# milenage.c

```
00001 /*----------------------------------------------------------------------
00002  *          Example algorithms f1, f1*, f2, f3, f4, f5, f5*
00003  *----------------------------------------------------------------------
00004  *
00005  *  A sample implementation of the example 3GPP authentication and
00006  *  key agreement functions f1, f1*, f2, f3, f4, f5 and f5*.  This is
00007  *  a byte-oriented implementation of the functions, and of the block
00008  *  cipher kernel function Rijndael.
00009  *
00010  *  This has been coded for clarity, not necessarily for efficiency.
00011  *
00012  *  The functions f2, f3, f4 and f5 share the same inputs and have
00013  *  been coded together as a single function.  f1, f1* and f5* are
00014  *  all coded separately.
00015  *
00016  *----------------------------------------------------------------------*/
00017
00018 #include "milenage.h"
00019 #include "rijndael.h"
00020
00021 /*------------------------- prototypes -------------------------*/
00022
00023
00024
00025 /*----------------------------------------------------------------------
00026  *                            Algorithm f1
00027  *----------------------------------------------------------------------
00028  *
00029  *  Computes network authentication code MAC-A from key K, random
00030  *  challenge RAND, sequence number SQN and authentication management
00031  *  field AMF.
00032  *
00033  *----------------------------------------------------------------------*/
00034
00035 void f1    ( u8 k[16], u8 rand[16], u8 sqn[6], u8 amf[2],
00036              u8 mac_a[8], u8 op[16] )
00037 {
00038   u8 op_c[16];
00039   u8 temp[16];
00040   u8 in1[16];
00041   u8 out1[16];
00042   u8 rijndaelInput[16];
00043   u8 i;
00044
00045   RijndaelKeySchedule( k );
00046
00047   ComputeOPc( op_c, op );
00048
00049   for (i=0; i<16; i++)
00050     rijndaelInput[i] = rand[i] ^ op_c[i];
00051   RijndaelEncrypt( rijndaelInput, temp );
00052
00053   for (i=0; i<6; i++)
00054   {
00055     in1[i]    = sqn[i];
00056     in1[i+8]  = sqn[i];
00057   }
```

```
00058   for (i=0; i<2; i++)
00059   {
00060     in1[i+6]  = amf[i];
00061     in1[i+14] = amf[i];
00062   }
00063
00064   /* XOR op_c and in1, rotate by r1=64, and XOR *
00065    * on the constant c1 (which is all zeroes)   */
00066
00067   for (i=0; i<16; i++)
00068     rijndaelInput[(i+8) % 16] = in1[i] ^ op_c[i];
00069
00070   /* XOR on the value temp computed before */
00071
00072   for (i=0; i<16; i++)
00073     rijndaelInput[i] ^= temp[i];
00074
00075   RijndaelEncrypt( rijndaelInput, out1 );
00076   for (i=0; i<16; i++)
00077     out1[i] ^= op_c[i];
00078
00079   for (i=0; i<8; i++)
00080     mac_a[i] = out1[i];
00081
00082   return;
00083 } /* end of function f1 */
00084
00085
00086
00087 /*-----------------------------------------------------------------
00088  *                            Algorithms f2-f5
00089  *-----------------------------------------------------------------
00090  *
00091  *  Takes key K and random challenge RAND, and returns response RES,
00092  *  confidentiality key CK, integrity key IK and anonymity key AK.
