HAI709I : Fondements cryptographiques de la sécurité, Université de Montpellier, 2024

30/10/2024. Lecture 4.

1 Basic arithmetic operations

In the class we discussed the fact that the standard algorithms computing

$$\langle a, b\rangle \mapsto a + b,$$

$$\langle a, b\rangle \mapsto a \cdot b,$$

$$\langle a, b, n\rangle \mapsto a + b \mod n,$$

$$\langle a, b, n\rangle \mapsto a \cdot b \mod n,$$

run in time bounded by $\mathrm{poly}(m)$, where m is the length of the input (measured in bits).

We also discussed an efficient algorithm that takes as input a polynomial with integer coefficients

$$P(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_k x^k,$$

and two integer numbers a, n, and returns the value of $P(x)$ at the number a computed modulo n, i.e.,

$$(P(a) \mod n) = c_0 + c_1 a + c_2 a^2 + \ldots + c_k a^k \mod n.$$

The usual method for for polynomial evaluation is so-called *Horner's scheme* that uses the following representation of a polynomial:

$$P(x) = (((\ldots(c_k \cdot x + c_{k-1}) \cdot x + c_{k-2}) \cdot x + \ldots + a_1) \cdot x + a_0$$

So we can evaluate a polynomial as follows:

```
input: c_0,...,c_k, a, n;
res:= c_k mod n;
for (i = k-1; i >= 0; i--) {
    res := res * a + c_i mod n
}
return res.
```

In all these examples we mentioned the *size of the input* of an algorithm, where input consisted of several integer numbers. We usually assume that integer numbers are represented by their binary expansions (e.g., the bit string `101001` stands for the number 41 as $41 = 2^5 + 2^3 + 2^0$). For example, the *size* of an input that contains an integer number n is equal to $\lceil \log n \rceil$, which is the number of binary digits in the binary expansion of n. An efficient algorithm that gets as input a binary expansion of n, cannot afford a loop with n iterations or any other procedures with n steps, this would be way too long (compared with the size of the input $\lceil \log n \rceil$).

It is instructive to observe that the algorithms performing the arithmetic operations modulo n are based on the *Euclidean division* (division with remainder). This well-known algorithm divides one integer (the dividend) by another (the divisor), in a way that produces an integer quotient and a natural number called *remainder* (strictly smaller than the absolute value of the divisor). It is not hard to see that this algorithm runs in polynomial time.

2 Fast exponentiation

In the class we discussed one standard algebraic algorithm — the algorithm of fast exponentiation. This algorithm takes as the input a triple of integer numbers, (a, k, n), and returns the value $a^k \mod n$. The problem of exponentiation may look trivial: we can take the number a, multiply it by itself k times,

$$\underbrace{a \times a \times \ldots \times a}_{k}$$

and the reduce the obtained result modulo n. However, this naive scheme is too "expensive" (it requires too much time and computer memory). Indeed, this procedure requires k operations of multiplication. If the binary expansion of k consists of m binary digits, then the suggested procedure runs in time exponential in m (i.e., exponential in the length of the input). Fortunately, there exists a much more efficient algorithm. We will explain it in two different ways.

The first explanation (adapted for the human perception). We begin with a representation of the number k by its binary expansion, $(k)_2 = \overline{k_m k_{m-1} \ldots k_1 k_0}$, which means that

$$k = k_0 + 2k_1 + 4k_2 + 8k_3 + \ldots + 2^m k_m$$

(each k_i is a binary digit, i.e., either 0 or 1). Then a^k can be represented as follows:

$$\begin{aligned} a^k = a^{k_0} \cdot a^{2k_1} \cdot a^{4k_2} \cdot a^{8k_3} \cdot \ldots \cdot a^{2^m k_m} &= a^{k_0} \cdot \left(a^2\right)^{k_1} \cdot \left(a^4\right)^{k_2} \cdot \left(a^8\right)^{k_3} \cdot \ldots \cdot \left(a^{2^m}\right)^{k_m} \\ &= \prod_{j\,:\,k_j=1} a^{2^j}. \end{aligned}$$

Now it is clear that we can compute $(a^k \mod n)$ in two stages:

(i) Compute sequentially the values $(a^{2^j} \mod n)$ for $j = 1, 2, \ldots, m$. Each next value can be computed as $a^{2^{j+1}} = \left(a^{2^j}\right)^2 \mod n$.

(ii) Compute the product $\prod_{j\,:\,k_j=1} a^{2^j} \mod n$, combining the values a^{2^j} such that $k_j = 1$.

The first stage consists of exactly m multiplications, the second stage consists of at most m multiplications (where $m + 1$ is the number of binary digits in k). Each operation of multiplication modulo n requires $\mathrm{poly}(\log n)$ elementary operation (on each stage we multiply two numbers with at most $\lceil \log_2 n \rceil$ binary digits, then divide the result by n using the Euclidean division algorithm, and take the obtained reminder). Thus, we have $O(\log k)$ stages, and each one can be done in time $\mathrm{poly}(\log n)$.

