/tmp/GautCP_bn_add.c	crypto/bn/bn_add.c

/tmp/GautCP_bn_add.c

```
+--  5 lines: Copyright 1995-2018 The OpenSSL Project
 * in the file LICENSE in the source distribution or
 * https://www.openssl.org/source/license.html
 */

#include "internal/cryptlib.h"
#include "bn_local.h"

/* signed add of b to a. */
int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *
{
    int ret, r_neg, cmp_res;

+-- 61 lines: bn_check_top(a);------------------------
    const BN_ULONG *ap, *bp;
    BN_ULONG *rp, carry, t1, t2;

    bn_check_top(a);
    bn_check_top(b);

    if (a->top < b->top) {
        const BIGNUM *tmp;

        tmp = a;
```

crypto/bn/bn_add.c

```
+--  5 lines: Copyright 1995-2018 The OpenSSL Project
 * in the file LICENSE in the source distribution or
 * https://www.openssl.org/source/license.html
 */

#include "internal/cryptlib.h"
#include "bn_local.h"
#include "bn_par.h"

void *bn_add_sub_words_thread(void *ptr) {
    BN_ULONG c;
    add_sub_args *args = (add_sub_args *) ptr;

    const BN_ULONG* ap = args->a;
    const BN_ULONG* bp = args->b;
    BN_ULONG* rp = args->r;
    BN_ULONG min = args->n;

    if (args->type == '+')
        c = bn_add_words(rp, ap, bp, min);
    else if (args->type == '-')
        c = bn_sub_words(rp, ap, bp, min);

    args->carry = c;
    pthread_exit(NULL);
}

void bn_resolve_carry (BN_ULONG carry, add_sub_args*
    int i = 0;
    BN_ULONG t;
    while (carry && i < arg->n) {
        t = arg->r[i];
        t = (t + carry) & BN_MASK2;
        carry = (t < carry);
        arg->r[i] = t;
        i++;
    }
    if(i == arg->n) {
        arg->carry += carry;
    }
}
void bn_resolve_borrow (BN_ULONG borrow, add_sub_args
    int i = 0;
    BN_ULONG t, t1, c = borrow;
    while (c && i < arg->n) {
        t = arg->r[i];
        t1 = (t - c) & BN_MASK2;
        arg->r[i] = t1;

        //check overflow
        c = (t1 > t);
        i++;
    }
    if(i == arg->n) {
        arg->carry += c;
    }
}

/* signed add of b to a. */
int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *
{
    int ret, r_neg, cmp_res;

+-- 61 lines: bn_check_top(a);------------------------
    const BN_ULONG *ap, *bp;
    BN_ULONG *rp, carry, t1, t2;

    bn_check_top(a);
    bn_check_top(b);

    // a must be longer than b, if otherwise, swap
    if (a->top < b->top) {
        const BIGNUM *tmp;

        tmp = a;
```

```
        tmp = a;
        a = b;
        b = tmp;
+-- 8 lines: }-----------------------------------------
    r->top = max;

    ap = a->d;
    bp = b->d;
    rp = r->d;

    carry = bn_add_words(rp, ap, bp, min);
    rp += min;
    ap += min;

    while (dif) {
        dif--;
        t1 = *(ap++);
+-- 32 lines: t2 = (t1 + carry) & BN_MASK2;----------
        return 0;

    ap = a->d;
    bp = b->d;
    rp = r->d;

    borrow = bn_sub_words(rp, ap, bp, min);
```

```
        tmp = a;
        a = b;
        b = tmp;
+-- 8 lines: }-----------------------------------------
    r->top = max;

    ap = a->d;
    bp = b->d;
    rp = r->d;

    // thread init
    pthread_t thr[NUM_THREADS];
    int rc;

    /* create a thread_data_t argument array */
    add_sub_args thr_data[NUM_THREADS];

    /* create threads, divide array */
    int new_n = min/NUM_THREADS;
    int l_idx = 0;

    for (int i = 0; i < NUM_THREADS; ++i) {
        l_idx = new_n * i;
        // printf("l_idx %d, h_idx %d\n", l_idx, l_id
        thr_data[i].a = &ap[l_idx];
        thr_data[i].b = &bp[l_idx];
        thr_data[i].r = &rp[l_idx];
        thr_data[i].type = '+';

        if (i == (NUM_THREADS - 1))
            thr_data[i].n = new_n + min % NUM_THREADS
        else
            thr_data[i].n = new_n;

        if ((rc = pthread_create(&thr[i], NULL, bn_ad
          fprintf(stderr, "error: pthread_create, rc:
          return EXIT_FAILURE;
        }
    }
    /* block until all threads complete */
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], NULL);
        // printf("t%d %d\n", i, thr_data[i].carry);
    }

    /* Resolve Carry */
    BN_ULONG tmp_carry;
    for (int i = 0; i < NUM_THREADS - 1; ++i) {
        tmp_carry = thr_data[i].carry;
        bn_resolve_carry(tmp_carry, &thr_data[i+1]);
    }
    carry = thr_data[NUM_THREADS-1].carry;
    rp += min;
    ap += min;

    while (dif) {
        dif--;
        t1 = *(ap++);
+-- 32 lines: t2 = (t1 + carry) & BN_MASK2;----------
        return 0;

    ap = a->d;
    bp = b->d;
    rp = r->d;

    // create threads
    pthread_t thr[NUM_THREADS];
    int rc;

    /* create a thread_data_t argument array */
    add_sub_args thr_data[NUM_THREADS];

    /* create threads, divide array */
    int new_n = min/NUM_THREADS;
    int l_idx = 0;

    for (int i = 0; i < NUM_THREADS; ++i) {
```

```
                                                    for (int i = 0; i < NUM_THREADS; ++i) {
                                                        l_idx = new_n * i;
                                                        // printf("l_idx %d, h_idx %d\n", l_idx, l_id
                                                        thr_data[i].a = &ap[l_idx];
                                                        thr_data[i].b = &bp[l_idx];
                                                        thr_data[i].r = &rp[l_idx];
                                                        thr_data[i].type = '-';

                                                        if (i == (NUM_THREADS - 1))
                                                            thr_data[i].n = new_n + min % NUM_THREADS
                                                        else
                                                            thr_data[i].n = new_n;

                                                        if ((rc = pthread_create(&thr[i], NULL, bn_ac
                                                          fprintf(stderr, "error: pthread_create, rc:
                                                          return EXIT_FAILURE;
                                                        }
                                                    }
                                                    /* block until all threads complete */
