Notes Cryptography

Brendon Mendicino

March 20, 2023

Contents

1 Introduction

Ho voglia di piangere

Definition 1.1 – Modulo Operator

$$a = b \pmod{n} : b = n \cdot q + a$$
$$a, b, q, n \in \mathbb{Z}$$

Definition 1.2 – Congurnce Modulo n

$$a \equiv b \pmod{n}$$
$$a \pmod{n} = b \pmod{n}$$

Example 1.1

Show that $a \equiv b \pmod{n}$ if and only if n divides $a - b$.

Proof.

$$r = a \pmod{n} \implies a = aq + r, \quad q \in \mathbb{Z}$$
$$a \pmod{n} = b \pmod{n} = x$$
$$b = nq_1 + x, a = nq_2 + x$$
$$\frac{b-a}{n} = \frac{nq_1 + x - (nq_2 + x)}{n} = q_1 - q_2$$
$$\boxed{q_1, q_2 \in \mathbb{Z} \implies \frac{b-a}{n} \in \mathbb{Z}}$$

□

1.1 Symmetric Cryptography

A symmetric cryptosystem Π consist of three algorithms:

- Decryption
- Encryption
- Generation

Definition 1.3 – IND-secure

A system Π can be defined **IND-secure** if, given two plain-text as inputs (P_0, P_1) to Π and by randomly choosing one of them, there is no better chance of 0.5 to determine whether the ciphertext was generated from P_0 or P_1.

2 Randomness

In mathematics there is no definition for randomness, in a paper published by Lehmer (1951) tries to give an idea of what random numbers can be: *"A pseudo-random is a vague notion embodying the idea of a sequence in which each term is unpredictable and whose digits pass a certain test"*. One the methods involving random number is the **Monte Carlo** used, for example, in the computation of differential equation or the computation of π.

By Lehmar one way of computing a sequence is:

$$u_n = f(u_{n-1}, u_{n-2}, \ldots, u_{n-k});$$

By using recursion it's possible to get the next number.

Definition 2.1 – Lehmar Generator

$$s_0 = \text{seed}$$
$$s_{i+1} \equiv a \cdot s_i + b \pmod{m}, i \in \mathbb{N}$$

Example 2.1 – rand() in ANSI C

$$s_0 = 12345$$
$$s_{i+1} \equiv 1103515245 \cdot s_i + 12345 \pmod{2^{31}}, i \in \mathbb{N}$$

Definition 2.2 – Pseudo Random Generator Function

Where: $\mathbb{Z}_2 = 0, 1$, which is a binary number, $\mathbb{Z}_2^m = [0, 1, \ldots, 0, 1]$ is sequence of bits.
PRNG is defined as:

$$G : \mathbb{Z}_2^n \to \mathbb{Z}_2^{l(n)}, l(n) > n(\text{expansion factor})$$

Such that no adversary can succeed with probability $> 1/2$ if: given a sequence of random numbers that the cryptographer has, by flipping a coin he sends to

the adversary:

- 0: sends a sequence $\mathbb{Z}_2^{l(n)}$ of random bits;
- 1: sends a sequence $\mathbb{Z}_2^{l(n)}$ of bits created with G, staring from $\mathbb{Z}_2^n$ random bits;

If the adversary can guess with probability $> 1/2$ if G function is used then G is **IND-secure**.

2.1 Stream Cipher

There are two operation modes with stream cipher:

- **synchronized mode**:
- **unsynchronized mode**:

Definition 2.3 – Concatenation

$$Enc_k(m_j) = \text{bits of } IV_j || G(k, IV_j, 1^{|m_j|}) \\ < IV_j, G(k, IV_j, 1^{|m_j|}) j >$$

$1^{|m_j|}$ = unary notation, it means passing the length of the sequence of bits (it interpreted in base 1, which means that there are as much 1s as the length of the bits required).

2.2 Linear Feedback Shift Register

In order to construct the bit stream, let's say there are m registers which contain a single bit, at every clock the next bit of the key stream is extracted from the last register, also at every clock every register is shifted to the left. When shifting there is feedback loop that, when shifting, computes the next value of the initial register. One way of accomplishing this is using the **Fibonacci LFSR**.

The output is computed by:

$$s_m = f(\boldsymbol{s}) = \sum_{j=0}^{m-1} s_j \cdot p_j$$

The coefficients are called **taps**, they resemble the way a tap opens or closes.