00093  *
00094  *-----------------------------------------------------------------*/
00095
00096 void f2345 ( u8 k[16], u8 rand[16],
00097              u8 res[8], u8 ck[16], u8 ik[16], u8 ak[6], u8 op[16] )
00098 {
00099   u8 op_c[16];
00100   u8 temp[16];
00101   u8 out[16];
00102   u8 rijndaelInput[16];
00103   u8 i;
00104
00105   RijndaelKeySchedule( k );
00106
00107   ComputeOPc( op_c, op );
00108
00109   for (i=0; i<16; i++)
00110     rijndaelInput[i] = rand[i] ^ op_c[i];
00111   RijndaelEncrypt( rijndaelInput, temp );
00112
00113   /* To obtain output block OUT2: XOR OPc and TEMP,    *
00114    * rotate by r2=0, and XOR on the constant c2 (which *
00115    * is all zeroes except that the last bit is 1).     */
00116
00117   for (i=0; i<16; i++)
00118     rijndaelInput[i] = temp[i] ^ op_c[i];
00119   rijndaelInput[15] ^= 1;
00120
00121   RijndaelEncrypt( rijndaelInput, out );
```

```
00122   for (i=0; i<16; i++)
00123     out[i] ^= op_c[i];
00124
00125   for (i=0; i<8; i++)
00126     res[i] = out[i+8];
00127   for (i=0; i<6; i++)
00128     ak[i]  = out[i];
00129
00130   /* To obtain output block OUT3: XOR OPc and TEMP,         *
00131    * rotate by r3=32, and XOR on the constant c3 (which    *
00132    * is all zeroes except that the next to last bit is 1). */
00133
00134   for (i=0; i<16; i++)
00135     rijndaelInput[(i+12) % 16] = temp[i] ^ op_c[i];
00136   rijndaelInput[15] ^= 2;
00137
00138   RijndaelEncrypt( rijndaelInput, out );
00139   for (i=0; i<16; i++)
00140     out[i] ^= op_c[i];
00141
00142   for (i=0; i<16; i++)
00143     ck[i] = out[i];
00144
00145   /* To obtain output block OUT4: XOR OPc and TEMP,         *
00146    * rotate by r4=64, and XOR on the constant c4 (which     *
00147    * is all zeroes except that the 2nd from last bit is 1). */
00148
00149   for (i=0; i<16; i++)
00150     rijndaelInput[(i+8) % 16] = temp[i] ^ op_c[i];
00151   rijndaelInput[15] ^= 4;
00152
00153   RijndaelEncrypt( rijndaelInput, out );
00154   for (i=0; i<16; i++)
00155     out[i] ^= op_c[i];
00156
00157   for (i=0; i<16; i++)
00158     ik[i] = out[i];
00159
00160   return;
00161 } /* end of function f2345 */
00162
00163
00164 /*-----------------------------------------------------------------
00165  *                            Algorithm f1*
00166  *-----------------------------------------------------------------
00167  *
00168  *  Computes resynch authentication code MAC-S from key K, random
00169  *  challenge RAND, sequence number SQN and authentication management
00170  *  field AMF.
00171  *
00172  *-----------------------------------------------------------------*/
00173
00174 void f1star( u8 k[16], u8 rand[16], u8 sqn[6], u8 amf[2],
00175              u8 mac_s[8], u8 op[16] )
00176 {
00177   u8 op_c[16];
00178   u8 temp[16];
00179   u8 in1[16];
00180   u8 out1[16];
00181   u8 rijndaelInput[16];
00182   u8 i;
00183
00184   RijndaelKeySchedule( k );
00185
```

```
00186   ComputeOPc( op_c, op );
00187
00188   for (i=0; i<16; i++)
00189     rijndaelInput[i] = rand[i] ^ op_c[i];
00190   RijndaelEncrypt( rijndaelInput, temp );
00191
00192   for (i=0; i<6; i++)
00193   {
00194     in1[i]    = sqn[i];
00195     in1[i+8]  = sqn[i];
00196   }
00197   for (i=0; i<2; i++)
00198   {
00199     in1[i+6]  = amf[i];
00200     in1[i+14] = amf[i];
00201   }
00202
00203   /* XOR op_c and in1, rotate by r1=64, and XOR *
00204    * on the constant c1 (which is all zeroes)   */
00205
00206   for (i=0; i<16; i++)
00207     rijndaelInput[(i+8) % 16] = in1[i] ^ op_c[i];
00208
00209   /* XOR on the value temp computed before */
00210
00211   for (i=0; i<16; i++)
00212     rijndaelInput[i] ^= temp[i];
00213
00214   RijndaelEncrypt( rijndaelInput, out1 );
00215   for (i=0; i<16; i++)
00216     out1[i] ^= op_c[i];
00217
00218   for (i=0; i<8; i++)
00219     mac_s[i] = out1[i+8];
00220
00221   return;
00222 } /* end of function f1star */
00223
00224
00225 /*-----------------------------------------------------------------
00226  *                             Algorithm f5*
00227  *-----------------------------------------------------------------
00228  *
00229  *  Takes key K and random challenge RAND, and returns resynch
00230  *  anonymity key AK.
