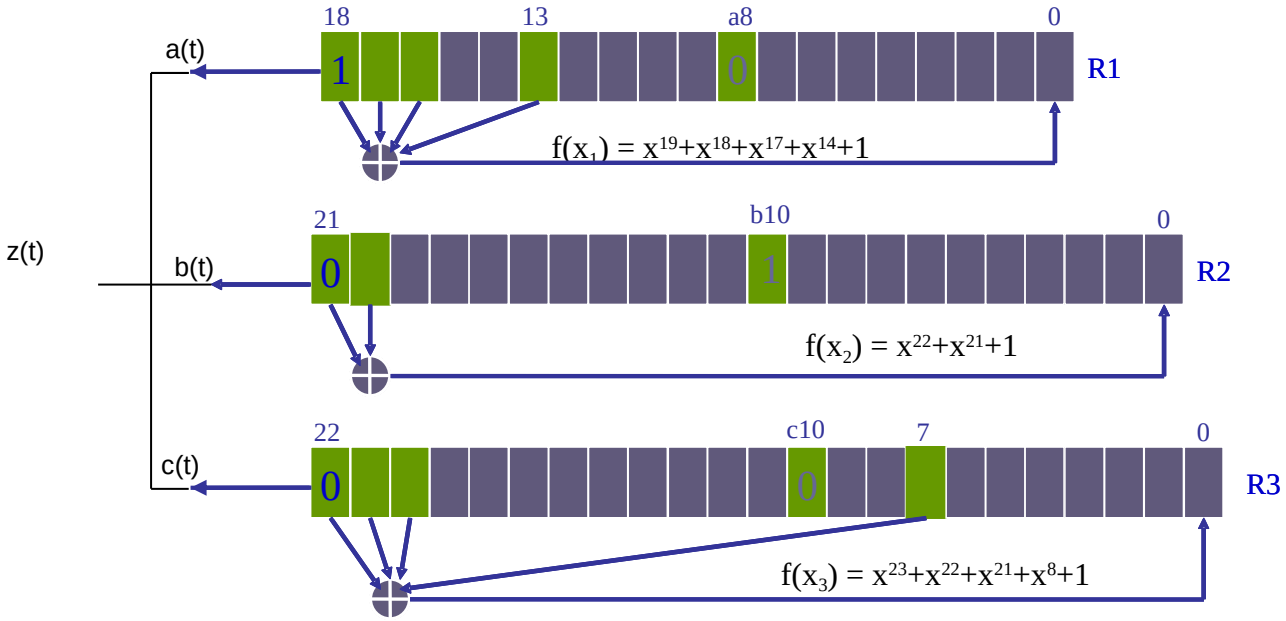
CIFRADO A5 EN GSM

1. Descripción del cifrado

El cifrado A5/1 fue propuesto en el estándar europeo GSM en 1987 (también adoptado por los Estados Unidos) para proporcionar confidencialidad en las comunicaciones con interfaz inalámbrica entre el teléfono móvil y la estación base. Existe una versión más débil (A5/2) propuesta para la exportación, principalmente a países del éste, pero desde 2006 ha dejado de ser recomendada y los teléfonos móviles actuales ya no la soportan. Ninguna de estas versiones se puso a disposición de los desarrolladores ni público, pero se dedujo por ingeniería inversa en 1999.

En GSM la comunicación se produce como secuencias de ráfagas. En un canal típico y en una dirección, una ráfaga se envía cada 4.615 milisegundos y contiene 114 bits disponibles para la información. A5/1 genera una secuencia cifrante de 114 bits que combinan con un XOR con los 114 bits de la trama de comunicación antes de la modulación. A5/1 se inicializa mediante una clave de 64 bits junto con un número de trama conocido públicamente de 22 bits.

Es un cifrado en flujo de basado en Registros de Desplazamiento Realimentados Linealmente y una función no lineal, concretamente la función mayoría (i.e. un combinador no lineal). El esquema de funcionamiento se muestra a continuación:



Los polinomios utilizados son:

LFSR1: $p_1(x) = x^{19} + x^{18} + x^{17} + x^{14} + 1$, genera a(t)
LFSR2: $p_2(x) = x^{22} + x^{21} + 1$, genera b(t)
LFSR2: $p_3(x) = x^{23} + x^{22} + x^{21} + x^8 + 1$, genera c(t)

Además las posiciones que determinan la entrada a la función mayoría son: del LFSR1, la posición 8, del LFSR2, la posición 10 y por último, del LFSR3 la posición 10. Dicha función mayoría viene definida por la expresión $F(a_8, b_{10}, c_{10}) = a_8 * b_{10} \oplus a_8 * c_{10} \oplus b_{10} * c_{10}$.