It's possible to represent the state of the flip-flops at every clock cycle using a matrix:

$$\boldsymbol{L} = \begin{bmatrix} p_{m-1} & 1 & 0 & \dots & 0 \\ p_{m-2} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_1 & 0 & 0 & \dots & 1 \\ p_0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

The registers have all possible state of 2^m, after some time $(2^m - 1)$ that the $\boldsymbol{L}$ matrix is applied, there will come a state such that the sequence of the bits start to loop, the reason is that the set is finate.

Let's image to use the LFSR to generate a cryptographic algorithm: we have a message n-bit long, and the ciphertext is the xor with the sequence produced by the LFSR, the key is shared and contains the initial state of the flip-flop the caracteristic polynomial. But this can be easily broken: the attacker uses KPA (assuming that he knows a segment of the cleartext), from this information he can recover the key. Let's assume $m = 3$, and the key is $k = (\boldsymbol{p})$:

$$\begin{cases} s_3 = s_2 p_2 + s_1 p_1 + s_0 p_0 \\ s_4 = s_3 p_2 + s_2 p_1 + s_1 p_0 \\ s_5 = s_4 p_2 + s_3 p_1 + s_2 p_0 \end{cases}$$

Where $\boldsymbol{p}$ is the unknown variable, this linear equation can be easily solved, to get s_i the attacker knows some part of the cleartext, and he can extract them by $c_i \oplus x_i$, where x_i is known part of the cleartext and the attacker knows *at least* $2m$ consecutive bits.

LFSR should not be used in cryptography but in the past people used a combinations of this generators, called **Geffe Generator**, but also with this kind of generator a KPA is possible.

2.2.1 Galois LSFR

The difference with the Fibonacci one is that in this case there are polynomial multiplications. The state of the flip-flop represent a polynomial. The state is:

$$s(x) = g_0 + g_1 x + \cdots + g_{m-1} x^{m-1}$$

The reason why polynomials are used is that Galois theory is used in many algorithms.

2.3 Permutation

A permutation is defined as function such that:

$$f : \mathbb{Z}_2^n \to \mathbb{Z}_2^n$$

...

3 Openssl

Openssl has two versions 1.x and 3.x and 1.x will be dropped soon, but many applications still use it. The low level software implementations of the algorithm was a big mess, so a layer was created on top of it called *EVP Crypto API*, that just takes in the parameters and does a translation handling all the data.

Typical use of openssl:

1. include libraries
2. load facilities: load the functions required
3. create the context: select the tools, like a certain symmetric algorithm
4. initialize the context: assign IV, nonce, key...
5. operate on the context: provide the data on which the machine will work
6. finalize on the context: perform the concluding operations on the last output, like putting the padding, or the length of the digest
7. free the context: all the objects are *one time objects*, at the end of the operations the objects need to be freed;
8. free facilities

Usually the mode of use of the libraries is the incremental mode, which allow get small blocks a data encrypted.

To get an object the library is called which will return the function pointer to the implementation.

```
1 EVP_CHIPER *c = EVP_bf_cbc();
```

3.1 Symmetric Encryption

The **context** where the data and algorithms are stored are the $\texttt{EVP}_C IPHER_C ONTEXT data structure$.

3.2 Random Numbers

To generate unpredictable sequences of numbers a *Pseudo-Random Number Generator* is used. In OpenSSL this is implemented with a *Deterministic Random Bit Generator*. All PRNG are initialized with a seed. In Linux there is a file: */dev/random* that is an interpolation of different non-deterministic values like: screen data, core temperatures, keystrokes, etc. Altough they are not used for cryptographic algorithms.

OpenSSL implements a *public PRNG* (main one) and a *private PRNG* (ideally used to generate random numbers that will stay private). The functions used to generate the random numbers are:

```
RAND_bytes(buff, len);
RAND_priv_bytes(buff, len);
```

When a seed needs to be initilized OpenSSL provides a simple machanism to do so:

```
int rc = RAND_load_file("/dev/random", 32);
if (rc != 32) {
  // RAND_load_file failed
}
```

Another library to generate PRN is libsodium, which is much smaller than OpenSSL.

It's possible that the functions to load the seed may fail.

3.3 Big Numbers

To represent big numbers is very useful to use the OpenSSL BIGNUM implementation, this is due to the fact that it's very difficoult to operate on large numbers on a modern CPU, in fact the registers are usually 8 byte large, and it's not very simple to implent fast operatino on those numbers.

```
BIGNUM *num;

num = BG_new();

BF_free(num);
```

To copy a BIGNUM:

```
BIGNUM *a, *b, *c;

// ...

BN_copy(a, b);
c = BN_dup(b);
```