Cracking a LFSR stream cipher

Project 2

Due Th 10/5

Summary

In this project we will execute a known plaintext attack on a LFSR stream cipher.

In particular, we will crack an LFSR with degree $m = 64$. Given will be a long ciphertext which is known to be the encryption of a book from Project Gutenberg. The first few bytes of works on Project Gutenberg are all very similar, which allows for a known plaintext attack. The target ciphertext is in this directory and has the name `target_ciphertext`.

Using known plaintext, we will recover $2m = 128$ bits of keystream, and solve a system of linear equations to determine the tap bits $p_0, p_1, \ldots, p_{63}$. If the basic idea of this plan is not clear to you, please read the section of the textbook on this topic (Chapter 2).

In order to do the attack you will first need to implement some parts of an LFSR. Then you will use Sage together with some known plaintext to produce the original state and tap variables used in the encryption of the target. Finally you will use your LFSR to decrypt the target and produce the solution.

`lfsr.c`

I have prepared an outline of the code you need to write in the file called `lfsr.c` which you should find in this directory.

This program implements a 64 bit LFSR based on the C type `uint64_t`. You can think of variables of this type as 64 bit unsigned integers. We use one of these types to represent the state of the LFSR (i.e. $s_{m-1}, s_{m-2}, \ldots, s_1, s_0$) and one to represent the tap bits (i.e. $p_{m-1}, p_{m-2}, \ldots, p_1, p_0$).

You can compile and run this code from a terminal on cocalc with these commands:

```
~/project_2$ gcc lfsr.c
~/project_2$ ./a.out
```

Optionally you could use the `clang` compiler rather than `gcc`; generally `clang` has more helpful error messages than `gcc`.

The output is a file called `ct.dat`. Right now it is identical to the plaintext input, namely `toy_pt.txt`. After you implement the missing functions in `lfsr.c`, the output should be identical to the data in the file `ct.toy`. You can check whether they agree from a cocalc terminal with this command:

```
~/project_2$ diff ct.dat ct.toy
```

If there is no output, then you have successfully implemented your LFSR. If there is output saying that the files differ, then your LFSR does not yet work correctly. By confirming that `ct.dat` and `ct.toy` agree, you can be confident that your LFSR works before moving on to other parts of the project.

We will say more below about the missing functions which you need to implement, after we introduce the basic datatypes used to implement our 64 bit LFSR.

The code in `lfsr.c` represents a 64 bit LFSR in the following way. The 64 bits of both the state bits and the taps are represented using the `uint64_t`, which is a 64 bit integer type. The least significant bits of a `uint64_t` correspond to the small indexes for s_i and p_i. For example, the binary number (expressed in hex)

```
uint64_t state = 0x0000000000000005
```

would indicate that $s_0 = 1$, $s_1 = 0$, $s_2 = 1$, and $s_i = 0$ when $i \notin \{0, 1, 2\}$.

Similarly the binary number (expressed in hex)

```
uint64_t taps = 0x6000000000000001
```

indicates that $p_0 = 1$, $p_{62} = p_{61} = 1$ and $p_i = 0$ when $i \notin \{0, 61, 62\}$.

Most of the code you need to implement an LFSR has already been written. There are a few crucial functions left for you to write, which will be described in detail below.

The LFSR L is a struct type defined in the file as follows:

```
typedef struct {
    uint64_t state;
    uint64_t taps;
} LFSR;
```

Thus L has two fields, which are both 64 bit integers. These represent the variables $s_{m-1}, s_{m-2}, \ldots, s_1, s_0$ and $p_{m-1}, p_{m-2}, \ldots, p_1, p_0$ as described above.

Currently the `main()` function has this code:

```
int main()
{
    LFSR L;
    uint64_t initial_state = 0xbeefcafebabecab1;
    uint64_t taps = 0xdeaddeedacedface;
    init_LFSR(&L,initial_state,taps);
    encrypt("toy_pt.txt","ct.txt",&L);
    init_LFSR(&L,initial_state,taps);
    decrypt("ct.txt","toy_ot.txt",&L);
   //get_128_keystream_bits("target_ciphertext","<you fill this in>");
    //shape_keystream();
}
```

The first four lines of this function declare and initialize the state and tap variables of the LFSR. Right now these are initialized with sample values. In the course of decrypting `target_ciphertext`, you will determine the settings used to produce the keystream that encrypted that ciphertext.