De esta forma, si el bit de la celda del registro coincide con el resultado de F, dicho registro estará en movimiento y se desplazará, en caso contrario no desplazará, quedando bloqueado en dicha etapa.

Un ejemplo de cómo se utiliza la función mayoría es el siguiente:

Supongamos $(a_8,b_{10},c_{10}) = (0,0,1)$ en la etapa i, entonces $F(a_8,b_{10},c_{10}) = 0$ con lo que los registros de desplazamiento 1 y 2 generarán en la siguiente etapa un nuevo bit cada uno y actualizarán sus estados, mientras que el LFSR3 mantendrá sin cambios su estado, generando en la etapa siguiente el mismo bit de salida.

Finalmente, la secuencia de salida del A5 se obtiene de la siguiente manera: $z(t) = a(t) \oplus b(t) \oplus c(t)$. A continuación se muestra una traza:

Semilla: 1001000100011010001

0101100111100010011010

10111100110111100001111

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				1	0	0	1	0	0	0	1	0	0	0	1	1	0	1	0	0	0	1
	0	1	0	1	1	0	0	1	1	1	1	0	0	0	1	0	0	1	1	0	1	0
1	0	1	1	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1

F(0,0,1) = 0 Registro tres queda paralizado

$z(t) = a(t) \oplus b(t) \oplus c(t) = 0$

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				0	0	1	0	0	0	1	0	0	0	1	1	0	1	0	0	0	1	1
	1	0	1	1	0	0	1	1	1	1	0	0	0	1	0	0	1	1	0	1	0	1
1	0	1	1	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1

F(1,0,1) = 1 Registro dos queda paralizado

$z(t) = a(t) \oplus b(t) \oplus c(t) = 0$

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				0	1	0	0	0	1	0	0	0	1	1	0	1	0	0	0	1	1	1
	1	0	1	1	0	0	1	1	1	1	0	0	0	1	0	0	1	1	0	1	0	1
0	1	1	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1	0

F(1,0,1) = 1 Registro dos queda paralizado

$z(t) = a(t) \oplus b(t) \oplus c(t) = 1$

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				1	0	0	0	1	0	0	0	1	1	0	1	0	0	0	1	1	1	0
	1	0	1	1	0	0	1	1	1	1	0	0	0	1	0	0	1	1	0	1	0	1
1	1	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0

F(0,0,1) = 0 Registro tres queda paralizado

$z(t) = a(t) \oplus b(t) \oplus c(t) = 1$

22	21	20	19	18	17	16	15	14	13	12	11	10	9		8	7	6	5	4	3	2	1	0
				0	0	0	1	0	0	0	1	1	0		1	0	0	0	1	1	1	0	1
	0	1	1	0	0	1	1	1	1	0	0	0	1		0	0	1	1	0	1	0	1	1
1	1	1	1	0	0	1	1	0	1	1	1	1	0		0	0	0	1	1	1	1	0	0

F(1,0,1) = 1 Registro dos queda paralizado

$z(t) = a(t) \oplus b(t) \oplus c(t) = 1$

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				0	0	1	0	0	0	1	1	0	1	0	0	0	1	1	1	0	1	0
	0	1	1	0	0	1	1	1	1	0	0	0	1	0	0	1	1	0	1	0	1	1
1	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	1