00231  *
00232  *-----------------------------------------------------------------*/
00233
00234 void f5star( u8 k[16], u8 rand[16],
00235              u8 ak[6], u8 op[16] )
00236 {
00237   u8 op_c[16];
00238   u8 temp[16];
00239   u8 out[16];
00240   u8 rijndaelInput[16];
00241   u8 i;
00242
00243   RijndaelKeySchedule( k );
00244
00245   ComputeOPc( op_c, op );
00246
00247   for (i=0; i<16; i++)
00248     rijndaelInput[i] = rand[i] ^ op_c[i];
00249   RijndaelEncrypt( rijndaelInput, temp );
```

```
00250
00251   /* To obtain output block OUT5: XOR OPc and TEMP,         *
00252    * rotate by r5=96, and XOR on the constant c5 (which     *
00253    * is all zeroes except that the 3rd from last bit is 1). */
00254
00255   for (i=0; i<16; i++)
00256     rijndaelInput[(i+4) % 16] = temp[i] ^ op_c[i];
00257   rijndaelInput[15] ^= 8;
00258
00259   RijndaelEncrypt( rijndaelInput, out );
00260   for (i=0; i<16; i++)
00261     out[i] ^= op_c[i];
00262
00263   for (i=0; i<6; i++)
00264     ak[i] = out[i];
00265
00266   return;
00267 } /* end of function f5star */
00268
00269
00270 /*---------------------------------------------------------------------
00271  *  Function to compute OPc from OP and K.  Assumes key schedule has
00272     already been performed.
00273  *---------------------------------------------------------------------*/
00274
00275 void ComputeOPc( u8 op_c[16], u8 op[16] )
00276 {
00277   u8 i;
00278
00279   RijndaelEncrypt( op, op_c );
00280   for (i=0; i<16; i++)
00281     op_c[i] ^= op[i];
00282
00283   return;
00284 } /* end of function ComputeOPc */
```

---

## milenage.c

Go to the documentation of this file.

```
00001 /*
00002  * 3GPP AKA - Milenage algorithm (3GPP TS 35.205, .206, .207, .208)
00003  * Copyright (c) 2006-2007 <j@w1.fi>
00004  *
00005  * This program is free software; you can redistribute it and/or modify
00006  * it under the terms of the GNU General Public License version 2 as
00007  * published by the Free Software Foundation.
00008  *
00009  * Alternatively, this software may be distributed under the terms of BSD
00010  * license.
00011  *
00012  * See README and COPYING for more details.
00013  *
00014  * This file implements an example authentication algorithm defined for
3GPP
```

```
00015  * AKA. This can be used to implement a simple HLR/AuC into hlr_auc_gw to
allow
00016  * EAP-AKA to be tested properly with real USIM cards.
00017  *
00018  * This implementations assumes that the r1..r5 and c1..c5 constants
defined in
00019  * TS 35.206 are used, i.e., r1=64, r2=0, r3=32, r4=64, r5=96, c1=00..00,
00020  * c2=00..01, c3=00..02, c4=00..04, c5=00..08. The block cipher is assumed
to
00021  * be AES (Rijndael).
