Discrete logarithms and cryptosystems

- Recall that we know now that there are finite fields of all sizes p^n for primes p and integers $n > 0$. We will write F_{p^n} or F_q where q is a power of a prime for these fields from now on.
- The main fact about F_q is that it has a multiplicative generator i.e. a g of multiplicative order $q-1$, and so every non-zero element of F_q has the form g^t for some unique t with $0 \leq t < q$. If $a = g^t$ we will write $L_g(a) = t$ and say that the discrete logarithm of a with respect to (the base) g is t.
- The use of discrete logs in cryptography is based on the idea that the complexity of computing the logarithm in finite fields is high. One advantage of discrete logs over RSA is that one can use fields of size q for large q without having to find large primes. In fact, the prime used is often 2.

Computations in F_{2^n}

- Suppose we represent F_{2^n} as $F_2[x]/(f)$ where f is an irreducible polynomial of degree n.
- Every element of F_{2^n} is then naturally represented as a polynomial of degree less than n, say of the form

$$a_0 + a_1 x + \ldots + a_{n-1} x^{n-1} \text{ where } a_i \in F_2 \text{ for all } i.$$

  This can be more succintly written as an n-tuple $(a_0, a_1, \ldots, a_{n-1}) \in F_2^n$.
- Addition then becomes

$$(a_0, a_1, \ldots, a_{n-1}) + (b_0, b_1, \ldots, b_{n-1}) = (a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1})$$

  where all the calculations are happening in F_2. In fact, these are XOR or "exclusive or" calculations.

Computations in F_{2^n}, cont'd

- Multiplication is also implemented in a computer-friendly way essentially by "right shift": if we want to multiply some n-tuple by x we have two cases: if the polynomial is of degree $< n-1$ then the tuple is of the form $(a_0, \ldots, a_{n-2}, 0)$ and the result is

$$(0, a_0, \ldots, a_{n-2}).$$

Otherwise, the tuple looks like $(a_0, \ldots, a_{n-2}, 1)$ and multiplying by x yields

$$(0, a_0, \ldots, a_{n-2}) + (c_0, c_1, \ldots, c_{n-1})$$

where $f(x) = x^n + c_{n-1}x^{n-1} + \ldots + c_0$.

ElGamal public key cryptosystem

- Taher ElGamal was one of the founders of cryptosystems for internet browsers. He developed what is now known as the ElGamal cryptosystem for Netscape in the 1990's and was responsible for the internet protocol known as SSL or "secure socket layer". His system was based on discrete logarithms. Here is how this works:
- Bob chooses a finite F_q, a multiplicative generator g and a number a. He computes $b = g^a$ and makes the triple (F_q, g, b) public but keeps the knowledge of a, the logarithm, to himself.
- When we say that Bob chooses a field F_q, he actually chooses a particular representation of it in the form $Z_p[x]/(f)$. For instance, he could make the prime p and the polynomial f known publicly. Similarly he gives g and b in the form of a polynomial subject to whatever form he presents F_q.

ElGamal public key cryptosystem, cont'd

- Alice also has to encode her message as a polynomial. If we think as we did with RSA that Alice is encoding a message m as a number less than $q = p^n$ then we can convert m to a polynomial as follows: write

$$m = a_0 + a_1 p + a_2 p^2 + \ldots + a_{n-1} p^{n-1}$$

  where $a_0, \ldots, a_{n-1}$ are integers between 0 and $p - 1$. This is m's base p representation or its p-adic representation.
- Now represent m as the polynomial

$$a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}.$$

- Now that Alice has her message as a polynomial, she picks a random integer k and computes $r = g^k$ and $t = mb^k$ and sends the pair (r, t) to Bob.
- How can Bob decode m from (r, t)? Remember he knows a, the discrete log of b in base g. He calculates

$$tr^{-a} = mb^k g^{-ak} = mg^{ak} g^{-ak} = m \text{ in } F_q.$$