

```
1 /*
2  * A pedagogical implementation of the GSM A5/1 and A5/2 "voice privacy"
3  * encryption algorithms.
4  *
5  * Copyright (C) 1998-1999: Marc Briceno, Ian Goldberg, and David Wagner
6  *
7  * The source code below is optimized for instructional value and clarity.
8  * Performance will be terrible, but that's not the point.
9  *
10 * This software may be export-controlled by US law.
11 *
12 * This software is free for commercial and non-commercial use as long as
13 * the following conditions are adhered to.
14 * Copyright remains the authors' and as such any Copyright notices in
15 * the code are not to be removed.
16 * Redistribution and use in source and binary forms, with or without
17 * modification, are permitted provided that the following conditions
18 * are met:
19 *
20 * 1. Redistributions of source code must retain the copyright
21 *   notice, this list of conditions and the following disclaimer.
22 * 2. Redistributions in binary form must reproduce the above copyright
23 *   notice, this list of conditions and the following disclaimer in the
24 *   documentation and/or other materials provided with the distribution.
25 *
26 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESS OR IMPLIED
27 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
28 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
29 * IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY
30 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
31 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
32 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
33 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER
34 * IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
35 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
36 * IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
37 *
38 * The license and distribution terms for any publicly available version
39 * or derivative of this code cannot be changed. i.e. this code cannot
40 * simply be copied and put under another distribution license
41 * [including the GNU Public License].
42 *
43 * Background: The Global System for Mobile communications is the most
44 * widely deployed digital cellular telephony system in the world. GSM
45 * makes use of four core cryptographic algorithms, none of which has
46 * been published by the GSM MOU. This failure to subject the
47 * algorithms to public review is all the more puzzling given that over
48 * 215 million GSM subscribers are expected to rely on the claimed
49 * security of the system.
50 *
```

---

51 \* The four core GSM cryptographic algorithms are:  
52 \* A3 authentication algorithm  
53 \* A5/1 "stronger" over-the-air voice-privacy algorithm  
54 \* A5/2 "weaker" over-the-air voice-privacy algorithm  
55 \* A8 voice-privacy key generation algorithm  
56 \*  
57 \* In April of 1998, our group showed that COMP128, the algorithm used by the  
58 \* overwhelming majority of GSM providers for both A3 and A8 functionality  
59 \* is fatally flawed and allows for cloning of GSM mobile phones.  
60 \*  
61 \* Furthermore, we demonstrated that all A8 implementations we could locate,  
62 \* including the few that did not use COMP128 for key generation, had been  
63 \* deliberately weakened by reducing the keyspace from 64 bits to 54 bits.  
64 \* The remaining 10 bits are simply set to zero!  
65 \*  
66 \* See <http://www.scard.org/gsm> for additional information.  
67 \*  
68 \* [May 1999]  
69 \* One question so far unanswered is if A5/1, the "stronger" of the two  
70 \* widely deployed voice-privacy algorithm is at least as strong as the  
71 \* key. Meaning: "Does A5/1 have a work factor of at least 54 bits"?  
72 \* Absent a publicly available A5/1 reference implementation, this question  
73 \* could not be answered. We hope that our reference implementation below,  
74 \* which has been verified against official A5/1 test vectors, will provide  
75 \* the cryptographic community with the base on which to construct the  
76 \* answer to this important question.  
77 \*  
78 \* Initial indications about the strength of A5/1 are not encouraging.  
79 \* A variant of A5, while not A5/1 itself, has been estimated to have a  
80 \* work factor of well below 54 bits. See <http://jya.com/crack-a5.htm> for  
81 \* background information and references.  
82 \*  
83 \* With COMP128 broken and A5/1 published below, we will now turn our  
84 \* attention to A5/2.  
85 \*  
86 \* [August 1999]  
87 \* 19th Annual International Cryptology Conference - Crypto'99  
88 \* Santa Barbara, California  
89 \*  
90 \* A5/2 has been added to the previously published A5/1 source. Our  
91 \* implementation has been verified against official test vectors.  
92 \*  
93 \* This means that our group has now reverse engineered the entire set  
94 \* of cryptographic algorithms used in the overwhelming majority of GSM  
95 \* installations, including all the over-the-air "voice privacy" algorithms.  