00022  */
00023
00024 #include "includes.h"
00025
00026 #include "common.h"
00027 #include "crypto/aes_wrap.h"
00028 #include "milenage.h"
00029
00030
00042 int milenage_f1(const u8 *opc, const u8 *k, const u8 *_rand,
00043                 const u8 *sqn, const u8 *amf, u8 *mac_a, u8 *mac_s)
00044 {
00045         u8 tmp1[16], tmp2[16], tmp3[16];
00046         int i;
00047
00048         /* tmp1 = TEMP = E_K(RAND XOR OP_C) */
00049         for (i = 0; i < 16; i++)
00050                 tmp1[i] = _rand[i] ^ opc[i];
00051         if (aes_128_encrypt_block(k, tmp1, tmp1))
00052                 return -1;
00053
00054         /* tmp2 = IN1 = SQN || AMF || SQN || AMF */
00055         os_memcpy(tmp2, sqn, 6);
00056         os_memcpy(tmp2 + 6, amf, 2);
00057         os_memcpy(tmp2 + 8, tmp2, 8);
00058
00059         /* OUT1 = E_K(TEMP XOR rot(IN1 XOR OP_C, r1) XOR c1) XOR OP_C */
00060
00061         /* rotate (tmp2 XOR OP_C) by r1 (= 0x40 = 8 bytes) */
00062         for (i = 0; i < 16; i++)
00063                 tmp3[(i + 8) % 16] = tmp2[i] ^ opc[i];
00064         /* XOR with TEMP = E_K(RAND XOR OP_C) */
00065         for (i = 0; i < 16; i++)
00066                 tmp3[i] ^= tmp1[i];
00067         /* XOR with c1 (= ..00, i.e., NOP) */
00068
00069         /* f1 || f1* = E_K(tmp3) XOR OP_c */
00070         if (aes_128_encrypt_block(k, tmp3, tmp1))
00071                 return -1;
00072         for (i = 0; i < 16; i++)
00073                 tmp1[i] ^= opc[i];
00074         if (mac_a)
00075                 os_memcpy(mac_a, tmp1, 8); /* f1 */
00076         if (mac_s)
00077                 os_memcpy(mac_s, tmp1 + 8, 8); /* f1* */
00078         return 0;
00079 }
00080
00081
00094 int milenage_f2345(const u8 *opc, const u8 *k, const u8 *_rand,
00095                   u8 *res, u8 *ck, u8 *ik, u8 *ak, u8 *akstar)
00096 {
00097         u8 tmp1[16], tmp2[16], tmp3[16];
00098         int i;
```

```
00099
00100         /* tmp2 = TEMP = E_K(RAND XOR OP_C) */
00101         for (i = 0; i < 16; i++)
00102                 tmp1[i] = _rand[i] ^ opc[i];
00103         if (aes_128_encrypt_block(k, tmp1, tmp2))
00104                 return -1;
00105
00106         /* OUT2 = E_K(rot(TEMP XOR OP_C, r2) XOR c2) XOR OP_C */
00107         /* OUT3 = E_K(rot(TEMP XOR OP_C, r3) XOR c3) XOR OP_C */
00108         /* OUT4 = E_K(rot(TEMP XOR OP_C, r4) XOR c4) XOR OP_C */
00109         /* OUT5 = E_K(rot(TEMP XOR OP_C, r5) XOR c5) XOR OP_C */
00110
00111         /* f2 and f5 */
00112         /* rotate by r2 (= 0, i.e., NOP) */
00113         for (i = 0; i < 16; i++)
00114                 tmp1[i] = tmp2[i] ^ opc[i];
00115         tmp1[15] ^= 1; /* XOR c2 (= ..01) */
00116         /* f5 || f2 = E_K(tmp1) XOR OP_c */
00117         if (aes_128_encrypt_block(k, tmp1, tmp3))
00118                 return -1;
00119         for (i = 0; i < 16; i++)
00120                 tmp3[i] ^= opc[i];
00121         if (res)
00122                 os_memcpy(res, tmp3 + 8, 8); /* f2 */
00123         if (ak)
00124                 os_memcpy(ak, tmp3, 6); /* f5 */
00125
00126         /* f3 */
00127         if (ck) {
00128                 /* rotate by r3 = 0x20 = 4 bytes */
