LFSR provides encryption

Antonella Di Lillo and Giovanni Motta, Brandeis University, Waltham, MA

LFSRs (linear-feedback shift registers) find extensive use in cryptography. For example, the cryptographic algorithms in the GSM (Global System for Mobile communications) mobile-phone system rely on the use of LFSRs. An LFSR comprises a register containing a sequence of bits and a feedback function. In general, this function is an XOR (exclusive-OR) operation on certain bits in the register. The list of these bits is a “tap sequence.” You use an LFSR to generate a pseudorandom sequence of bits that undergo an XOR operation. The XOR result then connects to the input of the LFSR. Repeating the process at the decoder side returns the original sequence of bits. Listing 1 presents the encryption and decryption process. To generate a pseudorandom sequence, you load the register with a nonzero content, and the software then computes the XOR of the taps and shifts all bits in the register one bit to the left. Finally, the routine inserts the results of the XOR operation in the rightmost position (Figure 1). The program adds this 1-bit result to the sequence and repeats the procedure to generate other bits.

LFSRs are well-suited to hardware implementations, but their use in software programs unfortunately often suffers from inefficient implementations. LFSRs with few taps, or sparse LFSRs, are easier to use because you need calculate the XOR of only a few bits. On the other hand, for cryptographic purposes, you must avoid sparse polynomials because the resulting algorithms are easy to break. The C program in Listing 1 has the advantage of implementing the XOR of a 32-bit integer with an efficient algorithm, thus making an efficient way to implement the software of an LFSR. The program encrypts a standard input into a standard output one character at a time. You use an LFSR to generate eight random bits at a time, and the routine XORs the random bits to the current character. The encryption key is a nonzero integer that you use as the initial status of the register. The key is the same for both encoding and decoding. The variable taps represent the 32 binary coefficients of a primitive polynomial of degree 31. Several other choices are possible; Listing 1 suggests five of them in the comments. You can safely remove the code that describes the number of characters encrypted and the running time to make the program even smaller. You can download the software from the Web version of this article at www.ednmag.com.

Listing 1—LFSR Encryption and Decryption

```c
// Usage: LFSR <Key>
// Key is a non-zero 32 bit integer
// Alternative taps: 0x8F75C1CF, 0xC679E105,
// 0x844EC703, 0x8B4F7253, 0xDF3B011,
#include <stdio.h>
#include <time.h>

// XOR of the bits in a 32 bit integer
int xor_32(int a) {
    a ^= a >> 1;
    a ^= a >> 2;
    a ^= a >> 4;
    a ^= a >> 8;
    a ^= a >> 16;
    return(a & 0x1);
}

int main(int argc, char *argv[]) {  // Usage:
    int LFSR, c_char, i, cnt = 0, kcps;
    int taps = 0x924D1187; // Polynomial
    clock_t start, double interval;

    if(argc < 2) {  // Usage
        fprintf(stderr, "Usage: \n");
        fprintf(stderr, "LFSR <\\n> \n", argv[0]);
        return -1;
    }
    if(!(LFSR = atoi(argv[1]))) { // LFSR Status
        fprintf(stderr, "Key must be non-zero\n");
        return -1;
    }

    start = clock();
    while((c_char = fgetc(stdin)) != EOF) {  // CLOCKS_PER_SEC:
        for(i=0; i<8; i++)
            LFSR = (LFSR<<1) | xor_32(LFSR & taps);
        fputc((LFSR & 0xFF) * c_char, stdout);
        cnt++;
    }

    interval = (double)(clock()-start) /
               CLOCKS_PER_SEC;
    kcps = (int)(cnt / interval);
    fprintf(stderr, "Encoded \%d chars, \n", kcps);
    fprintf(stderr, "in \%d seconds, \n", interval);
    fprintf(stderr, "\%d char/sec\n", kcps);
    return 0;
}
```