                                                    for (int i = 0; i < NUM_THREADS; ++i) {
                                                        pthread_join(thr[i], NULL);
                                                        // printf("t%d %d\n", i, thr_data[i].carry);
                                                    }

                                                    /* Resolve Carry */
                                                    BN_ULONG tmp_carry;
                                                    for (int i = 0; i < NUM_THREADS - 1; ++i) {
                                                        tmp_carry = thr_data[i].carry;
                                                        bn_resolve_borrow(tmp_carry, &thr_data[i+1]);
                                                    }
                                                    borrow = thr_data[NUM_THREADS-1].carry;
    ap += min;                                      ap += min;
    rp += min;                                      rp += min;

    while (dif) {                                   while (dif) {
        dif--;                                          dif--;
        t1 = *(ap++);                                   t1 = *(ap++);
+--  8 lines: t2 = (t1 - borrow) & BN_MASK2;---------+--  8 lines: t2 = (t1 - borrow) & BN_MASK2;---------
    r->top = max;                                   r->top = max;
    r->neg = 0;                                     r->neg = 0;
    bn_pollute(r);                                  bn_pollute(r);

    return 1;                                       return 1;
}                                               }
```

/tmp/4UAgEe_bn_exp.c	crypto/bn/bn_exp.c
+-- 6 lines: Copyright 1995-2019 The OpenSSL Project	+-- 6 lines: Copyright 1995-2019 The OpenSSL Project
* https://www.openssl.org/source/license.html	* https://www.openssl.org/source/license.html
*/	*/
#include "internal/cryptlib.h"	#include "internal/cryptlib.h"
#include "internal/constant_time.h"	#include "internal/constant_time.h"
#include "bn_local.h"	#include "bn_local.h"
	#include <math.h>
#include <stdlib.h>	#include <stdlib.h>
	#include "bn_par.h"
#ifdef _WIN32	#ifdef _WIN32
# include <malloc.h>	# include <malloc.h>
# ifndef alloca	# ifndef alloca
# define alloca _alloca	# define alloca _alloca
# endif	# endif
#elif defined(__GNUC__)	#elif defined(__GNUC__)
# ifndef alloca	# ifndef alloca
# define alloca(s) __builtin_alloca((s))	# define alloca(s) __builtin_alloca((s))
# endif	# endif
#elif defined(__sun)	#elif defined(__sun)
# include <alloca.h>	# include <alloca.h>
#endif	#endif
	#ifndef min
	#define min(a,b) (((a) < (b)) ? (a) : (b))
	#endif
#include "rsaz_exp.h"	#include "rsaz_exp.h"
#undef SPARC_T4_MONT	#undef SPARC_T4_MONT
#if defined(OPENSSL_BN_ASM_MONT) && (defined(__sparc_	#if defined(OPENSSL_BN_ASM_MONT) && (defined(__sparc_
# include "sparc_arch.h"	# include "sparc_arch.h"
extern unsigned int OPENSSL_sparcv9cap_P[];	extern unsigned int OPENSSL_sparcv9cap_P[];
+-- 52 lines: # define SPARC_T4_MONT	+-- 52 lines: # define SPARC_T4_MONT
return ret;	return ret;
}	}
int BN_mod_exp(BIGNUM *r, const BIGNUM *a, const BIGN	int BN_mod_exp(BIGNUM *r, const BIGNUM *a, const BIGN
BN_CTX *ctx)	BN_CTX *ctx)
{	{
int ret;	int ret;
bn_check_top(a);	bn_check_top(a);
bn_check_top(p);	bn_check_top(p);
bn_check_top(m);	bn_check_top(m);
+--192 lines:	+--192 lines:
BN_CTX_end(ctx);	BN_CTX_end(ctx);
BN_RECP_CTX_free(&recp);	BN_RECP_CTX_free(&recp);
bn_check_top(r);	bn_check_top(r);
return ret;	return ret;
}	}
int BN_mod_exp_mont(BIGNUM *rr, const BIGNUM *a, cons	int bn_mod_exp_mont_seq(BIGNUM *rr, const BIGNUM *a,
const BIGNUM *m, BN_CTX *ctx, BN_	const BIGNUM *m, BIGNUM **val, I
	int window)
{	{
int i, j, bits, ret = 0, wstart, wend, window, wv	BN_CTX_start(ctx);
int start = 1;	int i, j, bits, ret = 0, wstart, wend, wvalue, st
BIGNUM *d, *r;	BIGNUM *r;
const BIGNUM *aa;	
/* Table of variables obtained from 'ctx' */	
BIGNUM *val[TABLE_SIZE];	
BN_MONT_CTX *mont = NULL;	
if (BN_get_flags(p, BN_FLG_CONSTTIME) != 0	
|| BN_get_flags(a, BN_FLG_CONSTTIME) != (	
|| BN_get_flags(m, BN_FLG_CONSTTIME) != (	
return BN_mod_exp_mont_consttime(rr, a, p, m,	
}	
bn_check_top(a);	r = BN_CTX_get(ctx);
bn_check_top(p);	

```
    bn_check_top(p);
    bn_check_top(m);

    if (!BN_is_odd(m)) {
        BNerr(BN_F_BN_MOD_EXP_MONT, BN_R_CALLED_WITH_
        return 0;
    }
    bits = BN_num_bits(p);
    if (bits == 0) {
        /* x**0 mod 1, or x**0 mod -1 is still zero.
        if (BN_abs_is_word(m, 1)) {
            ret = 1;
            BN_zero(rr);
        } else {
            ret = BN_one(rr);
        }
        return ret;
    }

    BN_CTX_start(ctx);
    d = BN_CTX_get(ctx);
    r = BN_CTX_get(ctx);
    val[0] = BN_CTX_get(ctx);
    if (val[0] == NULL)
        goto err;

    /*
     * If this is not done, things will break in the
     */

    if (in_mont != NULL)
        mont = in_mont;
    else {
        if ((mont = BN_MONT_CTX_new()) == NULL)
            goto err;
        if (!BN_MONT_CTX_set(mont, m, ctx))
            goto err;
    }

    if (a->neg || BN_ucmp(a, m) >= 0) {
        if (!BN_nnmod(val[0], a, m, ctx))
            goto err;
        aa = val[0];
    } else
        aa = a;
    if (!bn_to_mont_fixed_top(val[0], aa, mont, ctx))
        goto err;               /* 1 */

    window = BN_window_bits_for_exponent_size(bits);
    if (window > 1) {
        if (!bn_mul_mont_fixed_top(d, val[0], val[0],
            goto err;           /* 2 */
        j = 1 << (window - 1);
        for (i = 1; i < j; i++) {
            if (((val[i] = BN_CTX_get(ctx)) == NULL)
                !bn_mul_mont_fixed_top(val[i], val[i
                goto err;
        }
    }

    start = 1;                  /* This is used to av
                                 * when there is only
                                 * buffer. */
    wvalue = 0;                 /* The 'value' of the
+-- 61 lines: wstart = bits - 1;          The top bit
        wstart -= wend + 1;
        wvalue = 0;
        start = 0;
        if (wstart < 0)
            break;
    }
```

```
    bits = BN_num_bits(p);
    if (bits == 0) {
        /* x**0 mod 1, or x**0 mod -1 is still zero.