00129                 for (i = 0; i < 16; i++)
00130                         tmp1[(i + 12) % 16] = tmp2[i] ^ opc[i];
00131                 tmp1[15] ^= 2; /* XOR c3 (= ..02) */
00132                 if (aes_128_encrypt_block(k, tmp1, ck))
00133                         return -1;
00134                 for (i = 0; i < 16; i++)
00135                         ck[i] ^= opc[i];
00136         }
00137
00138         /* f4 */
00139         if (ik) {
00140                 /* rotate by r4 = 0x40 = 8 bytes */
00141                 for (i = 0; i < 16; i++)
00142                         tmp1[(i + 8) % 16] = tmp2[i] ^ opc[i];
00143                 tmp1[15] ^= 4; /* XOR c4 (= ..04) */
00144                 if (aes_128_encrypt_block(k, tmp1, ik))
00145                         return -1;
00146                 for (i = 0; i < 16; i++)
00147                         ik[i] ^= opc[i];
00148         }
00149
00150         /* f5* */
00151         if (akstar) {
00152                 /* rotate by r5 = 0x60 = 12 bytes */
00153                 for (i = 0; i < 16; i++)
00154                         tmp1[(i + 4) % 16] = tmp2[i] ^ opc[i];
00155                 tmp1[15] ^= 8; /* XOR c5 (= ..08) */
00156                 if (aes_128_encrypt_block(k, tmp1, tmp1))
00157                         return -1;
00158                 for (i = 0; i < 6; i++)
00159                         akstar[i] = tmp1[i] ^ opc[i];
00160         }
00161
00162         return 0;
```

```
00163 }
00164
00165
00179 void milenage_generate(const u8 *opc, const u8 *amf, const u8 *k,
00180                        const u8 *sqn, const u8 *_rand, u8 *autn, u8 *ik,
00181                        u8 *ck, u8 *res, size_t *res_len)
00182 {
00183         int i;
00184         u8 mac_a[8], ak[6];
00185
00186         if (*res_len < 8) {
00187                 *res_len = 0;
00188                 return;
00189         }
00190         if (milenage_f1(opc, k, _rand, sqn, amf, mac_a, NULL) ||
00191             milenage_f2345(opc, k, _rand, res, ck, ik, ak, NULL)) {
00192                 *res_len = 0;
00193                 return;
00194         }
00195         *res_len = 8;
00196
00197         /* AUTN = (SQN ^ AK) || AMF || MAC */
00198         for (i = 0; i < 6; i++)
00199                 autn[i] = sqn[i] ^ ak[i];
00200         os_memcpy(autn + 6, amf, 2);
00201         os_memcpy(autn + 8, mac_a, 8);
00202 }
00203
00204
00214 int milenage_auts(const u8 *opc, const u8 *k, const u8 *_rand, const u8
*auts,
00215                   u8 *sqn)
00216 {
00217         u8 amf[2] = { 0x00, 0x00 }; /* TS 33.102 v7.0.0, 6.3.3 */
00218         u8 ak[6], mac_s[8];
00219         int i;
00220
00221         if (milenage_f2345(opc, k, _rand, NULL, NULL, NULL, NULL, ak))
00222                 return -1;
00223         for (i = 0; i < 6; i++)
00224                 sqn[i] = auts[i] ^ ak[i];
00225         if (milenage_f1(opc, k, _rand, sqn, amf, NULL, mac_s) ||
00226             memcmp(mac_s, auts + 6, 8) != 0)
00227                 return -1;
00228         return 0;
00229 }
00230
00231
00241 int gsm_milenage(const u8 *opc, const u8 *k, const u8 *_rand, u8 *sres, u8
*kc)
00242 {