96 \*  
97 \* The "voice privacy" algorithm A5/2 proved especially weak. Which perhaps  
98 \* should come as no surprise, since even GSM MOU members have admitted that  
99 \* A5/2 was designed with heavy input by intelligence agencies to ensure  
100 \* breakability. Just how insecure is A5/2? It can be broken in real time

---

```
101  * with a work factor of a mere 16 bits. GSM might just as well use no "voice
102  * privacy" algorithm at all.
103  *
104  * We announced the break of A5/2 at the Crypto'99 Rump Session.
105  * Details will be published in a scientific paper following soon.
106  *
107  *
108  * -- Marc Briceno      <marc@scard.org>
109  *   Voice:            +1 (925) 798-4042
110  *
111  */
112
113
114 #include <stdio.h>
115
116
117 /* Masks for the shift registers */
118 #define R1MASK  0x07FFFF /* 19 bits, numbered 0..18 */
119 #define R2MASK  0x3FFFFFF /* 22 bits, numbered 0..21 */
120 #define R3MASK  0x7FFFFFF /* 23 bits, numbered 0..22 */
121 #ifndef A5_2
122 #define R4MASK  0x01FFFF /* 17 bits, numbered 0..16 */
123 #endif /* A5_2 */
124
125
126 #ifndef A5_2
127 /* Middle bit of each of the three shift registers, for clock control */
128 #define R1MID   0x000100 /* bit 8 */
129 #define R2MID   0x000400 /* bit 10 */
130 #define R3MID   0x000400 /* bit 10 */
131 #else /* A5_2 */
132 /* A bit of R4 that controls each of the shift registers */
133 #define R4TAP1  0x000400 /* bit 10 */
134 #define R4TAP2  0x000008 /* bit 3 */
135 #define R4TAP3  0x000080 /* bit 7 */
136 #endif /* A5_2 */
137
138
139 /* Feedback taps, for clocking the shift registers.
140  * These correspond to the primitive polynomials
141  *  $x^{19} + x^5 + x^2 + x + 1$ ,  $x^{22} + x + 1$ ,
142  *  $x^{23} + x^{15} + x^2 + x + 1$ , and  $x^{17} + x^5 + 1$ . */
143
144
145 #define R1TAPS  0x072000 /* bits 18,17,16,13 */
146 #define R2TAPS  0x300000 /* bits 21,20 */
147 #define R3TAPS  0x700080 /* bits 22,21,20,7 */
148 #ifndef A5_2
149 #define R4TAPS  0x010800 /* bits 16,11 */
150 #endif /* A5_2 */
```

```
151
152
153 typedef unsigned char byte;
154 typedef unsigned long word;
155 typedef word bit;
156
157
158 /* Calculate the parity of a 32-bit word, i.e. the sum of its bits modulo 2
159 */
160 bit parity(word x) {
161     x ^= x>>16;
162     x ^= x>>8;
163     x ^= x>>4;
164     x ^= x>>2;
165     x ^= x>>1;
166     return x&1;
167 }
168
169
170 /* Clock one shift register. For A5/2, when the last bit of the frame
171 * is loaded in, one particular bit of each register is forced to '1';
172 * that bit is passed in as the last argument. */
173 #ifndef A5_2
174 word clockone(word reg, word mask, word taps) {
175 #else /* A5_2 */
176 word clockone(word reg, word mask, word taps, word loaded_bit) {
177 #endif /* A5_2 */
178     word t = reg & taps;
179     reg = (reg << 1) & mask;
180     reg |= parity(t);
181 #ifdef A5_2
182     reg |= loaded_bit;
183 #endif /* A5_2 */
184     return reg;
185 }
186
187
188 /* The three shift registers. They're in global variables to make the code
189 * easier to understand.
190 * A better implementation would not use global variables. */
191 word R1, R2, R3;
192 #ifdef A5_2
193 word R4;
194 #endif /* A5_2 */
195
196
197 /* Return 1 iff at least two of the parameter words are non-zero. */
198 bit majority(word w1, word w2, word w3) {
199     int sum = (w1 != 0) + (w2 != 0) + (w3 != 0);
200     if (sum >= 2)
```

```

201         return 1;
202     else
203         return 0;
204 }
205
206
207 /* Clock two or three of R1,R2,R3, with clock control
208  * according to their middle bits.
209  * Specifically, we clock Ri whenever Ri's middle bit
210  * agrees with the majority value of the three middle bits. For A5/2,
211  * use particular bits of R4 instead of the middle bits. Also, for A5/2,
212  * always clock R4.
213  * If allP == 1, clock all three of R1,R2,R3, ignoring their middle bits.
