Chiffrierung nach ELGAMAL

Christian Koch
96NKT

14. Januar 2000

Vorgegeben seien eine natürliche Zahl L sowie ein Klartext $\mathscr{P}$. Zu erzeugen sind zunächst eine Primzahl p mit L Bits, so daß auch $(p-1)/2$ eine Primzahl ist, sowie ein primitives Element $\alpha \in \mathbf{Z}_p^*$. Weiterhin ist ein a mit $2 \leq a \leq p-2$ als geheimer Schlüssel auszuwählen.

Der Klartext $\mathscr{P}$ ist in Blöcke der Länge $L-1$ Bit zu zerlegen und nach dem ELGAMAL-Verfahren zu verschlüsseln!

Sowohl Ver- als auch Entschlüsselung sind zu programmieren!

Das ELGAMAL-Verfahren ist ein asymmetrisches Chiffrierverfahren, d. h., zur Entschlüsselung ist ein anderer Schlüssel notwendig als zur Verschlüsselung. Derartige Chiffrierverfahren werden aber aufgrund ihrer Rechenintensität nur in hybriden Systemen eingesetzt, so daß damit nur ein symmetrischer Sitzungsschlüssel oder ein MAC geschützt wird.

Entscheidend für die Programmierung ist die effiziente Verarbeitung sehr großer natürlicher Zahlen. ARIBAS bietet hierfür zwar eine sehr gute Unterstützung, jedoch keine Interaktion mit Hilfe einer grafischen Oberfläche.

Nach einer Suche im Internet fiel die Wahl auf die GNU Multiple Precision Library (`http://www.swox.com/gmp/`) in der aktuellen Version 2.0.2. Sie ist in C geschrieben und auch unter MS Windows einsetzbar. Die zugehörige Dokumentation liegt diesem Projekt als HTML-Datei bei.

Die Datei `global.h` definiert die Typen für das Schlüsselpaar.

```
typedef  struct
{
      mpz_t  p;
      mpz_t  alpha;
      mpz_t  beta;
}  PubKey;


typedef  struct
{
      mpz_t  p;
      mpz_t  a;
}  SecKey;
```

`nextstrongprime` findet die nächste starke Primzahl `mpzPrime`, die auf `mpzNumber` folgt.

```
void
nextstrongprime(mpz_t mpzPrime, mpz_t const mpzNumber)
{
    mpz_t mpztemp;

    mpz_set(mpzPrime, mpzNumber);
    if ((mpz_get_ui(mpzPrime) % 2) == 0) {
        mpz_add_ui(mpzPrime, mpzPrime, 1);
        if (mpz_probab_prime_p(mpzPrime, 25) == 1)
            return;                                          10
    }

    mpz_init(mpztemp);
    do {
        mpz_add_ui(mpzPrime, mpzPrime, 2);
        while (mpz_probab_prime_p(mpzPrime, 25) != 1)
            mpz_add_ui(mpzPrime, mpzPrime, 2);
        mpz_sub_ui(mpztemp, mpzPrime, 1);
        mpz_tdiv_q_2exp(mpztemp, mpztemp, 1);
    }                                                        20
    while (mpz_probab_prime_p(mpztemp, 25) != 1);
    mpz_clear(mpztemp);

    return;
}
```

`genkeypair` erzeugt ein Schlüsselpaar, wobei der Modul `p` eine vorgebene Länge `L` und die Eigenschaft einer sog. starken Primzahl besitzt.

```
void
genkeypair(PubKey *pk, SecKey *sk, int const L)
{
    mpz_t mpzMaske;

    mpz_init_set_ui(mpzMaske, 1);
    mpz_mul_2exp(mpzMaske, mpzMaske, L);
    mpz_sub_ui(mpzMaske, mpzMaske, 1);

    srandom(time(NULL));                                    10
    do {
        mpz_random(pk->p, mpz_size(mpzMaske) + 2);
        mpz_and(pk->p, pk->p, mpzMaske);
        nextstrongprime(pk->p, pk->p);
    }
    while (mpz_sizeinbase(pk->p, 2) != L);
    mpz_set(sk->p, pk->p);
```

```
    srandom(time(NULL));
    do {
        do {                                                    20
            mpz_random(pk->alpha, mpz_size(pk->p));
            mpz_and(pk->alpha, pk->alpha, mpzMaske);
        }
        while (mpz_cmp(pk->alpha, pk->p) > 0);

        mpz_powm_ui(pk->beta, pk->alpha, 2, pk->p);
        if (mpz_cmp_ui(pk->beta, 1) == 0)
            continue;
        mpz_sub_ui(pk->beta, pk->p, 1);
        mpz_tdiv_q_2exp(pk->beta, pk->beta, 1);                 30
        mpz_powm(pk->beta, pk->alpha, pk->beta, pk->p);
    }
    while (mpz_cmp_ui(pk->beta, 1) == 0);

    mpz_sub_ui(pk->beta, pk->p, 2);
    do {
        mpz_random(sk->a, mpz_size(pk->p));
        mpz_and(sk->a, sk->a, mpzMaske);
    }
    while (mpz_cmp(sk->a, pk->beta) > 0);                       40
    mpz_clear(mpzMaske);

    mpz_powm(pk->beta, pk->alpha, sk->a, pk->p);

    return;
}
```

