Rapport – Semesteroppgave i datasikkerhet

Harald Dahle (795955) og Joakim L. Gilje (796196)

Sammendrag
Oppgaven går ut på å implementere RSA-krypteringen. Deloppgaver for denne krypteringen er å implementere nøkkelgenerator, kryptering og dekryptering. Det å implementere en nøkkelgenerator for RSA-krypteringen krever igjen at det er tilgjengelig en funksjon for å finne primtall.

Valg av verktøy
Da offentlig-nøkkel kryptering involverer bruk av store tall (som går langt utover 32 og 64-bits), har det også vært bruk for funksjonalitet for å jobbe med så store tall. Vi anså det som langt utenfor rammene av oppgaven å implementere et slikt bibliotek. På grunn av dette valgte vi å se etter et bibliotek og funksjoner for å jobbe mot svært store tall fra en tredje part.

Valget falt på libcrypto, som er en del av OpenSSL [1]. Dette ga oss også en fordel da hovedoppgaven til libcrypto er nettopp kryptering, og vi har hatt muligheten til å verifisere f.eks. primtallene funnet ved vår implementasjon mot OpenSSL.

Oppgaven er programmert i C. Utenom libcrypto er det kun benyttet standard C funksjoner. Oppgaven er utviklet og testet på IRIX, Solaris, MacOS X og Linux.

Om RSA
RSA er en algoritme for kryptering og dekryptering. Spesielt for RSA i motsetning tradisjonelle krypteringsalgoritmer er at man har en offentlig og en privat nøkkel. Den offentlige nøkkelen kan gjøres tilgjengelig for resten av verden (også «skumle» mennesker), mens man må se til at den private nøkkelen holdes hemmelig.

RSA nøkkelgenerering består i 5 steg:
1. finn to ulike primtall, p og q
2. finn produktet av p og q, n
3. finn totient, (p-1)(q-1)
4. finn e, hvor e er mindre enn totient og relativt primtall til totient
5. finn d, som er invers av e modulo totient

Offentlig nøkkel består av e og n, mens privat nøkkel består av d og n. Kryptering består da av $C = M^{(e)} \bmod n$, mens dekryptering består av $M = C^{(d)} \bmod n$. Her er M meldingen i klartekst, mens C er den krypterte meldingen (cipher-tekst).

Implementering av oppgaven
For å lære implementasjonen av RSA, har vi brukt Wikipedia [2] og Cryptography and Network Security [3]. Det å generere RSA nøkler krever i tillegg implementering av «Modular Exponentiation», «MillerRabin-test» og «Extended Euclidean Greatest Common Divisor». Her har vi fått hjelp av Wikipedia [4][5], Cryptography and Network Security [6], samt Algorithm Design [7]. RSA kryptering og dekryptering består kun av «Modular Exponentiation».

«Modular Exponentiation» er regnestykker på følgende form: $a^{(b)} \bmod c$. Årsaken til at man benytter en spesiell algoritme her, er for å slippe å regne ut a^b som blir et nesten uendelig stort tall. I stedet «kortes» tallet hele tiden ned ved å trekke modulo-delen inn i regnestykket.

«MillerRabin-test» er en algoritme for å sjekke om et tall kan være primtall. Dette er en av de største usikkerhetene ved hele nøkkelgenereringen da man aldri kan være sikker på at det er et primtall. For et tall n vil det kreve en sjekk på alle tall opp til $\sqrt{n}$ før man kan verifisere at n er et primtall. Da RSA-nøkler med størrelse på 2048 bit genereres ved to 1024-tall, blir det for mange tester å utføre i praksis. «MillerRabin-testen» vil istedet gi beskjed dersom det med sikkerhet kan

sies at tallet er faktoriserbart. For å minke sannsynligheten for å feilaktig godkjenne et faktoriserbart tall som et primtall, blir tallene testet 10 ganger. I gjennomsnitt skal man utføre $0.5\ln(n)$ søk [6].

«Extended Euclidean GCD» er en måte å regne ut den største felles nevner (GCD) mellom to tall. Det brukes under nøkkelgenerering under steg 4 og steg 5 (stegene er også angitt i kildekoden, under funksjonen generateKey(unsigned int)). I steg 4 ønsker en å sjekke at tallet e er relativt primtall til totienten, mens man i steg 5 ønsker å finne et tall d som tilfredstiller formelen $de \equiv 1 \; mod \; totient$. Begge deler kan finnes ved denne algoritmen, man må bare ta hensyn til hvilket tall som skal returneres fra funksjonen.

