

Assignment 7 / Data Security

Kenan Esau

April 2001

Tutor: Mr. Schmidt

Course: M.Sc Distributed Systems Engineering

Lecturer: Mr. Owens

Contents

1	Introduction	3
2	Simple Ciphers – The Vignère encryption	3
2.1	Index of Coincidence	3
3	Symmetric-Key Encryption	4
3.1	Security of DES	5
3.2	The DES-Feistel Ladder	5
3.3	Feistel-Ladders/Encryption and Decryption	6
4	IDEA	7
4.1	Security of IDEA	8
5	Public-Key Encryption	9
5.1	Proof of RSA	9
5.2	RSA Encryption	11
5.2.1	Creating the Secret Key	11
5.2.2	Encrypting and Decrypting the Message	12
5.3	The Guts of RSA	13
5.4	Breaking RSA	13
6	Public-Key based Protocols – Shamir’s No-Key Protocol	15
7	Message Authentication	16
8	Steganography	17
9	Conclusions	19
A	The program get_key	19
B	The program crypt	22
C	The program prim	26

1 Introduction

This assignment answers the questions related to cryptography from the task description for the workshop called “Coding for Data Compression, Data Security and Error Control”. Different encryption algorithms were modelled with a visual programming environment for cryptography called Cantata.

2 Simple Ciphers – The Vignère encryption

The following sections show the Vignère cipher. It is a polyalphabetic cipher which uses a key word to select from a set of alphabets.

On figure 1 you can see the cantata-model of a Vignère encryption, decryption and a crack. The two reader-glyphs are used to read a pass phrase and a text file. The output of the file-reader is connected to a reader, the encryption glyph and a file statistics glyph. The file statistics are needed to crack the cipher. The encryption glyph hands its output on to a viewer and to the decryption glyph. The Kassiski Test glyph is also connected to the output of the encryption glyph. The Kassiski Test produces a guess of the length of the key used for this cipher. The key length and the file statistics are needed for successful cracking of this cipher.

2.1 Index of Coincidence

If you measure the evenness of a distribution by calculating the sums of the squares of the probabilities of the symbols of an alphabet Σ by the formula:

$$I_c = \sum_{i=1}^n p(\alpha)$$

Where n is the number of symbols in the alphabet – $|\Sigma|$, and α is the probability of a symbol. The result I_c is called Index of Coincidence and can reach from $1/n$ to 1, where $1/n$ means an absolute even distribution and a value of 1 means a completely skewed distribution, where only one of the n possible symbols occurs.

The value of I_c is the probability that two characters are the same if they are chosen at random according to the distribution [1]. By further consideration of the message statistics it is possible to use this measure to obtain the period which is essential for breaking polyalphabetic ciphers like this.