`encrypt` verschlüsselt `no_of_blocks` Klartextblöcke `mpzIn_p` unter Verwendung des öffentlichen Schlüssels `pk`.

```
mpz_t *
encrypt(mpz_t * const mpzIn_p, PubKey const pk, int const no_of_blocks)
{
    int i;
    mpz_t mpztemp, mpzMaske, mpzlambda, *mpzOut_p;

    mpzOut_p = (mpz_t *) malloc(2 * no_of_blocks * sizeof (mpz_t));
    if (mpzOut_p == NULL)
        return (mpzOut_p);

    for (i = 0; i < 2 * no_of_blocks; i++)
        mpz_init(mpzOut_p[i]);
```

10

```
    mpz_init_set_ui(mpzMaske, 1);
    mpz_mul_2exp(mpzMaske, mpzMaske, mpz_sizeinbase(pk.p, 2));
    mpz_sub_ui(mpzMaske, mpzMaske, 1);

    mpz_init(mpztemp);
    mpz_init(mpzlambda);
    srandom(time(NULL));
    for (i = 0; i < no_of_blocks; i++) {                              20
        mpz_sub_ui(mpztemp, pk.p, 2);
        do {
            mpz_random(mpzlambda, mpz_size(pk.p));
            mpz_and(mpzlambda, mpzlambda, mpzMaske);
        }
        while (mpz_cmp(mpzlambda, mpztemp) > 0);

        mpz_powm(mpzOut_p[2*i], pk.alpha, mpzlambda, pk.p);
        mpz_powm(mpzOut_p[2*i+1], pk.beta, mpzlambda, pk.p);
        mpz_mul(mpzOut_p[2*i+1], mpzOut_p[2*i+1], mpzIn_p[i]);        30
        mpz_mod(mpzOut_p[2*i+1], mpzOut_p[2*i+1], pk.p);
    }
    mpz_clear(mpztemp);
    mpz_clear(mpzMaske);
    mpz_clear(mpzlambda);

    return mpzOut_p;
}
```

`decrypt` entschlüsselt `no_of_blocks` Geheimtextblöcke `mpzIn_p` unter Verwendung des privaten Schlüssels `sk`.

```
mpz_t *
decrypt(mpz_t * const mpzIn_p, SecKey const sk, int const no_of_blocks)
{
    int i;
    mpz_t *mpzOut_p;

    mpzOut_p = (mpz_t *) malloc(no_of_blocks * sizeof (mpz_t));
    if (mpzOut_p == NULL)
        return (mpzOut_p);

    for (i = 0; i < no_of_blocks; i++)
        mpz_init(mpzOut_p[i]);

    for (i = 0; i < no_of_blocks; i++) {
        mpz_powm(mpzOut_p[i], mpzIn_p[2*i], sk.a, sk.p);
        mpz_invert(mpzOut_p[i], mpzOut_p[i], sk.p);
        mpz_mul(mpzOut_p[i], mpzOut_p[i], mpzIn_p[2*i+1]);
        mpz_mod(mpzOut_p[i], mpzOut_p[i], sk.p);
    }

    return mpzOut_p;
}
```

Dem Programm kann ein Befehlszeilenargument übergeben werden, welches die Zahlenbasis für die Darstellung der Schlüsselparameter vorgibt. Wird das Programm z. B. durch `elgamal 2` gestartet, dann werden alle Angaben im Binärsystem dargestellt. Vorgabe ist aber das Dezimalsystem.

Die Schlüsseldaten können vorgegeben werden, oder es kann auch ein neues Schlüsselpaar einer bestimmten Länge des Moduls p generiert werden. Zum gegenwärtigen Zeitpunkt habe ich die Bitlänge aufgrund der Berechnungsdauer auf 200 begrenzt. Die Daten werden unter Benutzung der eingetragenen Schlüsseldaten ver- oder entschlüsselt.