        if (BN_abs_is_word(m, 1)) {
            BN_zero(rr);
            if (!bn_to_mont_fixed_top(rr, rr, mont, c
                goto err;
        } else {
            if (!bn_to_mont_fixed_top(rr, BN_value_or
                goto err;
        }
        ret = 1;
        return ret;
    }

    start = 1;                  /* This is used to av
                                 * when there is only
                                 * buffer. */
    wvalue = 0;                 /* The 'value' of the
+-- 61 lines: wstart = bits - 1;          The top bit
        wstart -= wend + 1;
        wvalue = 0;
        start = 0;
        if (wstart < 0)
            break;
    }
    // printf("rte %s\n", BN_bn2hex(r));
    BN_copy(rr, r);
```

```
    ret = 1;
 err:
    BN_CTX_end(ctx);
    bn_check_top(rr);
    return ret;
}


void* bn_mod_exp_mont_thread(void *ptr) {
    exp_args *args = (exp_args *) ptr;

    BIGNUM *r = args->r;
    const BIGNUM *a = args->a;
    BIGNUM *p = args->p;
    const BIGNUM *m = args->m;
    BN_MONT_CTX *mont = args->mont_ctx;
    BN_ULONG ri = args->ri;
    int window = args->window;
    BIGNUM **val = args->val;
    BN_CTX *ctx = BN_CTX_new();

    // set p
    BN_CTX_start(ctx);

    // printf("%d pta %s\n", ri, BN_bn2hex(p));
    BN_lshift(p, p, ri);
    // printf("ptb %s\n", BN_bn2hex(p));

    //exp
    bn_mod_exp_mont_seq(r, a, p, m, val, ctx, mont, v
    // printf("rt %s\n", BN_bn2hex(r));

    BN_CTX_free(ctx);

    pthread_exit(NULL);
}

void bn_slice(BIGNUM* r, const BIGNUM *a, int ri, int
    // shift right
    BN_rshift(r, a, ri);

    //mask
    BN_mask_bits(r, next_ri - ri);
}

int opt_num_of_thread(int n) {
    float n_f = (float) n;
    float lambda = 1.0 / 3.0;
    float log_lamdba =  log(lambda);
    float log_a = log(1.0 / (1 + 2*n_f*(1 - lambda)))
    float gamma = log_a / log_lamdba;

    return (int) round(gamma + 0.5);
}

int count_partition(int n, int par_num, int nthread)
    int k = nthread;
    float lambda = 1.0 / 3.0;
    float alpha = (float) n / (1.0 - pow(lambda, k));

    float p = alpha*(1-pow(lambda, par_num));

    return (int) p;
}

int BN_mod_exp_mont(BIGNUM *rr, const BIGNUM *a, cons
                    const BIGNUM *m, BN_CTX *ctx, BN_
{

    // printf("exp_mont\n");
    pthread_t thr[NUM_THREADS];
    int rc, i, j, window;
    int bits, ret = 0;
    BIGNUM *d, *r;
    const BIGNUM *aa;
    /* Table of variables obtained from 'ctx' */
```

```
                                                    BIGNUM *val[TABLE_SIZE];
                                                    BN_MONT_CTX *mont = NULL;

                                                    if (BN_get_flags(p, BN_FLG_CONSTTIME) != 0
                                                            || BN_get_flags(a, BN_FLG_CONSTTIME) != 0
                                                            || BN_get_flags(m, BN_FLG_CONSTTIME) != 0
                                                        return BN_mod_exp_mont_consttime(rr, a, p, m,
                                                    }

                                                    bn_check_top(a);
                                                    bn_check_top(p);
                                                    bn_check_top(m);

                                                    if (!BN_is_odd(m)) {
                                                        BNerr(BN_F_BN_MOD_EXP_MONT, BN_R_CALLED_WITH_
                                                        return 0;
                                                    }
                                                    bits = BN_num_bits(p);
                                                    if (bits == 0) {
                                                        /* x**0 mod 1, or x**0 mod -1 is still zero.
                                                        if (BN_abs_is_word(m, 1)) {
                                                            ret = 1;
                                                            BN_zero(rr);
                                                        } else {
                                                            ret = BN_one(rr);
                                                        }
                                                        return ret;
                                                    }

                                                    BN_CTX_start(ctx);
                                                    d = BN_CTX_get(ctx);
                                                    r = BN_CTX_get(ctx);
                                                    val[0] = BN_CTX_get(ctx);
                                                    if (val[0] == NULL)
                                                        goto err;

    /*                                              /*
     * Done with zero-padded intermediate BIGNUMs. Fi  * If this is not done, things will break in the
     * removes padding [if any] and makes return valu
     * API consumer.
     */                                             */
#if defined(SPARC_T4_MONT)
    if (OPENSSL_sparcv9cap_P[0] & (SPARCV9_VIS3 | SPA   if (in_mont != NULL)
        j = mont->N.top;        /* borrow j */              mont = in_mont;
        val[0]->d[0] = 1;       /* borrow val[0] */     else {
        for (i = 1; i < j; i++)                             if ((mont = BN_MONT_CTX_new()) == NULL)
            val[0]->d[i] = 0;
        val[0]->top = j;
        if (!BN_mod_mul_montgomery(rr, r, val[0], mor
            goto err;                                       goto err;
                                                        if (!BN_MONT_CTX_set(mont, m, ctx))
                                                            goto err;
                                                    }

                                                    if (a->neg || BN_ucmp(a, m) >= 0) {
                                                        if (!BN_nnmod(val[0], a, m, ctx))
                                                            goto err;
                                                        aa = val[0];
    } else                                          } else
#endif                                                  aa = a;

                                                    // change aa to montgomery form
                                                    if (!bn_to_mont_fixed_top(val[0], aa, mont, ctx))
                                                        goto err;               /* 1 */

                                                    //precompute val
                                                    window = BN_window_bits_for_exponent_size(bits);
                                                    if (window > 1) {
                                                        if (!bn_mul_mont_fixed_top(d, val[0], val[0],
                                                            goto err;           /* 2 */
                                                        j = 1 << (window - 1);
                                                        for (i = 1; i < j; i++) {
                                                            if (((val[i] = BN_CTX_get(ctx)) == NULL)
                                                                !bn_mul_mont_fixed_top(val[i], val[i
                                                                goto err;
                                                        }
                                                    }
```

```
                                                    if (bits > MIN_BITS_EXP_PARALLEL) {
                                                        /* create a thread_data_t argument array */
                                                        exp_args thr_data[NUM_THREADS];

                                                        /* create threads, divide array */
                                                        int nthread = min(NUM_THREADS, opt_num_of_thr
                                                        int next_ri = 0, ri = count_partition(bits, C
                                                        BIGNUM *rt, *pt;
                                                        for (int i = 0; i < nthread; ++i) {
                                                            if (i == (nthread - 1))
                                                                next_ri = bits;
                                                            else
                                                                next_ri = count_partition(bits, i+1,

                                                            rt = BN_CTX_get(ctx);
                                                            pt = BN_CTX_get(ctx);

                                                            // split p to smaller
                                                            bn_slice(pt, p, ri, next_ri);
                                                            // printf("pi \n"); print_bn(thr_data[i].