F(0,0,0) = 0 Ningún registro queda paralizado

$$z(t) = a(t) \oplus b(t) \oplus c(t) = 1$$

2. *Implementación*

En el siguiente enlace puedes encontrar una implementación en C tanto de A5/1 como de A5/2:
http://www.cryptodox.com/A5/2#Source_Code
http://www.scard.org/gsm/a51.html

```
/*
 * A pedagogical implementation of A5/1.
 *
 * Copyright (C) 1998-1999: Marc Briceno, Ian Goldberg, and David Wagner
 *
 * The source code below is optimized for instructional value and clarity.
 * Performance will be terrible, but that's not the point.
 * The algorithm is written in the C programming language to avoid ambiguities
 * inherent to the English language. Complain to the 9th Circuit of Appeals
 * if you have a problem with that.
 *
 * This software may be export-controlled by US law.
 *
 * This software is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.
 * Copyright remains the authors' and as such any Copyright notices in
 * the code are not to be removed.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The license and distribution terms for any publicly available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution license
 * [including the GNU Public License.]
 *
 * Background: The Global System for Mobile communications is the most widely
 * deployed cellular telephony system in the world. GSM makes use of
 * four core cryptographic algorithms, neither of which has been published by
 * the GSM MOU. This failure to subject the algorithms to public review is all
 * the more puzzling given that over 100 million GSM
 * subscribers are expected to rely on the claimed security of the system.
 *
 * The four core GSM algorithms are:
 * A3           authentication algorithm
 * A5/1         "strong" over-the-air voice-privacy algorithm
 * A5/2         "weak" over-the-air voice-privacy algorithm
 * A8           voice-privacy key generation algorithm
 *
 * In April of 1998, our group showed that COMP128, the algorithm used by the
 * overwhelming majority of GSM providers for both A3 and A8
 * functionality was fatally flawed and allowed for cloning of GSM mobile
 * phones.
 * Furthermore, we demonstrated that all A8 implementations we could locate,
 * including the few that did not use COMP128 for key generation, had been
```

```
 * deliberately weakened by reducing the keyspace from 64 bits to 54 bits.
 * The remaining 10 bits are simply set to zero!
 *
 * See http://www.scard.org/gsm for additional information.
 *
 * The question so far unanswered is if A5/1, the "stronger" of the two
 * widely deployed voice-privacy algorithm is at least as strong as the
 * key. Meaning: "Does A5/1 have a work factor of at least 54 bits"?
 * Absent a publicly available A5/1 reference implementation, this question
 * could not be answered. We hope that our reference implementation below,
 * which has been verified against official A5/1 test vectors, will provide
 * the cryptographic community with the base on which to construct the
 * answer to this important question.
 *
 * Initial indications about the strength of A5/1 are not encouraging.
 * A variant of A5, while not A5/1 itself, has been estimated to have a
 * work factor of well below 54 bits. See http://jya.com/crack-a5.htm for
 * background information and references.
 *
 * With COMP128 broken and A5/1 published below, we will now turn our attention
 * to A5/2. The latter has been acknowledged by the GSM community to have
 * been specifically designed by intelligence agencies for lack of security.
 *
 */


#include <stdio.h>

/* Masks for the three shift registers */
#define R1MASK  0x07FFFF /* 19 bits, numbered 0..18 */
#define R2MASK  0x3FFFFF /* 22 bits, numbered 0..21 */
#define R3MASK  0x7FFFFF /* 23 bits, numbered 0..22 */

/* Middle bit of each of the three shift registers, for clock control */
#define R1MID   0x000100 /* bit 8 */
#define R2MID   0x000400 /* bit 10 */
#define R3MID   0x000400 /* bit 10 */

/* Feedback taps, for clocking the shift registers.
 * These correspond to the primitive polynomials
 * x^19 + x^5 + x^2 + x + 1, x^22 + x + 1,
 * and x^23 + x^15 + x^2 + x + 1. */
#define R1TAPS  0x072000 /* bits 18,17,16,13 */
#define R2TAPS  0x300000 /* bits 21,20 */
#define R3TAPS  0x700080 /* bits 22,21,20,7 */

/* Output taps, for output generation */
#define R1OUT   0x040000 /* bit 18 (the high bit) */
#define R2OUT   0x200000 /* bit 21 (the high bit) */
#define R3OUT   0x400000 /* bit 22 (the high bit) */

typedef unsigned char byte;
typedef unsigned long word;
typedef word bit;

/* Calculate the parity of a 32-bit word, i.e. the sum of its bits modulo 2 */
bit parity(word x) {
   x ^= x>>16;
   x ^= x>>8;
   x ^= x>>4;
   x ^= x>>2;
   x ^= x>>1;
   return x&1;
}

/* Clock one shift register */
word clockone(word reg, word mask, word taps) {
   word t = reg & taps;
   reg = (reg << 1) & mask;
   reg |= parity(t);
   return reg;
}
```

```
/* The three shift registers.  They're in global variables to make the code
 * easier to understand.
 * A better implementation would not use global variables. */
word R1, R2, R3;

/* Look at the middle bits of R1,R2,R3, take a vote, and
 * return the majority value of those 3 bits. */