If the number a is much larger than k and n, then the very first stage can be more expensive: we need to reduce a modulo n, which requires $\mathrm{poly}(\log a, \log n)$ operations (since $\lceil \log_2 a \rceil$ is the number of digits in the standard binary expansion of a). Anyways, the time of computation is polynomial in the total size of the input, which consists of the binary expansions of the numbers a, k, n .

The second explanation (adapted for the computer programming). Substantially the same algorithm of exponentiation can be reformulated as follows:

```
inputs: a, k, n;
z:= 1;
t:= k;
y:= a;
while t>0 {
  if ( t is even )  {
      y:= y * y mod n;
      t:= t/2;
  } else {
      z := z * y mod n;
      t:= t-1;
  }
}
return z.
```

It is easy to see that this algorithm maintains the invariant

$$z \times y^t = a^k \mod p.$$

Indeed, this equality is true just after the initialisation. Then, on each iteration of the loop, we update the values of y, z, t so that the equality remains true. Thus, when the value of t achieves 0, the variable z contains the value $a^k \mod n$.

In this algorithm, the operations

```
y:= y * y mod p;
t:= t/2;
```

are executed $\lceil \log_2 k \rceil$ times. The operations

```
z := z * y mod p;
t:= t-1;
```

are executed as many times as there are 1's in the binary representation of k, i.e., at most $\lceil \log_2 k \rceil$ times. It follows that the algorithm runs in time that polynomially depends on the size of the input (on the number of digits in the numbers a, k, p).

3 Pseudo-random generators and computationally secure schemes

In this section we show that a computationally secure encryption scheme can be constructed with help of pseudo-random generator. We begin with the definition of a pseudo-random generator.

Definition 1. We say that a function

$$F : \{0,1\}^{k(n)} \to \{0,1\}^n$$

is a *pseudo-random generator* if

- $k(n) < n$
- $F(x)$ is computed (by a deterministic algorithm) in time $\mathrm{poly}(n)$
- for every poly-time algorithms Test (deterministic or randomised) the difference

$$\left|\mathrm{Prob}_{x\in_R\{0,1\}^{k(n)}}[\mathrm{Test}(F(x))=1]-\mathrm{Prob}_{y\in_R\{0,1\}^n}[\mathrm{Test}(y)=1]\right|$$

  is negligibly small.

This definition can be interpreted as follows. A pseudo-random generator is a function F that transforms a *seed* x of lenght $k(n)$ in a longer output $y = F(x)$ of length n. If we choose a seed x at random (with a uniform distribution on the set of all strings $\{0,1\}^k$), then the generator induces some probability distribution on the set of values $F(x)$ on the set of strings of length $\{0,1\}^n$. Of course, this distribution is *not* a uniform distribution on $\{0,1\}^n$. However, for an observer with a polynomial computational power this output looks "very similar" to a uniform distribution. This means that if a testing porcedure $\mathrm{Test}()$ tries to distinguish between "good" and "bad" outcomes, than the fractions of "good" and "bad" strings among truly random ones (i.e., $\mathrm{Prob}_{y\in_R\{0,1\}^n}[\mathrm{Test}(y)=1]$) and pseudo-random ones (i.e., $\mathrm{Prob}_{x\in_R\{0,1\}^k}[\mathrm{Test}(x)=1]$) are "almost the same". The word "almost" means that the difference between these probabilities is negligibly small. This condition means that for practical reasons we can use pseudo-random strings instead of truly random ones, and all realisable tests would not see the difference.

Remark 1. The very fact that *pseudo-random generators exist* is highly non-trivial. It is conjectured that they do exist, but this hypothesis remains unproven. This hypothesis is stronger than the famous unproven conjecture $\mathrm{P} \neq \mathrm{NP}$.

Sketch of an encryption scheme secure against any realistic adversary. We can combine pseudo-random generators with Vernam's encryption scheme and obtain a scheme where this size of the key is smaller (possibly, significantly smaller) then the size of a clear message. To this end, we take a pseudo-random generator $F : \{0,1\}^{\ell(n)} \to \{0,1\}^n$ and define an encryption scheme $\Pi = \langle Gen, Enc, Dec\rangle$, where

- the algorithm Gen samples a random seed $s \in \{0,1\}^{\ell(n)}$ and returns a (pseudo)random key $k = F(s)$
- the algorithm $Enc(m,k)$ computes the bitwise XOR of the clear message m and the key k
- the algorithm $Dec(e,k)$ computes the bitwise XOR of the encrypted message e and the key k

In the next lecture we will explain why this scheme is secure (in some natural sense) against any adversary with "realistic" computational resources.