Resultater
Programmet er et enkelt kommandolinjebasert program. Man kan generere nøkler ved argumentet 'genkey' og ønsket bitlengde for nøkkelen. Programmet vil da generere et offentlig nøkkelpar og lagre offentlig nøkkel i filen pub, mens privat nøkkel lagres i priv. I tidligere versjoner av programmet ble libcrypto også brukt til å verifisere primtallene p og q under nøkkelgenerering.

For å kryptere kaller man opp programmet med argumentet 'encrypt' samt hvilken nøkkel som skal brukes til krypteringen. Kryptert tekst lagres i filen cipher.txt. Tilsvarende for dekryptering kalles programmet med argumentet 'decrypt' samt hvilken nøkkel som skal brukes til dekrypteringen. Dekryptert tekst lagres i filen plain.txt. Programmet vil lese inn data fra stdin, og ofte er det enkleste på en Unix-plattform å bruke en «pipe», |, for å sende data inn til programmet.

Eksempler på bruk
```
# generering av et 2048-bits nøkkelpar
./RSAcrypt genkey 2048

# kryptere filen tekst
cat tekst | ./RSAcrypt encrypt mottakers-offentlige-nøkkel

# dekryptere en mottatt fil mottatt-tekst
cat mottatt-tekst | ./RSAcrypt decrypt min-private-nøkkel
```

Vi har verifisert «Modular Exponentiation», «Extended Euclidean GCD» og «Miller-Rabin test» mot implementasjoner som allerede eksisterer i OpenSSL, og fått bekreftet at disse fungerer korrekt.

Begrensninger av implementeringen
Når man krypterer en tekst blir den krypterte teksten lagret som en heksadesimal streng. Det betyr at den krypterte teksten bruker dobbel så stor plass som nødvendig.

Under dekryptering blir den dekrypterte meldingen lagret som en streng, noe som forhindrer kryptering av binærer.

Disse to tilfellene ville vært trivielle å forbedre, men vi ønsket ikke å komplisere oppgaven unødvendig. Vi har oppnådd hva vi ønsket.

Konklusjon
Programmet fungerer, og vi kan med sikkerhet si at programmet transformerer en klartekst til noe annet. Det er dessverre vanskelig for oss å verifisere en 2048-bits kryptering, men programmet har blitt testet ut fra blant annet eksempler gitt i boken med suksess.

Referanser

[1] OpenSSL: http://www.openssl.org
[2] Wikipedia artikkel om RSA: http://en.wikipedia.org/wiki/RSA
[3] Cryptography and Network Security: kapittel 9
[4] Wikipedia artikkel om Miller-Rabin primtallstest:
http://en.wikipedia.org/wiki/Miller-Rabin_primality_test
[5] Wikipedia artikkel om Extended Euclidean algoritmen:
http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm
[6] Cryptography and Network Security: Kapittel 8
[7] Algorithm Design: Kapittel 10.1

Koden vår ligger under CVS-kontroll på http://jgilje.net/cgi-bin/cvsweb/RSAcrypt/.