The last two lines in this file are commented out, but you should comment them back in as you fill in the functions necessary for them to run. We will describe these functions in detail below.

The code currently in `lfsr.c` should compile and run. There is a chance that there will be problems if the code is compiled on Visual Studio because GNU builtin commands are used in the `parity()` function in `lfsr.c`.

This version of the partity function should work on Windows:

```
int parity(uint64_t n){
/*For use on non-GNU compilers*/
/*Downloaded from https://stackoverflow.com/questions/43883473/working-inline-assembly-in-c-
  n ^= n >> 1;
  n ^= n >> 2;
  n = (n & 0x1111111111111111) * 0x1111111111111111;
  return (n >> 60) & 1;
}
```

You can comment out the provided parity function and replace it with the above if you prefer not to work in GNU.

Completing the LFSR

To complete the code in `lfsr.c` and produce a working LFSR you need to write two functions. These are the `read_lfsr` and `next_state` functions shown below.

```
int read_lfsr(LFSR* L)
```

```
{
/*Return the current output bit (the rightmost bit of the state variable) */


/* You implement this*/

      return 0; // remove this line when you properly implement the function.

}

void next_state(LFSR* L)
{
/*Takes LFSR.
  Returns nothing.
 Side effect: advances L to next state.(shift to the right and replace leftmost bit with appropr
*/

 /* You implement this.
    Hint:  make use of the parity() function provided above*/

}
```

As described in the comments, the `read_lfsr` function should return the rightmost bit of the LFSR `state` variable. Note that `L` is passed as a pointer, so the `state` field must be accessed as `L->state` not `L.state`.

The `next_state` function updates the state of the LFSR according to the value of the tap bits. The simplest way to do this is to AND the state and taps variables, and then return the parity of the result. This works because AND is coordinate-wise multiplication modulo 2, and parity is a summation of bits modulo 2.

As discussed above, you can check the correctness of your code by comparing `ct.dat` and `ct.toy`.

When you have a working LFSR, you can use it to decrypt `target_ciphertext` as soon as you know the initial state and tap settings used to produce that ciphertext.

Recovering the state and tap variables

In order to recover the state and tap settings used to encrypt `target_ciphertext`, we must first use a known-plaintext attack to recover the initial state, $s_0, s_1, \ldots, s_{63}$.

To do this, complete the function `get_128_keystream_bits`, described below.

```
void get_128_keystream_bits(char* ct,char* kpt)
{
/*This function takes 16 bytes of ciphertext and 16 bytes of
  known plaintext.
  The output is a file named "key_stream.txt" that contains 128 bits
  of keystream.  The stream is represented in ASCII, with 128 lines, and
  either a "0" or a "1" on each line.  */

/*This is the plaintext attack in which you recover 2m keystream bits.

      You implement this.
*/

}
```

As discussed in the book, a known plaintext attack works by XORing the known plaintext with the ciphertext (in this case `target_ciphertext`). This exposes a certain amount of the keystream. Because the period of the LFSR is 64, you need $2 \cdot 64/8 = 16$ bytes of keystream. For your 16 bytes of known plaintext, you might want to investigate what the first 16 characters tend to be for books posted as plaintext on Project Gutenberg.

Once you have the first 8 bytes of keystream, you know the initial state of the LFSR, and you can fill in this value back in `main()`.

That is, change

```
    uint64_t initial_state = 0xbeefcafebabecab1;
```

to

```
    uint64_t initial_state = 0xZZZZZZZZZZZZZZZZ;
```

where `0xZZZZZZZZZZZZZZZZ` is the initial state expressed in hex. Recall that s_0 goes in the rightmost bit, and s_{63} goes in the leftmost bit.

All that remains to be done is to recover the inital taps.

Recovering the taps

This task is outlined in detail in the file `matrix.sagews` in this folder. Please read this document.

Grading:

In order to grade this project I will be testing the following:

1. Does “read_lfsr” work
2. Does “next_state” work
3. Does “get_128_keystream_bits” work
4. Does “shape_keystream” work
5. Did you decrypt `target_ciphertext`.

There are many opportunities for partial credit. Please begin early and post questions on Piazza or ask them in class.

Submission:

Please simply leave your files in the `project_2` folder and they will be automatically collected.