bit majority() {
   int sum;
   sum = parity(R1&R1MID) + parity(R2&R2MID) + parity(R3&R3MID);
   if (sum >= 2)
          return 1;
   else
          return 0;
}

/* Clock two or three of R1,R2,R3, with clock control
 * according to their middle bits.
 * Specifically, we clock Ri whenever Ri's middle bit
 * agrees with the majority value of the three middle bits.*/
void clock() {
   bit maj = majority();
   if (((R1&R1MID)!=0) == maj)
          R1 = clockone(R1, R1MASK, R1TAPS);
   if (((R2&R2MID)!=0) == maj)
          R2 = clockone(R2, R2MASK, R2TAPS);
   if (((R3&R3MID)!=0) == maj)
          R3 = clockone(R3, R3MASK, R3TAPS);
}

/* Clock all three of R1,R2,R3, ignoring their middle bits.
 * This is only used for key setup. */
void clockallthree() {
   R1 = clockone(R1, R1MASK, R1TAPS);
   R2 = clockone(R2, R2MASK, R2TAPS);
   R3 = clockone(R3, R3MASK, R3TAPS);
}

/* Generate an output bit from the current state.
 * You grab a bit from each register via the output generation taps;
 * then you XOR the resulting three bits. */
bit getbit() {
   return parity(R1&R1OUT)^parity(R2&R2OUT)^parity(R3&R3OUT);
}

/* Do the A5/1 key setup.  This routine accepts a 64-bit key and
 * a 22-bit frame number. */
void keysetup(byte key[8], word frame) {
   int i;
   bit keybit, framebit;

   /* Zero out the shift registers. */
   R1 = R2 = R3 = 0;

   /* Load the key into the shift registers,
    * LSB of first byte of key array first,
    * clocking each register once for every
    * key bit loaded.  (The usual clock
    * control rule is temporarily disabled.) */
   for (i=0; i<64; i++) {
          clockallthree(); /* always clock */
          keybit = (key[i/8] >> (i&7)) & 1; /* The i-th bit of the
key */
          R1 ^= keybit; R2 ^= keybit; R3 ^= keybit;
   }

   /* Load the frame number into the shift
    * registers, LSB first,
    * clocking each register once for every
    * key bit loaded.  (The usual clock
    * control rule is still disabled.) */
```

```
   for (i=0; i<22; i++) {
          clockallthree(); /* always clock */
          framebit = (frame >> i) & 1; /* The i-th bit of the frame #
*/
          R1 ^= framebit; R2 ^= framebit; R3 ^= framebit;
   }

   /* Run the shift registers for 100 clocks
    * to mix the keying material and frame number
    * together with output generation disabled,
    * so that there is sufficient avalanche.
    * We re-enable the majority-based clock control
    * rule from now on. */
   for (i=0; i<100; i++) {
          clock();
   }

   /* Now the key is properly set up. */
}

/* Generate output.  We generate 228 bits of
 * keystream output.  The first 114 bits is for
 * the A->B frame; the next 114 bits is for the
 * B->A frame.  You allocate a 15-byte buffer
 * for each direction, and this function fills
 * it in. */
void run(byte AtoBkeystream[], byte BtoAkeystream[]) {
   int i;

   /* Zero out the output buffers. */
   for (i=0; i<=113/8; i++)
          AtoBkeystream[i] = BtoAkeystream[i] = 0;

   /* Generate 114 bits of keystream for the
    * A->B direction.  Store it, MSB first. */
   for (i=0; i<114; i++) {
          clock();
          AtoBkeystream[i/8] |= getbit() << (7-(i&7));
   }

   /* Generate 114 bits of keystream for the
    * B->A direction.  Store it, MSB first. */
   for (i=0; i<114; i++) {
          clock();
          BtoAkeystream[i/8] |= getbit() << (7-(i&7));
   }
}

/* Test the code by comparing it against
 * a known-good test vector. */
void test() {
   byte key[8] = {0x12, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF};
   word frame = 0x134;
   byte goodAtoB[15] = { 0x53, 0x4E, 0xAA, 0x58, 0x2F, 0xE8, 0x15,
                         0x1A, 0xB6, 0xE1, 0x85, 0x5A, 0x72, 0x8C, 0x00 };
   byte goodBtoA[15] = { 0x24, 0xFD, 0x35, 0xA3, 0x5D, 0x5F, 0xB6,
                         0x52, 0x6D, 0x32, 0xF9, 0x06, 0xDF, 0x1A, 0xC0 };
   byte AtoB[15], BtoA[15];
   int i, failed=0;

   keysetup(key, frame);
   run(AtoB, BtoA);

   /* Compare against the test vector. */
   for (i=0; i<15; i++)
          if (AtoB[i] != goodAtoB[i])
                 failed = 1;
   for (i=0; i<15; i++)
          if (BtoA[i] != goodBtoA[i])
                 failed = 1;

   /* Print some debugging output. */
```

```
   printf("key: 0x");
   for (i=0; i<8; i++)
          printf("%02X", key[i]);
   printf("\n");
   printf("frame number: 0x%06X\n", (unsigned int)frame);
   printf("known good output:\n");
   printf(" A->B: 0x");
   for (i=0; i<15; i++)
          printf("%02X", goodAtoB[i]);
   printf("  B->A: 0x");
   for (i=0; i<15; i++)
          printf("%02X", goodBtoA[i]);
   printf("\n");
   printf("observed output:\n");
   printf(" A->B: 0x");
   for (i=0; i<15; i++)
          printf("%02X", AtoB[i]);
   printf("  B->A: 0x");
   for (i=0; i<15; i++)
          printf("%02X", BtoA[i]);
   printf("\n");

   if (!failed) {
          printf("Self-check succeeded: everything looks ok.\n");
          return;
   } else {
          /* Problems!  The test vectors didn't compare*/
          printf("\nI don't know why this broke; contact the authors.\n");
          exit(1);
   }
}

int main(void) {
   test();
   return 0;
}
```