214  * This is only used for key setup. If loaded == 1, then this is the last
215  * bit of the frame number, and if we're doing A5/2, we have to set a
216  * particular bit in each of the four registers. */
217 void clock(int allP, int loaded) {
218     #ifndef A5_2
219         bit maj = majority(R1&R1MID, R2&R2MID, R3&R3MID);
220         if (allP || ((R1&R1MID)!=0) == maj)
221             R1 = clockone(R1, R1MASK, R1TAPS);
222         if (allP || ((R2&R2MID)!=0) == maj)
223             R2 = clockone(R2, R2MASK, R2TAPS);
224         if (allP || ((R3&R3MID)!=0) == maj)
225             R3 = clockone(R3, R3MASK, R3TAPS);
226     #else /* A5_2 */
227         bit maj = majority(R4&R4TAP1, R4&R4TAP2, R4&R4TAP3);
228         if (allP || ((R4&R4TAP1)!=0) == maj)
229             R1 = clockone(R1, R1MASK, R1TAPS, loaded<<15);
230         if (allP || ((R4&R4TAP2)!=0) == maj)
231             R2 = clockone(R2, R2MASK, R2TAPS, loaded<<16);
232         if (allP || ((R4&R4TAP3)!=0) == maj)
233             R3 = clockone(R3, R3MASK, R3TAPS, loaded<<18);
234         R4 = clockone(R4, R4MASK, R4TAPS, loaded<<10);
235     #endif /* A5_2 */
236 }
237
238
239 /* Generate an output bit from the current state.
240  * You grab a bit from each register via the output generation taps;
241  * then you XOR the resulting three bits. For A5/2, in addition to
242  * the top bit of each of R1,R2,R3, also XOR in a majority function
243  * of three particular bits of the register (one of them complemented)
244  * to make it non-linear. Also, for A5/2, delay the output by one
245  * clock cycle for some reason. */
246 bit getbit() {
247     bit topbits = (((R1 >> 18) ^ (R2 >> 21) ^ (R3 >> 22)) & 0x01);
248     #ifndef A5_2
249         return topbits;
250     #else /* A5_2 */

```

```

251     static bit delaybit = 0;
252     bit nowbit = delaybit;
253     delaybit = (
254         topbits
255         ^ majority(R1&0x8000, (~R1)&0x4000, R1&0x1000)
256         ^ majority((~R2)&0x10000, R2&0x2000, R2&0x200)
257         ^ majority(R3&0x40000, R3&0x10000, (~R3)&0x2000)
258     );
259     return nowbit;
260 #endif /* A5_2 */
261 }
262
263
264 /* Do the A5 key setup.  This routine accepts a 64-bit key and
265  * a 22-bit frame number. */
266 void keysetup(byte key[8], word frame) {
267     int i;
268     bit keybit, framebit;
269
270
271     /* Zero out the shift registers. */
272     R1 = R2 = R3 = 0;
273 #ifdef A5_2
274     R4 = 0;
275 #endif /* A5_2 */
276
277
278     /* Load the key into the shift registers,
279     * LSB of first byte of key array first,
280     * clocking each register once for every
281     * key bit loaded.  (The usual clock
282     * control rule is temporarily disabled.) */
283     for (i=0; i<64; i++) {
284         clock(1,0); /* always clock */
285         keybit = (key[i/8] >> (i&7)) & 1; /* The i-th bit of the key */
286         R1 ^= keybit; R2 ^= keybit; R3 ^= keybit;
287 #ifdef A5_2
288         R4 ^= keybit;
289 #endif /* A5_2 */
290     }
291
292
293     /* Load the frame number into the shift registers, LSB first,
294     * clocking each register once for every key bit loaded.
295     * (The usual clock control rule is still disabled.)
296     * For A5/2, signal when the last bit is being clocked in. */
297     for (i=0; i<22; i++) {
298         clock(1,i==21); /* always clock */
299         framebit = (frame >> i) & 1; /* The i-th bit of the frame # */
300         R1 ^= framebit; R2 ^= framebit; R3 ^= framebit;

```

```
301 #ifdef A5_2
302         R4 ^= framebit;
303 #endif /* A5_2 */
304     }
305
306
307     /* Run the shift registers for 100 clocks
308     * to mix the keying material and frame number
309     * together with output generation disabled,
310     * so that there is sufficient avalanche.
