A5/1 Soft Version Explained I

Some useful hints and preventions that should be taken

By Juan Chamero, juan.chamero@intag.org, as of April 2006

Source: GSM-Security Net, A Pedagogical Implementation of A5/1 (1999), from Marc Briceno, Ian Goldberg, and David Wagner, based on a A5/1 (chip) Reverse Engineering performed by Briceno. This algorithm has been emulated and optimized by other authors but the whole logic remained pretty much the same until now. However some cares should be taken when applied to GSM and related cryptanalysis work. This should be considered what's announced, a pedagogical implementation where differences with working algorithms could be significant in its mechanic and in its overall efficiency.

This is an algorithm to generate 228 bits « cipher masks » based on a Private Key **Kc** and a Public key **Fn**. It may be envisaged in five sections as depicted in different shadows below. As a States Machine it's constituted by three LFSR, Linear Feedback Shift registers working linked synchronically in two ways along time steps: Linear and Non Linear. The non linearity is generated via a Majority Function that rules registers shifting (clocking). If this function is deactivated the process becomes linear. We have built a fast version of this crucial algorithm. The version depicted below belongs to Briceno.

Warning: take a look to the small "main" shadowed in black. It departs fromclassical LFSR's mechanic as it clocks before calculating the outcome bits of the cipher mask instead of a real or virtual simultaneity. This becomes a significant feature if this version is used to emulateA5/1 working algorithms. Doing this way from two to three leftmost bits are ignored from the beginning. In Cryptanalysis work this characteristic would mean that more than a unique outcome will result of a given internal state for a given pair [Kc, Fn].

Steps S(0) to S(64) ⇔ Kc is bit per bit XORed to all Registers (Linear)
Steps S(65) to S(86) ⇔ Fn is bit per bit XORed to all Registers (Linear)
Steps S(87) to S(186) ⇔ 100 steps to randomize well before mask generation starts (Non Linear)
Steps S(187) to S(414) ⇔ 228 steps to generate the 228 bits Cipher Mask (Non Linear)

S(86): The Initial State, the interface that separates Linear from Non Linear behavior. If somehow inferred Kc could be retrieved by GF2 Matrix Algebra

Emerald ⇔ BB only Lineal
Gray ⇔ Non Lineal added
Black ⇔ Output bit generation
Red ⇔ Test of algorithm. You need to activate everything !
Black ⇔ Small main, activated here for test after its compilation.

A5/1 Desktop Basic Architecture

You will need it!

After successful Initialization checkup Script should stop asking what to do:

a) Run & Stop for Debugging Navigation purposes
b) Run N steps to take time
c) Run algorithm through step j
d) Run algorithm from step j to k

a) Detail

a1) Pair [Kc, Fn] loading

It must stop to feed Key (Kc);
It must stop to feed Frame

Note: If Kc remains the same should be an option to skip going to feed frame (the same Kc but different Fn)

a2) Recommended Preprogrammed stops and prints:
Si(64), Si(86), Si(186), and Si(414) for all i (i points to registers content R1, R2 and R3), printing them with the following format:

04 [{0}{1010100001110100010}{0-0}] [{0}{1111000110001101110101}{1-0}] [{0}{00000001000110101101011}{1-0}] [0]

Where:

- 04 ⇔ step, master clock
- [{0}{1010100001110100010}{0-0}] ⇔ {output R1 bit}{R1 content in bits}{(R1 tap bits XORed)-(Key bit)}
- [{0}{1111000110001101110101}{1-0}] ⇔ {output R2 bit}{R2 content in bits}{(R2 tap bits XORed)-(Key bit)}
- [{0}{00000001000110101101011}{1-0}] ⇔ {output R3 bit}{R3 content in bits}{(R3 tap bits XORed)-(Key bit)}
- [0] ⇔ Output bit, XOR of the three partial Outputs

a3) Pause: asking whether user wants to proceed with another pair.

b) Detail
Nothing special is required. N should be big enough to measure the unit Cipher Mask generation time precisely.

c, d) Details
Algorithm should go to any state j, for a given pair [Kc Fn] , stops and print S(j).

Note: Should be convenient for statistical analysis to print triads [d1 d2 d3], where di informs about register shifts along the non-linear process.

```
#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;
}
/*THIS IS THE NON LINEAL PART – IGNORE IF LINEAL TILL END*/
/* 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);
}

/*IF NOT OUTPUT GENERATION IGNORE THIS PART ALSO!*/

/* 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 BB 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;
		}

/*IF NOT ENTER INTO NON LINEARITY IGNORE THIS PART ALSO!*/
		/* 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;
}

/*END*/
```