Analiza dicha implementación, identificando cada uno de los elementos que componen el sistema.
Modifica dicho código para que desarrolle la traza descrita en el primer apartado de este documento, mostrando la secuencia cifrante generada y el texto cifrado.

3. Rompiendo el cifrado A5/2

En la segunda parte de esta práctica se va a utilizar la herramienta A52HackTool para romper el algoritmo A5/2. Esta herramienta, escrita en código C, permite simular un ataque con texto cifrado.

El código fuente está disponible en http://www.npag.fr/doc/projects/a52hacktool/source.zip y la documentación la puedes consultar en http://www.npag.fr/doc/projects/a52hacktool/doxygen/index.html.

Antes de trabajar con el código fuente es obligatorio leer el artículo Paglieri, N. and Benjamin, O., Implementation and performance analysis of Barkan, Biham and Keller's attack on A5/2, escrito por los autores de la herramienta en el que se describe detalladamente la implementación del ataque, http://www.npag.fr/doc/papers/a52hacktool.pdf.

El objetivo de esa práctica es analizar el ataque presentado en este artículo de acuerdo con los puntos siguientes:

- Describe en un informe los pasos a seguir a nivel teórico para desarrollar el ataque.
- Consulta antes de usar la herramienta la documentación generada con Doxygen, incluida en el directorio doc.
- Como ya sabrás a estas alturas, la herramienta necesita de la precomputación de un conjunto de matrices ($H \times P^{-1}$) relacionadas con el proceso de codificación y el de cifrado. Puesto que esta precomputación es muy costosa, estas matrices las tienes

disponibles en XXXXX y las debes descargar en el directorio bin que se genera al descargar y descomprimir la herramienta. Si lo prefieres las puedes generar con la opción –PRECOMPUTE de la aplicación.

- Debes comprobar el funcionamiento de la aplicación del ataque con los parámetros facilitados en ataques con la opción --AUTOTEST.
- Averigua cuáles son los parámetros necesarios y sus características para desarrollar el cifrado y descifrado con las opciones --ENCRYPT y –DECRYPT respectivamente.
- Averigua cuáles son los parámetros requeridos para realizar el ataque con la opción –ATTACK.
- Incluye en el informe las respuestas a los apartados anteriores junto con una descripción detallada de las dificultades encontradas al lanzar la herramienta.