311     * We re-enable the majority-based clock control
312     * rule from now on. */
313     for (i=0; i<100; i++) {
314         clock(0,0);
315     }
316     /* For A5/2, we have to load the delayed output bit. This does _not_
317     * change the state of the registers. For A5/1, this is a no-op. */
318     getbit();
319
320
321     /* Now the key is properly set up. */
322 }
323
324
325 /* Generate output. We generate 228 bits of
326 * keystream output. The first 114 bits is for
327 * the A->B frame; the next 114 bits is for the
328 * B->A frame. You allocate a 15-byte buffer
329 * for each direction, and this function fills
330 * it in. */
331 void run(byte AtoBkeystream[], byte BtoAkeystream[]) {
332     int i;
333
334
335     /* Zero out the output buffers. */
336     for (i=0; i<=113/8; i++)
337         AtoBkeystream[i] = BtoAkeystream[i] = 0;
338
339
340     /* Generate 114 bits of keystream for the
341     * A->B direction. Store it, MSB first. */
342     for (i=0; i<114; i++) {
343         clock(0,0);
344         AtoBkeystream[i/8] |= getbit() << (7-(i&7));
345     }
346
347
348     /* Generate 114 bits of keystream for the
349     * B->A direction. Store it, MSB first. */
350     for (i=0; i<114; i++) {
```

```
351         clock(0,0);
352         BtoAkeystream[i/8] |= getbit() << (7-(i&7));
353     }
354 }
355
356
357 /* Test the code by comparing it against
358  * a known-good test vector. */
359 void test() {
360 #ifndef A5_2
361     byte key[8] = {0x12, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF};
362     word frame = 0x134;
363     byte goodAtoB[15] = { 0x53, 0x4E, 0xAA, 0x58, 0x2F, 0xE8, 0x15,
364                          0x1A, 0xB6, 0xE1, 0x85, 0x5A, 0x72, 0x8C, 0x00 };
365     byte goodBtoA[15] = { 0x24, 0xFD, 0x35, 0xA3, 0x5D, 0x5F, 0xB6,
366                          0x52, 0x6D, 0x32, 0xF9, 0x06, 0xDF, 0x1A, 0xC0 };
367 #else /* A5_2 */
368     byte key[8] = {0x00, 0xfc, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
369     word frame = 0x21;
370     byte goodAtoB[15] = { 0xf4, 0x51, 0x2c, 0xac, 0x13, 0x59, 0x37,
371                          0x64, 0x46, 0x0b, 0x72, 0x2d, 0xad, 0xd5, 0x00 };
372     byte goodBtoA[15] = { 0x48, 0x00, 0xd4, 0x32, 0x8e, 0x16, 0xa1,
373                          0x4d, 0xcd, 0x7b, 0x97, 0x22, 0x26, 0x51, 0x00 };
374 #endif /* A5_2 */
375     byte AtoB[15], BtoA[15];
376     int i, failed=0;
377
378
379     keysetup(key, frame);
380     run(AtoB, BtoA);
381
382
383     /* Compare against the test vector. */
384     for (i=0; i<15; i++)
385         if (AtoB[i] != goodAtoB[i])
386             failed = 1;
387     for (i=0; i<15; i++)
388         if (BtoA[i] != goodBtoA[i])
389             failed = 1;
390
391
392     /* Print some debugging output. */
393     printf("key: 0x");
394     for (i=0; i<8; i++)
395         printf("%02X", key[i]);
396     printf("\n");
397     printf("frame number: 0x%06X\n", (unsigned int)frame);
398     printf("known good output:\n");
399     printf(" A->B: 0x");
400     for (i=0; i<15; i++)
```

```
401         printf("%02X", goodAtoB[i]);
402     printf(" B->A: 0x");
403     for (i=0; i<15; i++)
404         printf("%02X", goodBtoA[i]);
405     printf("\n");
406     printf("observed output:\n");
407     printf(" A->B: 0x");
408     for (i=0; i<15; i++)
409         printf("%02X", AtoB[i]);
410     printf(" B->A: 0x");
411     for (i=0; i<15; i++)
412         printf("%02X", BtoA[i]);
413     printf("\n");
414
415
416     if (!failed) {
417         printf("Self-check succeeded: everything looks ok.\n");
418         exit(0);
419     } else {
420         /* Problems! The test vectors didn't compare*/
421         printf("\nI don't know why this broke; contact the authors.\n"
422     );
423     }
424
425
426     int main(void) {
427         test();
428         return 0;
429     }
```