                                                            set_exp_arg(thr_data[i], rt, a, pt, m, mc

                                                            if ((rc = pthread_create(&thr[i], NULL, 
                                                              fprintf(stderr, "error: pthread_create,
                                                              return EXIT_FAILURE;
                                                            }
                                                            ri = next_ri;
                                                        }
                                                        if (!bn_to_mont_fixed_top(r, BN_value_one(),
                                                            goto err;
                                                        /* block until all threads complete */
                                                        for (int i = 0; i < nthread; ++i) {
                                                            pthread_join(thr[i], NULL);

                                                            // printf("t%d %s\n", i, BN_bn2hex(thr_da
                                                            if (!bn_mul_mont_fixed_top(r, r, thr_data
                                                                goto err;
                                                        }
                                                    } else {
                                                        bn_mod_exp_mont_seq(r, a, p, m, &(val[0]), ct
                                                    }

    if (!BN_from_montgomery(rr, r, mont, ctx))      if (!BN_from_montgomery(rr, r, mont, ctx))
        goto err;                                       goto err;
    ret = 1;                                        ret = 1;
 err:                                            err:
    if (in_mont == NULL)                            if (in_mont == NULL)
        BN_MONT_CTX_free(mont);                         BN_MONT_CTX_free(mont);
+--659 lines: BN_CTX_end(ctx);----------------- +--659 lines: BN_CTX_end(ctx);-----------------
    return ret;                                     return ret;
}                                               }

int BN_mod_exp_mont_word(BIGNUM *rr, BN_ULONG a, cons int BN_mod_exp_mont_word(BIGNUM *rr, BN_ULONG a, cons
                         const BIGNUM *m, BN_CTX *ctx                            const BIGNUM *m, BN_CTX *ctx
{                                               {
                                                    // printf("exp_mont_word\n");
    BN_MONT_CTX *mont = NULL;                       BN_MONT_CTX *mont = NULL;
    int b, bits, ret = 0;                           int b, bits, ret = 0;
    int r_is_one;                                   int r_is_one;
    BN_ULONG w, next_w;                             BN_ULONG w, next_w;
    BIGNUM *r, *t;                                  BIGNUM *r, *t;
    BIGNUM *swap_tmp;                               BIGNUM *swap_tmp;
+--258 lines: #define BN_MOD_MUL_WORD(r, w, m) \---- +--258 lines: #define BN_MOD_MUL_WORD(r, w, m) \----
```

/tmp/E1utE2_bn_mul.c	crypto/bn/bn_mul.c
+-- 4 lines: Copyright 1995-2018 The OpenSSL Project	+-- 4 lines: Copyright 1995-2018 The OpenSSL Project
* this file except in compliance with the License.	* this file except in compliance with the License.
* in the file LICENSE in the source distribution or	* in the file LICENSE in the source distribution or
* https://www.openssl.org/source/license.html	* https://www.openssl.org/source/license.html
*/	*/
#include <assert.h>	#include <assert.h>
	#include <pthread.h>
#include "internal/cryptlib.h"	#include "internal/cryptlib.h"
#include "bn_local.h"	#include "bn_local.h"
	#include "bn_par.h"
#if defined(OPENSSL_NO_ASM) || !defined(OPENSSL_BN_AS	#if defined(OPENSSL_NO_ASM) || !defined(OPENSSL_BN_AS
/*	/*
* Here follows specialised variants of bn_add_words	* Here follows specialised variants of bn_add_words
* They have the property performing operations on ar	* They have the property performing operations on ar
* The sizes of those arrays is expressed through cl	* The sizes of those arrays is expressed through cl
+--137 lines: * length (basically, min(len(a),len(b)	+--137 lines: * length (basically, min(len(a),len(b)
#ifdef BN_RECURSION	#ifdef BN_RECURSION
/*	/*
* Karatsuba recursive multiplication algorithm (cf.	* Karatsuba recursive multiplication algorithm (cf.
* Computer Programming, Vol. 2)	* Computer Programming, Vol. 2)
*/	*/
	pthread_mutex_t thr_count_lock;
	void *bn_mul_recursive_thread(void *ptr) {
	recursive_args *args = (recursive_args *) ptr;
	BN_ULONG *r = args->r;
	BN_ULONG *a = args->a;
	BN_ULONG *b = args->b;
	int n2 = args->n2;
	int dna = args->dna;
	int dnb = args->dnb;
	BN_ULONG *t = args->t;
	int *used_thr = args->used_thr;
	bn_mul_recursive(r, a, b, n2, dna, dnb, t, used_t
	pthread_exit(NULL);
	}
	void *bn_mul_part_recursive_thread(void *ptr) {
	recursive_args *args = (recursive_args *) ptr;
	BN_ULONG *r = args->r;
	BN_ULONG *a = args->a;
	BN_ULONG *b = args->b;
	int n = args->n2;
	int tna = args->dna;
	int tnb = args->dnb;
	BN_ULONG *t = args->t;
	int *used_thr = args->used_thr;
	bn_mul_part_recursive(r, a, b, n, tna, tnb, t, us
	pthread_exit(NULL);
	}
	void start_mul_recursive_thread(pthread_t *thr, recu
	int rc;
	pthread_mutex_lock(&thr_count_lock);
	(*used_thr)++;
	pthread_mutex_unlock(&thr_count_lock);
	set_recursive_arg((*arg), r, a, b, n2, dna, dnb,
	// printf("thread_created %d\n", *(arg->used_thr)
	if ((rc = pthread_create(thr, NULL, bn_mul_recurs
	fprintf(stderr, "error: pthread_create, rc: %
	exit(EXIT_FAILURE);
	} else {
	// printf("create%d success\n", *used_thr);
	}
	}
	void start_mul_part_recursive_thread(pthread_t *thr

```
                                                    void start_mul_part_recursive_thread(pthread_t *thr,
                                                        int rc;

                                                        pthread_mutex_lock(&thr_count_lock);
                                                        (*used_thr)++;
                                                        pthread_mutex_unlock(&thr_count_lock);
                                                        set_recursive_arg((*arg), r, a, b, n2, dna, dnb,

                                                        // printf("thread_created %d\n", *(arg->used_thr)
                                                        if ((rc = pthread_create(thr, NULL, bn_mul_part_
                                                            fprintf(stderr, "error: pthread_create, rc: %
                                                            exit(EXIT_FAILURE);
                                                        } else {
                                                            // printf("create%d success\n", *used_thr);
                                                        }
                                                    }

                                                    int get_used_thread(int* used_thr) {
                                                        pthread_mutex_lock(&thr_count_lock);
                                                        int u = *used_thr;
                                                        pthread_mutex_unlock(&thr_count_lock);
                                                        return u;
                                                    }
                                                    void set_used_thread(int* used_thr, int new_val) {
                                                        pthread_mutex_lock(&thr_count_lock);
                                                        *used_thr = new_val;
                                                        pthread_mutex_unlock(&thr_count_lock);
                                                    }

 /*-                                                 /*-
  * r is 2*n2 words in size,                         * r is 2*n2 words in size,
  * a and b are both n2 words in size.               * a and b are both n2 words in size.
  * n2 must be a power of 2.                         * n2 must be a power of 2.
  * We multiply and return the result.               * We multiply and return the result.