Kildekode

```
Revision 1.13, Thu Oct 27 12:29:17 2005 UTC
CVS Tags: HEAD


#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <openssl/bn.h>

BIGNUM* one;
BIGNUM* two;
BN_CTX* BN_context;

/*
        millerRabin-testen skal kjÃ¸res flere ganger
        derfor opprettes det et sett med BN som er felles
        for hvert kall
*/
typedef struct millerRabinBNTemp {
        BIGNUM* d;
        BIGNUM* a;
        BIGNUM* tmp;
        BIGNUM* compare;
        BIGNUM* range;
        BIGNUM* testM1;
        BIGNUM* r;
} millerRabinBNTemp;

/*
        denne koden er presentert flerne steder...
        (bl.a. Figur 9.7, side 273 i CNS)

        b: base, e: exponent, m: modulus
        target: BN hvor resultatet lagres
        r: temp-verdi (raskere kalkulasjon nÃ¥r denne kjÃ¸res ofte)
*/
void modExponentiationBN(BIGNUM* target, BIGNUM* b, BIGNUM* e, BIGNUM* m,
BIGNUM* r) {
        int i;

        BN_set_word(target, 1);

        for (i = BN_num_bits(e); i >= 0; i--) {
                BN_mod_sqr(r, target, m, BN_context);

                if (BN_is_bit_set(e,i)) {
                        BN_mod_mul(target, r, b, m, BN_context);
                } else {
                        BN_copy(target, r);
                }
        }

        return;
}

/*
        initialiser innholdet i struct millerRabinBNTemp
*/
void millerRabinBNInit(millerRabinBNTemp* bnTemp) {
```

```
        bnTemp->d = BN_new();
        bnTemp->a = BN_new();
        bnTemp->tmp = BN_new();
        bnTemp->compare = BN_new();
        bnTemp->range = BN_new();
        bnTemp->testM1 = BN_new();
        bnTemp->r = BN_new();
}

/*
        rydd opp etter millerRabin-testen
*/
void millerRabinBNFree(millerRabinBNTemp* bnTemp) {
        BN_free(bnTemp->d);
        BN_free(bnTemp->a);
        BN_free(bnTemp->tmp);
        BN_free(bnTemp->compare);
        BN_free(bnTemp->range);
        BN_free(bnTemp->testM1);
        BN_free(bnTemp->r);
}

/*
        Dette er en implementasjon av
        algoritmen presentert side 244 i CNS, og
        http://en.wikipedia.org/wiki/Miller-Rabin_primality_test

        test - nummer Ã¥ teste (n i boka)
        times - antall ganger Ã¥ test

        Miller-Rabin testen viser egentlig om et tall med
        sikkerhet kan faktoriseres.  Derfor returnerer testen sann (1)
        hvis tallet test kan faktoriseres.
*/
int millerRabinTestBN(BIGNUM* test, unsigned int times, millerRabinBNTemp*
bnTemp) {
        unsigned int twoExp = 0;
        unsigned int i, j;
        unsigned int ok = 0;

        BN_sub(bnTemp->testM1, test, one);
        BN_copy(bnTemp->d, bnTemp->testM1);

        while (! BN_is_odd(bnTemp->d)) {
                twoExp++;
                BN_rshift1(bnTemp->d, bnTemp->d);
        }

        // selve testen
        for (i = 0; i < times; i++) {
                BN_rand_range(bnTemp->a, test);
                if (BN_is_zero(bnTemp->a))
                        BN_one(bnTemp->a);

                ok = 0;

                modExponentiationBN(bnTemp->compare, bnTemp->a, bnTemp->d, test,
bnTemp->r);
                if (BN_is_one(bnTemp->compare)) {
//                      printf("inconclusive on T1: modExponentiation(a, yarr,
test) == 1\n");
                        continue;
                }
```

```
                for (j = 0; j < twoExp; j++) {
                        BN_set_word(bnTemp->range, (unsigned long) j);
                        BN_exp(bnTemp->tmp, two, bnTemp->range, BN_context);
                        BN_mul(bnTemp->tmp, bnTemp->tmp, bnTemp->d, BN_context);

                        modExponentiationBN(bnTemp->compare, bnTemp->a, bnTemp-
>tmp, test, bnTemp->r);

                        if (! BN_cmp(bnTemp->compare, bnTemp->testM1)) {
//                              printf("inconclusive on T2: modExponentiation(a,
pow(2, j) * yarr, test) == (test - 1)\n");
                                ok = 1;
                                break;
                        }
                }
                if (ok) continue;

//              printf("composite\n");
                return 1;
        }

//      printf("probably prime\n");
        return 0;
}


/*
        et ganske dÃ¥rlig funksjonsnavn...
        dette er en implementasjon av "extended euclidean algorithm"

        relPrime: angir om man tester pÃ¥ om man har funnet to relative primtall
                                (steg 4 i RSA-keygen)
*/
void extGCDBN(BIGNUM* target, BIGNUM* z, BIGNUM* x, int relPrime) {
        BIGNUM *a, *b, *c, *p, *q, *r, *s, *quot, *new_r, *new_s, *temp;
        a = BN_new(); b = BN_new(); c = BN_new(); p = BN_new();
        q = BN_new(); r = BN_new(); s = BN_new(); quot = BN_new();
        new_r = BN_new(); new_s = BN_new();        temp = BN_new();

        BN_copy(a,z);
        BN_copy(b,x);

