

Probable prime – Libgcrypt

```
/*
*****
* Generate a prime number (stored in secure memory)
*/
gcry_mpi_t
_gcry_generate_secret_prime (unsigned int nbits,
                             gcry_random_level_t random_level,
                             int (*extra_check)(void*, gcry_mpi_t),
                             void *extra_check_arg)
{
    gcry_mpi_t prime;

    prime = gen_prime (nbits, 1, random_level, extra_check, extra_check_arg);
    progress('\n');
    return prime;
}

static gcry_mpi_t
gen_prime (unsigned int nbits, int secret, int randomlevel,
           int (*extra_check)(void *, gcry_mpi_t), void *extra_check_arg)
{
    gcry_mpi_t prime, ptest, pminus1, val_2, val_3, result;
    int i;
    unsigned int x, step;
    unsigned int count1, count2;
    int *mods;

    /* if ( DBG_CIPHER ) */
    /* log_debug ("generate a prime of %u bits ", nbits ); */

    if (nbits < 16)
        log_fatal ("can't generate a prime with less than %d bits\n", 16);

    mods = xmalloc (no_of_small_prime_numbers * sizeof *mods);
    /* Make nbits fit into gcry_mpi_t implementation. */
    val_2 = mpi_alloc_set_ui( 2 );
    val_3 = mpi_alloc_set_ui( 3 );
    prime = secret? mpi_snew (nbits): mpi_new (nbits);
    result = mpi_alloc_like( prime );
    pminus1= mpi_alloc_like( prime );
    ptest = mpi_alloc_like( prime );
    count1 = count2 = 0;
    for (;;)
    { /* try forever */
        int dotcount=0;

        /* generate a random number */
        _gcry_mpi_randomize( prime, nbits, randomlevel );

        /* Set high order bit to 1, set low order bit to 1. If we are
           generating a secret prime we are most probably doing that
           for RSA, to make sure that the modulus does have the
           requested key size we set the 2 high order bits. */
        mpi_set_highbit (prime, nbits-1);
        if (secret)
            mpi_set_bit (prime, nbits-2);
        mpi_set_bit(prime, 0);

        /* Calculate all remainders. */
        for (i=0; (x = small_prime_numbers[i]); i++ )
            mods[i] = mpi_fdiv_r_ui(NULL, prime, x);

        /* Now try some primes starting with prime. */
        for(step=0; step < 20000; step += 2 )
        {
            /* Check against all the small primes we have in mods. */
            count1++;
            for (i=0; (x = small_prime_numbers[i]); i++ )
            {
                while ( mods[i] + step >= x )
                    mods[i] -= x;
                if ( !(mods[i] + step) )
                    break;
            }
            if ( x )
                continue; /* Found a multiple of an already known prime. */

            mpi_add_ui( ptest, prime, step );

            /* Do a fast Fermat test now. */
            count2++;
            mpi_sub_ui( pminus1, ptest, 1);
            mpi_powm( result, val_2, pminus1, ptest );
            if ( !mpi_cmp_ui( result, 1 ) )
            {
                /* Not composite, perform stronger tests */
                if (is_prime(ptest, 5, &count2 ) )
                {
                    if (!mpi_test_bit( ptest, nbits-1-secret ))
                    {
                        progress('\n');
                        log_debug ("overflow in prime generation\n");
                        break; /* Stop loop, continue with a new prime. */
                    }

                    if (extra_check && extra_check (extra_check_arg, ptest))
                    {
                        /* The extra check told us that this prime is
                           not of the caller's taste. */
                        progress ('/');
                    }
                    else
                    {
                        /* Got it. */
                        mpi_free(val_2);
                        mpi_free(val_3);
                        mpi_free(result);
                        mpi_free(pminus1);
                        mpi_free(prime);
                        xfree(mods);
                        return ptest;
                    }
                }
            }
            if (++dotcount == 10 )
            {
                progress('.');
                dotcount = 0;
            }
        }
        progress(':'); /* restart with a new random value */
    }
}

/*
* Return true if n is probably a prime
*/
static int
is_prime (gcry_mpi_t n, int steps, unsigned int *count)
{
    gcry_mpi_t x = mpi_alloc( mpi_get_nlimbs( n ) );
    gcry_mpi_t y = mpi_alloc( mpi_get_nlimbs( n ) );
    gcry_mpi_t z = mpi_alloc( mpi_get_nlimbs( n ) );
    gcry_mpi_t nminus1 = mpi_alloc( mpi_get_nlimbs( n ) );
    gcry_mpi_t a2 = mpi_alloc_set_ui( 2 );
    gcry_mpi_t q;
    unsigned i, j, k;
    int rc = 0;
    unsigned nbits = mpi_get_nbits( n );

    if (steps < 5) /* Make sure that we do at least 5 rounds. */
        steps = 5;

    mpi_sub_ui( nminus1, n, 1 );

    /* Find q and k, so that n = 1 + 2^k * q . */
    q = mpi_copy ( nminus1 );
    k = mpi_trailing_zeros ( q );
    mpi_tdiv_q_2exp (q, q, k);

    for (i=0 ; i < steps; i++ )
    {
        ++*count;
        if( !i )
        {
            mpi_set_ui( x, 2 );
        }
        else
        {
            _gcry_mpi_randomize( x, nbits, GCRY_WEAK_RANDOM );

            /* Make sure that the number is smaller than the prime and
               keep the randomness of the high bit. */
            if ( mpi_test_bit ( x, nbits-2) )
            {
                mpi_set_highbit ( x, nbits-2); /* Clear all higher bits. */
            }
            else
            {
                mpi_set_highbit( x, nbits-2 );
                mpi_clear_bit( x, nbits-2 );
            }
            gcry_assert (mpi_cmp (x, nminus1) < 0 && mpi_cmp_ui (x, 1) > 0);
        }
        mpi_powm ( y, x, q, n);
        if ( mpi_cmp_ui(y, 1) && mpi_cmp( y, nminus1 ) )
        {
            for ( j=1; j < k && mpi_cmp( y, nminus1 ); j++ )
            {
                mpi_powm(y, y, a2, n);
                if( !mpi_cmp_ui( y, 1 ) )
                    goto leave; /* Not a prime. */
            }
            if (mpi_cmp( y, nminus1 ) )
                goto leave; /* Not a prime. */
        }
        progress('+');
    }
    rc = 1; /* May be a prime. */

leave:
    mpi_free( x );
    mpi_free( y );
    mpi_free( z );
    mpi_free( nminus1 );
    mpi_free( q );
    mpi_free( a2 );

    return rc;
}
```