  * t must be 2*n2 words in size                     * t must be 2*n2 words in size
  * We calculate                                    * We calculate
  * a[0]*b[0]                                       * a[0]*b[0] a_low*b_low
  * a[0]*b[0]+a[1]*b[1]+(a[0]-a[1])*(b[1]-b[0])     * a[0]*b[0]+a[1]*b[1]+(a[0]-a[1])*(b[1]-b[0])
  * a[1]*b[1]                                       *       a_low*b_low + a_high*b_high + (a_low-a_high)
                                                    * a[1]*b[1] a_high*b_high
  */                                                */
 /* dnX may not be positive, but n2/2+dnX has to be */ /* dnX may not be positive, but n2/2+dnX has to be */
 void bn_mul_recursive(BN_ULONG *r, BN_ULONG *a, BN_UL void bn_mul_recursive(BN_ULONG *r, BN_ULONG *a, BN_UL
                       int dna, int dnb, BN_ULONG *t)                       int dna, int dnb, BN_ULONG *t,
 {                                                  {
     int n = n2 / 2, c1, c2;                        int n = n2 / 2, c1, c2;
     int tna = n + dna, tnb = n + dnb;              int tna = n + dna, tnb = n + dnb;
     unsigned int neg, zero;                        unsigned int neg, zero;
     BN_ULONG ln, lo, *p;                           BN_ULONG ln, lo, *p;

 +-- 19 lines: # ifdef BN_MUL_COMBA----------------- +-- 19 lines: # ifdef BN_MUL_COMBA-----------------
         if ((dna + dnb) < 0)                           if ((dna + dnb) < 0)
             memset(&r[2 * n2 + dna + dnb], 0,              memset(&r[2 * n2 + dna + dnb], 0,
                    sizeof(BN_ULONG) * -(dna + dnb));              sizeof(BN_ULONG) * -(dna + dnb));
         return;                                        return;
     }                                              }
     /* r=(a[0]-a[1])*(b[1]-b[0]) */                /* r=(a[0]-a[1])*(b[1]-b[0]) */
     c1 = bn_cmp_part_words(a, &(a[n]), tna, n - tna);  c1 = bn_cmp_part_words(a, &(a[n]), tna, n - tna);
     c2 = bn_cmp_part_words(&(b[n]), b, tnb, tnb - n);  c2 = bn_cmp_part_words(&(b[n]), b, tnb, tnb - n);
     zero = neg = 0;                                zero = neg = 0;
     switch (c1 * 3 + c2) {                         switch (c1 * 3 + c2) {
     case -4:                                       case -4: // a[0] < a[1], b[1] < b[0]
         bn_sub_part_words(t, &(a[n]), a, tna, tna - r      bn_sub_part_words(t, &(a[n]), a, tna, tna - r
         bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, r      bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, r
         break;                                         break;
     case -3:                                       case -3: // a[0] < a[1], b[1] == b[0]
         zero = 1;                                      zero = 1;
         break;                                         break;
     case -2:                                       case -2: // a[0] < a[1], b[1] > b[0]
         bn_sub_part_words(t, &(a[n]), a, tna, tna - r      bn_sub_part_words(t, &(a[n]), a, tna, tna - r
         bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, t      bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, t
         neg = 1;                                       neg = 1;
         break;                                         break;
     case -1:                                       case -1: // a[0] == a[1], b[1] < b[0]
     case 0:                                        case 0: // a[0] == a[1], b[1] =p b[0]
     case 1:                                        case 1: // a[0] == a[1], b[1] > b[0]
         zero = 1;                                      zero = 1;
         break;                                         break;
```

```
        break;
    case 2:
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna
        bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, r
        neg = 1;
        break;
    case 3:
        zero = 1;
        break;
    case 4:
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna
        bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, t
        break;
    }

# ifdef BN_MUL_COMBA
+-- 16 lines: if (n == 4 && dna == 0 && dnb == 0) { )

        bn_mul_comba8(r, a, b);
        bn_mul_comba8(&(r[n2]), &(a[n]), &(b[n]));
    } else
# endif                         /* BN_MUL_COMBA */
    {
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
        p = &(t[n2 * 2]);
        if (!zero)
            bn_mul_recursive(&(t[n2]), t, &(t[n]), n,
        else
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
            memset(&t[n2], 0, sizeof(*t) * n2);
        bn_mul_recursive(r, a, b, n, 0, 0, p);
        bn_mul_recursive(&(r[n2]), &(a[n]), &(b[n]),
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
    }

    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
     */

    c1 = (int)(bn_add_words(t, r, &(r[n2]), n2));
```

```
        break;
    case 2: // a[0] > a[1], b[1] < b[0]
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna
        bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, r
        neg = 1;
        break;
    case 3: // a[0] > a[1], b[1] == b[0]
        zero = 1;
        break;
    case 4: // a[0] > a[1], b[1] > b[0]
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna
        bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, t
        break;
    }

# ifdef BN_MUL_COMBA
+-- 16 lines: if (n == 4 && dna == 0 && dnb == 0) { )

        bn_mul_comba8(r, a, b);
        bn_mul_comba8(&(r[n2]), &(a[n]), &(b[n]));
    } else
# endif                         /* BN_MUL_COMBA */
    {
        if (n2 < MIN_BN_SIZE_MUL_RECURSIVE_PARALLEL)
            set_used_thread(used_thr, 99999);

        pthread_t thr[3];
        recursive_args arg[3];
        int running_cnt = 0, rc;
        BN_ULONG* tp[3];
        p = &(t[n2 * 2]);
        if (!zero) {
            if (get_used_thread(used_thr) < NUM_THREA
                tp[running_cnt] = (BN_ULONG *) calloc
                start_mul_recursive_thread(&(thr[runn
                running_cnt++;
            } else
                bn_mul_recursive(&(t[n2]), t, &(t[n])
        } else
            memset(&t[n2], 0, sizeof(*t) * n2);

        if (get_used_thread(used_thr) < NUM_THREADS)
            tp[running_cnt] = (BN_ULONG *) calloc(n2*
            start_mul_recursive_thread(&(thr[running_
            running_cnt++;
        } else
            bn_mul_recursive(r, a, b, n, 0, 0, p, use

        if (get_used_thread(used_thr) < NUM_THREADS)
            tp[running_cnt] = (BN_ULONG *) calloc(n2*
            start_mul_recursive_thread(&(thr[running_
            running_cnt++;
        } else
            bn_mul_recursive(&(r[n2]), &(a[n]), &(b[r

        /* block until all threads complete */
        // printf("running_cnt %d\n", running_cnt);
        for (int i = 0; i < running_cnt; i++) {
            // printf("i %d\n", i);
            if ((rc = pthread_join(thr[i], NULL))) {
                fprintf(stderr, "error: pthread_join,
                exit(EXIT_FAILURE);
            } else {
                // printf("join%d success\n", i);
            }
            // printf("t%d %d\n", i, thr_data[i].carr