        BN_one(p);
        BN_one(s);
        BN_zero(q);
        BN_zero(r);

        while (! BN_is_zero(b)) {
                BN_div(quot, c, a, b, BN_context);

                BN_copy(a,b);
                BN_copy(b,c);

                BN_mul(temp,quot,r,BN_context);
                BN_sub(new_r,p,temp);

                BN_mul(temp,quot,s,BN_context);
                BN_sub(new_s,q,temp);

                BN_copy(p,r);
                BN_copy(q,s);
                BN_copy(r,new_r);
                BN_copy(s,new_s);
//              printf("p %s, q %s, r %s, s %s\n", BN_bn2dec(p), BN_bn2dec(q),
```

```
BN_bn2dec(r), BN_bn2dec(s));
        }

        if (relPrime) {
                BN_copy(target, a);
        } else {
                if (q->neg) {
                        BN_add(target,s,q);
                } else {
                        BN_copy(target,q);
                }
        }

        BN_free(a); BN_free(b); BN_free(c); BN_free(p); BN_free(q);
        BN_free(r); BN_free(s); BN_free(quot); BN_free(new_r);
        BN_free(new_s); BN_free(temp);
//      printf("target %s\n", BN_bn2dec(target));
}

/*
        genererer en 'bits' stor RSA-nÃ¸kkel

        privat nÃ¸kkel lagres i filen 'priv'
        offentlig nÃ¸kkel lagres i filen 'pub'
*/
void generateKey(unsigned int bits) {
        BIGNUM* p = BN_new();
        BIGNUM* q = BN_new();

        BIGNUM* n = BN_new();
        BIGNUM* totient = BN_new();

        BIGNUM* d = BN_new();
        BIGNUM* e = BN_new();
        BIGNUM* gcd = BN_new();

        unsigned p_tests = 1;
        unsigned q_tests = 1;

        millerRabinBNTemp bnTemp;
        FILE* key;

        // pÃ¥ de fleste *NIX-plattformer blir random-generatoren til
        // OpenSSL automatisk seeded
        if (! RAND_status()) {
                printf("Error; random-generator has not been seeded\n");
                exit(1);
        }

        // STEG 1
        // generer et tilfeldig oddetall
        if (! BN_rand(p, (bits / 2), 1, 1)) {
                printf("error generating rand p\n");
        }
        if (! BN_rand(q, (bits / 2), 1, 1)) {
                printf("error generating rand q\n");
        }

        millerRabinBNInit(&bnTemp);

        printf("generating p\n");
        while (millerRabinTestBN(p, 10, &bnTemp)) {
                BN_add(p, p, two);
                p_tests++;
```

```
        }
        printf("generating q\n");
        while (millerRabinTestBN(q, 10, &bnTemp)) {
                BN_add(q, q, two);
                q_tests++;
        }

        millerRabinBNFree(&bnTemp);

        printf("p tested %d times, q tested %d times\n", p_tests, q_tests);

        // STEG 2
        BN_mul(n, p, q, BN_context);

        // STEG 3
        BN_sub_word(p, 1UL);
        BN_sub_word(q, 1UL);
        BN_mul(totient, p, q, BN_context);

        // STEG 4
        do {
                BN_set_word(e, 0);
                while (BN_is_one(e) || BN_is_zero(e)) {
                        BN_rand_range(e, totient);
                }

                extGCDBN(gcd, totient, e, 1);
        } while (! BN_is_one(gcd));

        // STEG 5
        extGCDBN(d, totient, e, 0);

        key = fopen("priv", "w");
        BN_print_fp(key, n); fprintf(key, "\n");
        BN_print_fp(key, d); fprintf(key, "\n");
        fclose(key);

        key = fopen("pub", "w");
        BN_print_fp(key, n); fprintf(key, "\n");
        BN_print_fp(key, e); fprintf(key, "\n");
        fclose(key);

        printf("REMEMBER TO PROTECT FILE 'priv'\n");
}

/*
        krypterer stdin, lagrer til cipher.txt
        bruker offentlig nÃ¸kkel fra 'key'
*/
void RSAencrypt(FILE* key) {
        BIGNUM *n, *e, *plain, *cipher, *temp;
        size_t n_size, e_size;
        char *nStr, *eStr;
        unsigned char *plaintext;
        unsigned int textPos = 0;
        unsigned int bitSize = 0;

        FILE* cipherText;

        n_size = 1;
        while (fgetc(key) != 0x0a)
                n_size++;
        e_size = 1;
        while (fgetc(key) != 0x0a)
```

```
                e_size++;

        if (n_size < e_size) {
                printf("error; n_size < e_size\n");
                exit(1);
        }
        bitSize = n_size - 1;

        nStr = malloc(n_size);
        eStr = malloc(e_size);

        rewind(key);
        fread(nStr, n_size, 1, key);
        fread(eStr, e_size, 1, key);
        nStr[bitSize] = 0x0;
        eStr[bitSize] = 0x0;

        n = BN_new();
        e = BN_new();
        plain = BN_new();
        cipher = BN_new();
        temp = BN_new();

        BN_hex2bn(&n, nStr);
        BN_hex2bn(&e, eStr);

        cipherText = fopen("cipher.txt", "w");

        plaintext = malloc(bitSize / 2);
        while (! feof(stdin)) {
                char current = getchar();