00243         u8 res[8], ck[16], ik[16];
00244         int i;
00245
00246         if (milenage_f2345(opc, k, _rand, res, ck, ik, NULL, NULL))
00247                 return -1;
00248
00249         for (i = 0; i < 8; i++)
00250                 kc[i] = ck[i] ^ ck[i + 8] ^ ik[i] ^ ik[i + 8];
00251
00252 #ifdef GSM_MILENAGE_ALT_SRES
00253         os_memcpy(sres, res, 4);
00254 #else /* GSM_MILENAGE_ALT_SRES */
00255         for (i = 0; i < 4; i++)
```

```
00256                 sres[i] = res[i] ^ res[i + 4];
00257 #endif /* GSM_MILENAGE_ALT_SRES */
00258         return 0;
00259 }
00260
00261
00276 int milenage_check(const u8 *opc, const u8 *k, const u8 *sqn, const u8
*_rand,
00277                    const u8 *autn, u8 *ik, u8 *ck, u8 *res, size_t
*res_len,
00278                    u8 *auts)
00279 {
00280         int i;
00281         u8 mac_a[8], ak[6], rx_sqn[6];
00282         const u8 *amf;
00283
00284         wpa_hexdump(MSG_DEBUG, "Milenage: AUTN", autn, 16);
00285         wpa_hexdump(MSG_DEBUG, "Milenage: RAND", _rand, 16);
00286
00287         if (milenage_f2345(opc, k, _rand, res, ck, ik, ak, NULL))
00288                 return -1;
00289
00290         *res_len = 8;
00291         wpa_hexdump_key(MSG_DEBUG, "Milenage: RES", res, *res_len);
00292         wpa_hexdump_key(MSG_DEBUG, "Milenage: CK", ck, 16);
00293         wpa_hexdump_key(MSG_DEBUG, "Milenage: IK", ik, 16);
00294         wpa_hexdump_key(MSG_DEBUG, "Milenage: AK", ak, 6);
00295
00296         /* AUTN = (SQN ^ AK) || AMF || MAC */
00297         for (i = 0; i < 6; i++)
00298                 rx_sqn[i] = autn[i] ^ ak[i];
00299         wpa_hexdump(MSG_DEBUG, "Milenage: SQN", rx_sqn, 6);
00300
00301         if (os_memcmp(rx_sqn, sqn, 6) <= 0) {
00302                 u8 auts_amf[2] = { 0x00, 0x00 }; /* TS 33.102 v7.0.0,
6.3.3 */
00303                 if (milenage_f2345(opc, k, _rand, NULL, NULL, NULL, NULL,
ak))
00304                         return -1;
00305                 wpa_hexdump_key(MSG_DEBUG, "Milenage: AK*", ak, 6);
00306                 for (i = 0; i < 6; i++)
00307                         auts[i] = sqn[i] ^ ak[i];
00308                 if (milenage_f1(opc, k, _rand, sqn, auts_amf, NULL, auts +
6))
00309                         return -1;
00310                 wpa_hexdump(MSG_DEBUG, "Milenage: AUTS", auts, 14);
00311                 return -2;
00312         }
00313
00314         amf = autn + 6;
00315         wpa_hexdump(MSG_DEBUG, "Milenage: AMF", amf, 2);
00316         if (milenage_f1(opc, k, _rand, rx_sqn, amf, mac_a, NULL))
00317                 return -1;
00318
00319         wpa_hexdump(MSG_DEBUG, "Milenage: MAC_A", mac_a, 8);
00320
00321         if (os_memcmp(mac_a, autn + 8, 8) != 0) {
00322                 wpa_printf(MSG_DEBUG, "Milenage: MAC mismatch");
00323                 wpa_hexdump(MSG_DEBUG, "Milenage: Received MAC_A",
00324                             autn + 8, 8);
00325                 return -1;
00326         }
00327
00328         return 0;
```

```
00329 }
```

---

http://hostap.epitest.fi/wpa_supplicant/devel/milenage_8h.html

http://mirror.umd.edu/roswiki/doc/diamondback/api/wpa_supplicant/html/milenage_8c.html

---