            free(tp[i]);
        }
    }

    /*-
     * t[n2] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the
     * r[0] holds (a[0]*b[0])
     * r[n2] holds (b[1]*b[1])
     */

    c1 = (int)(bn_add_words(t, r, &(r[n2]), n2));
```

```
    if (neg) {                  /* if t[32] is negati
        c1 -= (int)(bn_sub_words(&(t[n2]), t, &(t[n2]
    } else {
        /* Might have a carry */
        c1 += (int)(bn_add_words(&(t[n2]), &(t[n2]),
    }

    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0])+(a[0]*b[0]
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
     * c1 holds the carry bits
     */
    c1 += (int)(bn_add_words(&(r[n]), &(r[n]), &(t[n2
-------------------------------------------------------
-------------------------------------------------------
    if (c1) {
        p = &(r[n + n2]);
        lo = *p;
        ln = (lo + c1) & BN_MASK2;
        *p = ln;

+-- 14 lines: The overflow will stop before we over v

/*
 * n+tn is the word length t needs to be n*4 is size,
 */
/* tnX may not be negative but less than n */
void bn_mul_part_recursive(BN_ULONG *r, BN_ULONG *a,
                           int tna, int tnb, BN_ULONG
{
    int i, j, n2 = n * 2;
    int c1, c2, neg;
    BN_ULONG ln, lo, *p;

    if (n < 8) {
+-- 45 lines: bn_mul_normal(r, a, n + tna, b, n + tnb
    if (n == 8) {
        bn_mul_comba8(&(t[n2]), t, &(t[n]));
        bn_mul_comba8(r, a, b);
        bn_mul_normal(&(r[n2]), &(a[n]), tna, &(b[n])
        memset(&r[n2 + tna + tnb], 0, sizeof(*r) * (r
    } else {
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
        p = &(t[n2 * 2]);
        bn_mul_recursive(&(t[n2]), t, &(t[n]), n, 0,
        bn_mul_recursive(r, a, b, n, 0, 0, p);
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
-------------------------------------------------------
        i = n / 2;
        /*
         * If there is only a bottom half to the numb
         */
        if (tna > tnb)
            j = tna - i;
        else
            j = tnb - i;
        if (j == 0) {
```

```
    if (neg) {                  /* if t[n2] is negati
        c1 -= (int)(bn_sub_words(&(t[n2]), t, &(t[n2]
    } else {
        /* Might have a carry */
        c1 += (int)(bn_add_words(&(t[n2]), &(t[n2]),
    }

    /*-
     * t[n2] holds (a[0]-a[1])*(b[1]-b[0])+(a[0]*b[0]
     * r[0] holds (a[0]*b[0])
     * r[n2] holds (b[1]*b[1])
     * c1 holds the carry bits
     */
    c1 += (int)(bn_add_words(&(r[n]), &(r[n]), &(t[n2
+++ 1 line: 
    // resolve carry on r[n + n2] to last elmt
    if (c1) {
        p = &(r[n + n2]);
        lo = *p;
        ln = (lo + c1) & BN_MASK2;
        *p = ln;

+-- 14 lines: The overflow will stop before we over v

/*
 * n+tn is the word length t needs to be n*4 is size,
 */
/* tnX may not be negative but less than n */
void bn_mul_part_recursive(BN_ULONG *r, BN_ULONG *a,
                           int tna, int tnb, BN_ULONG
{
    int i, j, n2 = n * 2;
    int c1, c2, neg;
    BN_ULONG ln, lo, *p;

    if (n < 8) {
+-- 45 lines: bn_mul_normal(r, a, n + tna, b, n + tnb
    if (n == 8) {
        bn_mul_comba8(&(t[n2]), t, &(t[n]));
        bn_mul_comba8(r, a, b);
        bn_mul_normal(&(r[n2]), &(a[n]), tna, &(b[n])
        memset(&r[n2 + tna + tnb], 0, sizeof(*r) * (r
    } else {
        if (n2 < MIN_BN_SIZE_MUL_RECURSIVE_PARALLEL)
            set_used_thread(used_thr, 99999);


        pthread_t thr[3];
        recursive_args arg[3];
        int running_cnt = 0, rc;
        BN_ULONG* tp[3];
        p = &(t[n2 * 2]);

        if (get_used_thread(used_thr) < NUM_THREADS)
            tp[running_cnt] = (BN_ULONG *) calloc(n2*
            start_mul_recursive_thread(&(thr[running_
            running_cnt++;
        } else
            bn_mul_recursive(&(t[n2]), t, &(t[n]), n,

        if (get_used_thread(used_thr) < NUM_THREADS)
            tp[running_cnt] = (BN_ULONG *) calloc(n2*
            start_mul_recursive_thread(&(thr[running_
            running_cnt++;
        } else
            bn_mul_recursive(r, a, b, n, 0, 0, p, use

        i = n / 2;
        /*
         * If there is only a bottom half to the numb
         */
        if (tna > tnb)
            j = tna - i;
        else
            j = tnb - i;
        if (j == 0) {
```

```
            bn_mul_recursive(&(r[n2]), &(a[n]), &(b[
                             i, tna - i, tnb - i, p);
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
            memset(&r[n2 + i * 2], 0, sizeof(*r) * (r
        } else if (j > 0) {     /* eg, n == 16, i ==
            bn_mul_part_recursive(&(r[n2]), &(a[n]),
                                  i, tna - i, tnb - i
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
            memset(&(r[n2 + tna + tnb]), 0,
                   sizeof(BN_ULONG) * (n2 - tna - tnb
        } else {                /* (j < 0) eg, n == 1

            memset(&r[n2], 0, sizeof(*r) * n2);
            if (tna < BN_MUL_RECURSIVE_SIZE_NORMAL
+--  4 lines: && tnb < BN_MUL_RECURSIVE_SIZE_NORMAL)
                    i /= 2;
                    /*
                     * these simplified conditions wo
                     * difference between tna and tnb
                     */
                    if (i < tna || i < tnb) {
                        bn_mul_part_recursive(&(r[n2]
                                              &(a[n])
                                              i, tna
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
                        break;
                    } else if (i == tna || i == tnb)
                        bn_mul_recursive(&(r[n2]),
                                         &(a[n]), &(b
                                         i, tna - i,
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
                        break;
                    }
                }
            }
        }
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
    }

    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the
     * r[10] holds (a[0]*b[0])
```

```
            if (get_used_thread(used_thr) < NUM_THREA
                tp[running_cnt] = (BN_ULONG *) calloc
                start_mul_recursive_thread(&(thr[runn
                                           i, tna - i, tnb -
                running_cnt++;
            } else
                bn_mul_recursive(&(r[n2]), &(a[n]), &
                                 i, tna - i, tnb -
            memset(&r[n2 + i * 2], 0, sizeof(*r) * (r
        } else if (j > 0) {     /* eg, n == 16, i ==
            if (get_used_thread(used_thr) < NUM_THREA
                tp[running_cnt] = (BN_ULONG *) calloc
                start_mul_recursive_thread(&(thr[runn
                                           i, tna - i, tnb -
                running_cnt++;
            } else
                bn_mul_recursive(&(r[n2]), &(a[n]), &
                                 i, tna - i, tnb -
            memset(&(r[n2 + tna + tnb]), 0,
                   sizeof(BN_ULONG) * (n2 - tna - tnb
        } else {                /* (j < 0) eg, n == 1

            memset(&r[n2], 0, sizeof(*r) * n2);
            if (tna < BN_MUL_RECURSIVE_SIZE_NORMAL