                if (feof(stdin)) {
                        plaintext[bitSize  / 2] = 0x0;
                        textPos = bitSize;
//                      printf("end!\n");
                } else {
                        plaintext[textPos] = current;
                }

                textPos++;
                if (textPos >= (bitSize  / 2)) {
                        plaintext[bitSize / 2] = 0x0;
                        BN_bin2bn(plaintext, bitSize / 2, plain);
                        modExponentiationBN(cipher, plain, e, n, temp);
//                      printf("%s => %s\n", BN_bn2hex(plain),
BN_bn2hex(cipher));
                        fprintf(cipherText, "%0*s", bitSize, BN_bn2hex(cipher));

                        textPos = 0;
                        memset(plaintext, 0, bitSize / 2);
                }
        }

        fclose(cipherText);
}

/*
        dekrypterer stdin, lagrer til plain.txt
        bruker privat nÃ¸kkel fra 'key'
*/
void RSAdecrypt(FILE* key) {
        BIGNUM *n, *d, *plain, *cipher, *temp;
        size_t n_size, d_size;
        char *nStr, *dStr;
```

```
        char *ciphertext;
        unsigned char *plaintext;
        unsigned int textPos = 0;
        unsigned int bitSize = 0;

        FILE* plainText;

        n_size = 1;
        while (fgetc(key) != 0x0a)
                n_size++;
        d_size = 1;
        while (fgetc(key) != 0x0a)
                d_size++;

        if (n_size < d_size) {
                printf("error; n_size < d_size\n");
                exit(1);
        }
        bitSize = n_size - 1;

        nStr = malloc(n_size);
        dStr = malloc(d_size);

        rewind(key);
        fread(nStr, n_size, 1, key);
        fread(dStr, d_size, 1, key);
        nStr[bitSize] = 0x0;
        dStr[bitSize] = 0x0;

        n = BN_new();
        d = BN_new();
        plain = BN_new();
        cipher = BN_new();
        temp = BN_new();

        BN_hex2bn(&n, nStr);
        BN_hex2bn(&d, dStr);

        plainText = fopen("plain.txt", "w");

        ciphertext = malloc(bitSize);
        plaintext = malloc(bitSize);

        while (! feof(stdin)) {
                char current = getchar();
                ciphertext[textPos] = current;

                textPos++;
                if (textPos >= (bitSize)) {
                        ciphertext[bitSize] = 0x0;
                        BN_hex2bn(&cipher, ciphertext);
                        modExponentiationBN(plain, cipher, d, n, temp);
//                      printf("%s => %s\n", BN_bn2hex(cipher),
BN_bn2hex(plain));
                        BN_bn2bin(plain, plaintext);
                        fprintf(plainText, "%.*s", bitSize, plaintext);

                        textPos = 0;
                }
        }

        fclose(plainText);
}
```

```
int main (int argc, const char * argv[]) {
        FILE* key;

        one = BN_new();
        two = BN_new();
        BN_context = BN_CTX_new();

        BN_set_word(one, 1UL);
        BN_set_word(two, 2UL);

        if (argc <= 1) {
                printf("Usage: \n"
                           "\t'genkey (bits)': Generate Private/Public\n"
                           "\t'encrypt': encrypts stdin using key in current
dir\n"
                           "\t'decrypt': decrypts stdin using key in current
dir\n");
                return 1;
        }

        if (! strcmp(argv[1], "genkey")) {
                unsigned int bits;
                if (argc <= 2) {
                        printf("Missing keylength\n");
                        return 1;
                }
                bits = strtol(argv[2], NULL, 10);
                printf("Generating %d bits key\n", bits);
                // TODO: sjekk stÃ¸rrelsen av nÃ¸kkel

                generateKey(bits);
                return 0;
        }

        if (! strcmp(argv[1], "encrypt")) {
                if (argc <= 2) {
                        printf("Specify which private keyfile to use\n");
                        return 1;
                }
                key = fopen(argv[2], "r");
                if (! key) {
                        printf("Failed to open keyfile\n");
                        return 1;
                }

                RSAencrypt(key);
                fclose(key);
                return 0;
        }

        if (! strcmp(argv[1], "decrypt")) {
                if (argc <= 2) {
                        printf("Specify which private keyfile to use\n");
                        return 1;
                }
                key = fopen(argv[2], "r");
                if (! key) {
                        printf("Failed to open keyfile\n");
                        return 1;
                }

                RSAdecrypt(key);
                fclose(key);
                return 0;
```

```
        }

    return 0;
}
```