /* Reverse order */
384         /* Generate subkeys S using RS matrix */
385         tf_subkey->s[g][0] = rsm(r, 0x01, 0xa4, 0x55, 0x87, 0x5a, 0x58, 0xdb,
    0x9e);
386         tf_subkey->s[g][1] = rsm(r, 0xa4, 0x56, 0x82, 0xf3, 0x1e, 0xc6, 0x68,
    0xe5);
387         tf_subkey->s[g][2] = rsm(r, 0x02, 0xa1, 0xfc, 0xc1, 0x47, 0xae, 0x3d,
    0x19);
388         tf_subkey->s[g][3] = rsm(r, 0xa4, 0x55, 0x87, 0x5a, 0x58, 0xdb, 0x9e,
    0x03);
389     }
```

```
390     return tf_subkey;
391 }
392
393 key_t* Twofish_expand_key(uint8_t *s, uint32_t len)
394 {
395     int n;
396     pad_factor(len);
397     key_t* tf_key = (key_t*)malloc(sizeof(key_t));
398     uint8_t* ss = (uint8_t*)malloc(n);
399     for (int g=0; g<n; ++g)
400     {
401         *(ss+g) = 0x00;
402         if (g < len)
403             *(ss+g) = *(s+g);
404     }
405     tf_key->k = ss;
406     tf_key->len=n;
407     return tf_key;
408 }
409
410 uint8_t gf(uint8_t x, uint8_t y, uint16_t m)
411 {
412     uint8_t c, p = 0;
413     for (int i=0; i<8; ++i)
414     {
415         if (y & 0x1)
416             p ^= x;
417         c = x & 0x80;
418         x <<= 1;
419         if (c)
420             x ^= m;
421         y >>= 1;
422     }
423     return p;
424 }
425
```