+--  4 lines: && tnb < BN_MUL_RECURSIVE_SIZE_NORMAL)
                    i /= 2;
                    /*
                     * these simplified conditions wo
                     * difference between tna and tnb
                     */
                    if (i < tna || i < tnb) {
                        if (get_used_thread(used_thr)
                            tp[running_cnt] = (BN_ULO
                            start_mul_part_recursive_
                                                  &(a[n]),
                                                  i, tna -
                            running_cnt++;
                        } else
                            bn_mul_part_recursive(&(r
                                                  &(a
                                                  i,
                        break;
                    } else if (i == tna || i == tnb)
                        if (get_used_thread(used_thr)
                            tp[running_cnt] = (BN_ULO
                            start_mul_recursive_threa
                                                  &(a[n]),
                                                  i, tna -
                            running_cnt++;
                        } else
                            bn_mul_recursive(&(r[n2])
                                                  &(a[n]),
                                                  i, tna -
                        break;
                    }
                }
            }
        }

        /* block until all threads complete */
        // printf("running_cnt %d\n", running_cnt);
        for (int i = 0; i < running_cnt; i++) {
            // printf("i %d\n", i);
            if ((rc = pthread_join(thr[i], NULL))) {
                fprintf(stderr, "error: pthread_join,
                exit(EXIT_FAILURE);
            } else {
                // printf("join%d success\n", i);
            }
            // printf("t%d %d\n", i, thr_data[i].carr
            free(tp[i]);
        }
    }

    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the
     * r[10] holds (a[0]*b[0])
```

```
     * r[32] holds (b[1]*b[1])
+-- 41 lines: ----------------------------------------
 * r needs to be n2 words and t needs to be n2*2
 */
void bn_mul_low_recursive(BN_ULONG *r, BN_ULONG *a, B
                          BN_ULONG *t)
{
    int n = n2 / 2;

    bn_mul_recursive(r, a, b, n, 0, 0, &(t[0]));
    if (n >= BN_MUL_LOW_RECURSIVE_SIZE_NORMAL) {
        bn_mul_low_recursive(&(t[0]), &(a[0]), &(b[n]
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
        bn_mul_low_recursive(&(t[0]), &(a[n]), &(b[0]
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
    } else {
+-- 47 lines: bn_mul_low_normal(&(t[0]), &(a[0]), &(b
            goto err;
    } else
        rr = r;

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
    i = al - bl;
----------------------------------------------------------
#endif
#ifdef BN_MUL_COMBA
    if (i == 0) {
# if 0
        if (al == 4) {
            if (bn_wexpand(rr, 8) == NULL)
                goto err;
            rr->top = 8;
            bn_mul_comba4(rr->d, a->d, b->d);
            goto end;
        }
# endif
----------------------------------------------------------
        if (al == 8) {
            if (bn_wexpand(rr, 16) == NULL)
                goto err;
            rr->top = 16;
            bn_mul_comba8(rr->d, a->d, b->d);
            goto end;
        }
    }
#endif                          /* BN_MUL_COMBA */
#ifdef BN_RECURSION
    if ((al >= BN_MULL_SIZE_NORMAL) && (bl >= BN_MULL
        if (i >= -1 && i <= 1) {
----------------------------------------------------------
            /*
             * Find out the power of two lower or equ
             * two numbers
             */
            if (i >= 0) {
                j = BN_num_bits_word((BN_ULONG)al);
            }
            if (i == -1) {
                j = BN_num_bits_word((BN_ULONG)bl);
            }
            j = 1 << (j - 1);
----------------------------------------------------------
            assert(j <= al || j <= bl);
            k = j + j;
            t = BN_CTX_get(ctx);
            if (t == NULL)
                goto err;
            if (al > j || bl > j) {
----------------------------------------------------------
                if (bn_wexpand(t, k * 4) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 4) == NULL)
                    goto err;
                bn_mul_part_recursive(rr->d, a->d, b-
                                      j, al - j, bl -
            } else {            /* al <= j || bl <= j
```

```
     * r[32] holds (b[1]*b[1])
+-- 41 lines: ----------------------------------------
 * r needs to be n2 words and t needs to be n2*2
 */
void bn_mul_low_recursive(BN_ULONG *r, BN_ULONG *a, B
                          BN_ULONG *t)
{
    int n = n2 / 2;
    int u = 99;
    bn_mul_recursive(r, a, b, n, 0, 0, &(t[0]), &u);
    if (n >= BN_MUL_LOW_RECURSIVE_SIZE_NORMAL) {
        bn_mul_low_recursive(&(t[0]), &(a[0]), &(b[n]
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
        bn_mul_low_recursive(&(t[0]), &(a[n]), &(b[0]
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
    } else {
+-- 47 lines: bn_mul_low_normal(&(t[0]), &(a[0]), &(b
            goto err;
    } else
        rr = r;

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
    i = al - bl;
    // printf("i %d, al %d, bl %d\n", i, al, bl);
#endif
#ifdef BN_MUL_COMBA
    if (i == 0) {
# if 0
        if (al == 4) {
            if (bn_wexpand(rr, 8) == NULL)
                goto err;
            rr->top = 8;
            bn_mul_comba4(rr->d, a->d, b->d);
            goto end;
        }
# endif
    // printf("comba\n");
        if (al == 8) {
            if (bn_wexpand(rr, 16) == NULL)
                goto err;
            rr->top = 16;
            bn_mul_comba8(rr->d, a->d, b->d);
            goto end;
        }
    }
#endif                          /* BN_MUL_COMBA */
#ifdef BN_RECURSION
    if ((al >= BN_MULL_SIZE_NORMAL) && (bl >= BN_MULL
        if (i >= -1 && i <= 1) {
            // printf("recursion\n");
            /*
             * Find out the power of two lower or equ
             * two numbers
             */
            if (i >= 0) {
                j = BN_num_bits_word((BN_ULONG)al);
            }
            if (i == -1) {
                j = BN_num_bits_word((BN_ULONG)bl);
            }
            j = 1 << (j - 1);
            // printf("j %d\n", j);
            assert(j <= al || j <= bl);
            k = j + j;
            t = BN_CTX_get(ctx);
            if (t == NULL)
                goto err;
            if (al > j || bl > j) {
                // printf("mul-part-rec\n");
                if (bn_wexpand(t, k * 4) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 4) == NULL)
                    goto err;
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
```

```
                if (bn_wexpand(t, k * 2) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 2) == NULL)
                    goto err;
                bn_mul_recursive(rr->d, a->d, b->d, j
            }
            rr->top = top;
            goto end;
        }
    }
#endif                          /* BN_RECURSION */
    if (bn_wexpand(rr, top) == NULL)
        goto err;
    rr->top = top;
    bn_mul_normal(rr->d, a->d, al, b->d, bl);

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
 end:
#endif
    rr->neg = a->neg ^ b->neg;
+--  5 lines: rr->flags |= BN_FLG_FIXED_TOP;---------
 err:
    bn_check_top(r);
    BN_CTX_end(ctx);
    return ret;
}

void bn_mul_normal(BN_ULONG *r, BN_ULONG *a, int na,
{
    BN_ULONG *rr;

    if (na < nb) {
        int itmp;
        BN_ULONG *ltmp;

        itmp = na;
        na = nb;
        nb = itmp;
        ltmp = a;
        a = b;
        b = ltmp;

    }
    rr = &(r[na]);
    if (nb <= 0) {
        (void)bn_mul_words(r, a, na, 0);
        return;
    } else
        rr[0] = bn_mul_words(r, a, na, b[0]);
+-- 11 lines: for (;;) {----------------------------
        if (--nb <= 0)
            return;
        rr[4] = bn_mul_add_words(&(r[4]), a, na, b[4]
        rr += 4;
        r += 4;
        b += 4;
```

```
                int used_thread = 1;
                bn_mul_part_recursive(rr->d, a->d, b-
                                      j, al - j, bl -
            } else {            /* al <= j && bl <= j
                // al or bl is exacly the power of tw
                if (bn_wexpand(t, k * 2) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 2) == NULL)
                    goto err;
                int used_thread = 1;
                bn_mul_recursive(rr->d, a->d, b->d, j
            }
            rr->top = top;
            goto end;
        }
    }
#endif                          /* BN_RECURSION */
    if (bn_wexpand(rr, top) == NULL)
        goto err;
    rr->top = top;
    // printf("normal\n");
    bn_mul_normal(rr->d, a->d, al, b->d, bl);

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
 end:
#endif
    rr->neg = a->neg ^ b->neg;
+--  5 lines: rr->flags |= BN_FLG_FIXED_TOP;---------
 err:
    bn_check_top(r);
    BN_CTX_end(ctx);
    return ret;
}

void bn_mul_normal_seq(BN_ULONG *r, BN_ULONG *a, int
    BN_ULONG* rr;


    rr = &(r[na]);
    if (nb <= 0) {
        (void)bn_mul_words(r, a, na, 0);
        return;
    } else
        rr[0] = bn_mul_words(r, a, na, b[0]);
+-- 11 lines: for (;;) {----------------------------
        if (--nb <= 0)
            return;
        rr[4] = bn_mul_add_words(&(r[4]), a, na, b[4]
        rr += 4;
        r += 4;
        b += 4;
    }
}

void *bn_mul_normal_thread(void *ptr) {
    mul_normal_args *args = (mul_normal_args *) ptr;

    BN_ULONG* a = args->a;
    BN_ULONG* b = args->b;
    BN_ULONG* r = args->r;
    int na = args->na;
    int nb = args->nb;
    args->nr = na + nb;

    bn_mul_normal_seq(r, a, na, b, nb);
```

```
    pthread_exit(NULL);
}

void print_arr(BN_ULONG *a, int n) {
    for (int i = 0; i < n; i++) {
        printf("%lx\n", a[i]);
    }
}

void bn_mul_normal(BN_ULONG *r, BN_ULONG *a, int na,
{
    if (na < nb) {
        int itmp;
        BN_ULONG *ltmp;

        itmp = na;
        na = nb;
        nb = itmp;
        ltmp = a;
        a = b;
        b = ltmp;
    }

    if (nb > MIN_BN_SIZE_MUL_NORMAL_PARALLEL) {

        memset(r, 0, (na+nb)*sizeof(BN_ULONG));

        pthread_t thr[NUM_THREADS];
        int rc;


        /* create a thread_data_t argument array */
        mul_normal_args thr_data[NUM_THREADS];
        // BN_ULONG* r_tmp[NUM_THREADS];

        /* create threads, divide array */
        int new_nb = nb/NUM_THREADS;
        int l_idx = 0;

        for (int i = 0; i < NUM_THREADS; ++i) {
            if (i == (NUM_THREADS - 1))
                thr_data[i].nb = new_nb + nb % NUM_TH
            else
                thr_data[i].nb = new_nb;

            l_idx = new_nb * i;
            thr_data[i].a = a;
            thr_data[i].b = &(b[l_idx]);
            thr_data[i].na = na;
            thr_data[i].r = (BN_ULONG *) malloc((thr_
            if (thr_data[i].r == NULL) {
                fprintf(stderr, "error: malloc error
                exit(EXIT_FAILURE);
            }

            if ((rc = pthread_create(&thr[i], NULL, 1
              fprintf(stderr, "error: pthread_create,
              exit(EXIT_FAILURE);
            }
        }

        /* block until all threads complete */
        BN_ULONG carry;
        for (int i = 0; i < NUM_THREADS; ++i) {
            pthread_join(thr[i], NULL);

            int nr = thr_data[i].nr;
            carry = bn_add_words(r, r, thr_data[i].r,

            if (i != NUM_THREADS - 1) {
                r[nr] = carry;
            }
            r += thr_data[i].nb;
            free(thr_data[i].r);
        }
    } else { //non parallel
```

```
----------------------------------------------------------
    }
}

void bn_mul_low_normal(BN_ULONG *r, BN_ULONG *a, BN_U
{
    bn_mul_words(r, a, n, b[0]);
+-- 18 lines: for (;;) {----------------------------
```

```
        bn_mul_normal_seq(r, a, na, b, nb);
    }
}

void bn_mul_low_normal(BN_ULONG *r, BN_ULONG *a, BN_U
{
    bn_mul_words(r, a, n, b[0]);
+-- 18 lines: for (;;) {----------------------------
```

/tmp/MHDTdK_bn_sqr.c	crypto/bn/bn_sqr.c
+-- 75 lines: Copyright 1995-2018 The OpenSSL Project	+-- 75 lines: Copyright 1995-2018 The OpenSSL Project
k = j + j;	k = j + j;
if (al == j) {	if (al == j) {
if (bn_wexpand(tmp, k * 2) == NULL)	if (bn_wexpand(tmp, k * 2) == NULL)
goto err;	goto err;
bn_sqr_recursive(rr->d, a->d, al, tmp	bn_sqr_recursive(rr->d, a->d, al, tmp
} else {	} else {
if (bn_wexpand(tmp, max) == NULL)	if (!bn_mul_fixed_top(r, a, a, ctx))
goto err;	goto err;
bn_sqr_normal(rr->d, a->d, al, tmp->d	----------
}	}
}	}
#else	#else
if (bn_wexpand(tmp, max) == NULL)	if (bn_wexpand(tmp, max) == NULL)
goto err;	goto err;
bn_sqr_normal(rr->d, a->d, al, tmp->d);	bn_sqr_normal(rr->d, a->d, al, tmp->d);
+--105 lines: #endif----------	+--105 lines: #endif----------
else	else
memset(&t[n2], 0, sizeof(*t) * n2);	memset(&t[n2], 0, sizeof(*t) * n2);
bn_sqr_recursive(r, a, n, p);	bn_sqr_recursive(r, a, n, p);
bn_sqr_recursive(&(r[n2]), &(a[n]), n, p);	bn_sqr_recursive(&(r[n2]), &(a[n]), n, p);
/*-	/*-
* t[32] holds (a[0]-a[1])*(a[1]-a[0]), it is neg	* t[n2] holds (a[0]-a[1])*(a[1]-a[0]), it is neg
* r[10] holds (a[0]*b[0])	* r[0] holds (a[0]*b[0])
* r[32] holds (b[1]*b[1])	* r[n2] holds (b[1]*b[1])
*/	*/
c1 = (int)(bn_add_words(t, r, &(r[n2]), n2));	c1 = (int)(bn_add_words(t, r, &(r[n2]), n2));
/* t[32] is negative */	/* t[n2] is negative */
c1 -= (int)(bn_sub_words(&(t[n2]), t, &(t[n2]), r	c1 -= (int)(bn_sub_words(&(t[n2]), t, &(t[n2]), r
/*-	/*-
* t[32] holds (a[0]-a[1])*(a[1]-a[0])+(a[0]*a[0]	* t[n2] holds (a[0]-a[1])*(a[1]-a[0])+(a[0]*a[0]
* r[10] holds (a[0]*a[0])	* r[0] holds (a[0]*a[0])
* r[32] holds (a[1]*a[1])	* r[n] holds (a[1]*a[1])
* c1 holds the carry bits	* c1 holds the carry bits
*/	*/
c1 += (int)(bn_add_words(&(r[n]), &(r[n]), &(t[n2	c1 += (int)(bn_add_words(&(r[n]), &(r[n]), &(t[n2
if (c1) {	if (c1) {
p = &(r[n + n2]);	p = &(r[n + n2]);
lo = *p;	lo = *p;
+-- 18 lines: ln = (lo + c1) & BN_MASK2;----------	+-- 18 lines: ln = (lo + c1) & BN_MASK2;----------