3 Symmetric-Key Encryption

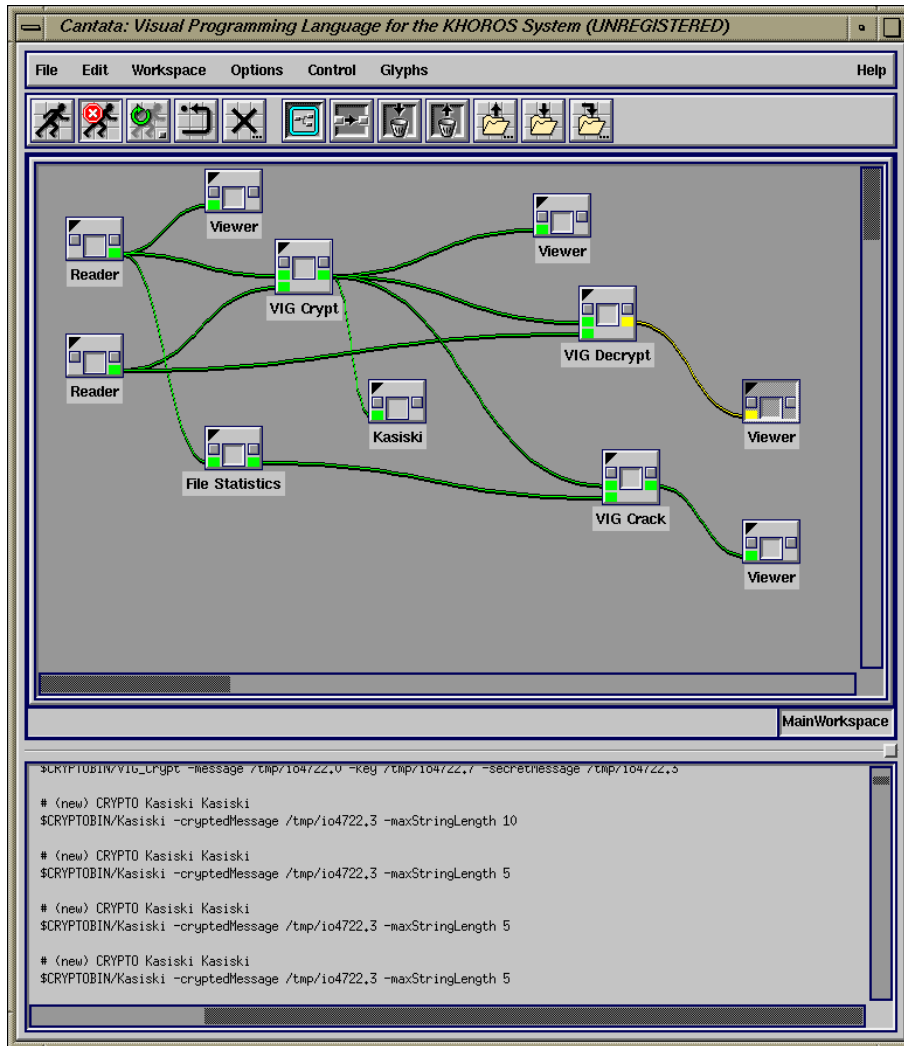


Figure 1: *Vignère encryption and Crack*

To break this cipher with a cipher text only attack you should run a Kasiski test at first to guess the period – the length of the key. If you have got the length of the key you can write the encrypted text to a two-dimensional array with l columns, where l is the length of the key. Now each column contains a string of symbols which was encrypted by a mono alphabetic cipher.

3 Symmetric-Key Encryption

Figure 2 shows the DES-algorithm modeled with Cantata. The Reader is used to read a text file. This data plus the key is presented to the encryption glyph of cantata. The encrypted message can be viewed with the lower of the two viewers. To decrypt the data again, the encryption glyph need the key plus the encrypted data. The decrypted message can be viewed with the upper viewer.

3.1 Security of DES

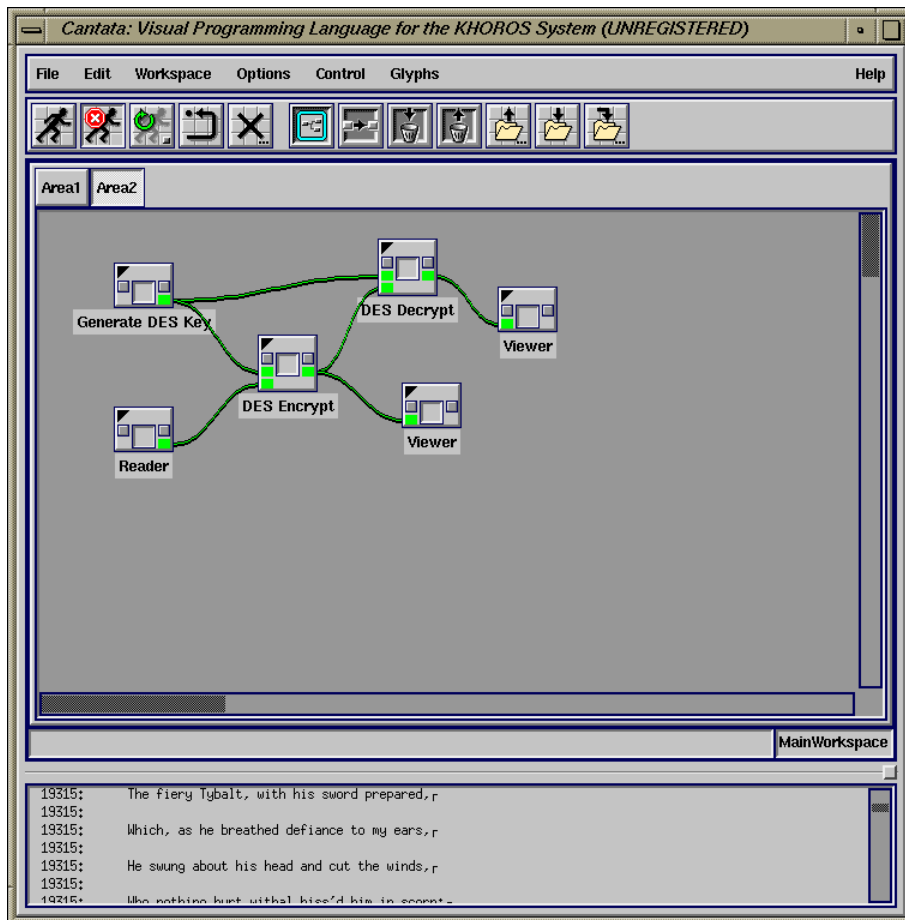


Figure 2: *DES in Cantata*

3.1 Security of DES

How long would it take to do a brute force attack on an DES-key. Since there are 2^{56} possible keys in DES, it would take:

$$\frac{2^{56}}{1 \times 10^{-9} \frac{1}{s}} = 7.206 \times 10^{25} s$$

That is approximately 2.2845×10^{18} years to test all possible keys (worst case). If it is possible to test one key per nanosecond.

3.2 The DES-Feistel Ladder

Figure 3 shows one round of a DES Feistel ladder. The E-Box expands its 32 bit input to 48 bits by permuting the bits and repeating some of them. The main purpose of this is to ensure that each input bit can affect the result of more than one S-Box [1]. This ensures that after a few rounds each output bit depends on every input bit. An additional purpose of the E-Box is to make the output the same size as the key for the XOR-operation. The resulting 48 bit-string can easily be compressed

3.3 Feistel-Ladders/Encryption and Decryption

again by the S-Box.

The S-Box¹ performs some nonlinear transformations on its input. Within the S-Box the 48 bit input is split up into 8 blocks of 6 bit. Each 6 bit block is mapped to a 4 bit result. Thus the output is 32 bit long again. The S-box is the crucial element of the DES-Cipher, most effort in designing DES was spent in designing the S-Boxes.

Finally the P-Box² permutes the result.

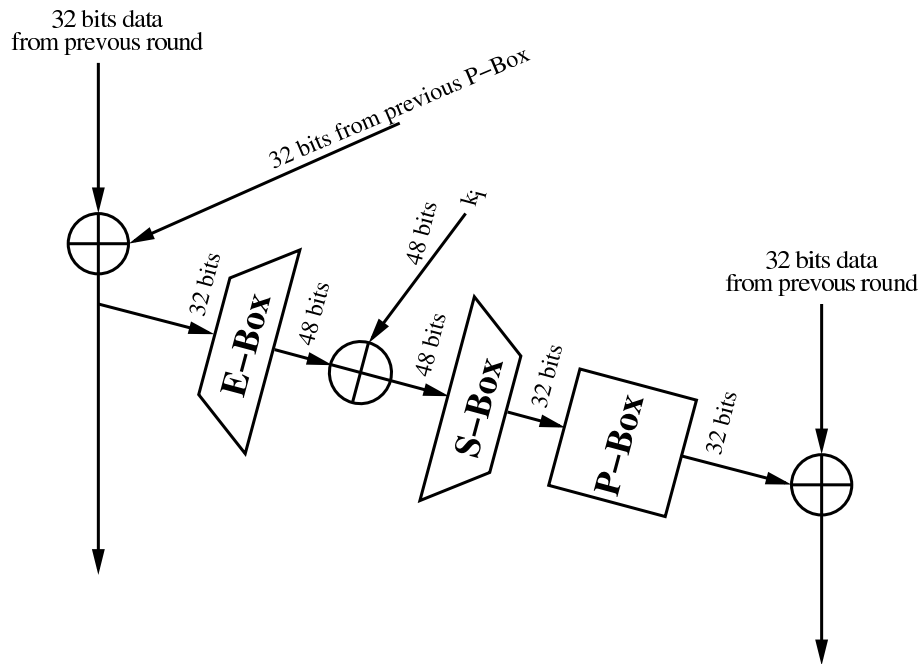


Figure 3: One step of a DES Feistel ladder

3.3 Feistel-Ladders/Encryption and Decryption

For the encryption the following formulas are applied:

$$L_{i+1} = R_i$$

$$R_{i+1} = L_i \oplus f_{S,i}(R_i)$$

The decryption uses the following formulas:

$$R_i = L_{i+1}$$

$$L_i = L_i \oplus f_{S,i}(R_i) = R_{i+1} \oplus f_{S,i}(R_i)$$

¹Substitution Box

²Permutation Box

4 IDEA

IDEA³ is a block oriented cipher which uses 128 bit keys and text blocks of 64 bits. It uses a Feistel ladder which is executed in eight rounds. At the end of the eighth round a transformation is done so that the same algorithm could be used for the decryption.

In figure 4 you can see the cantata model of an IDEA encryption and decryption. The reader-glyph is used to read data from a text file and the BigConst-glyph is used for the key.

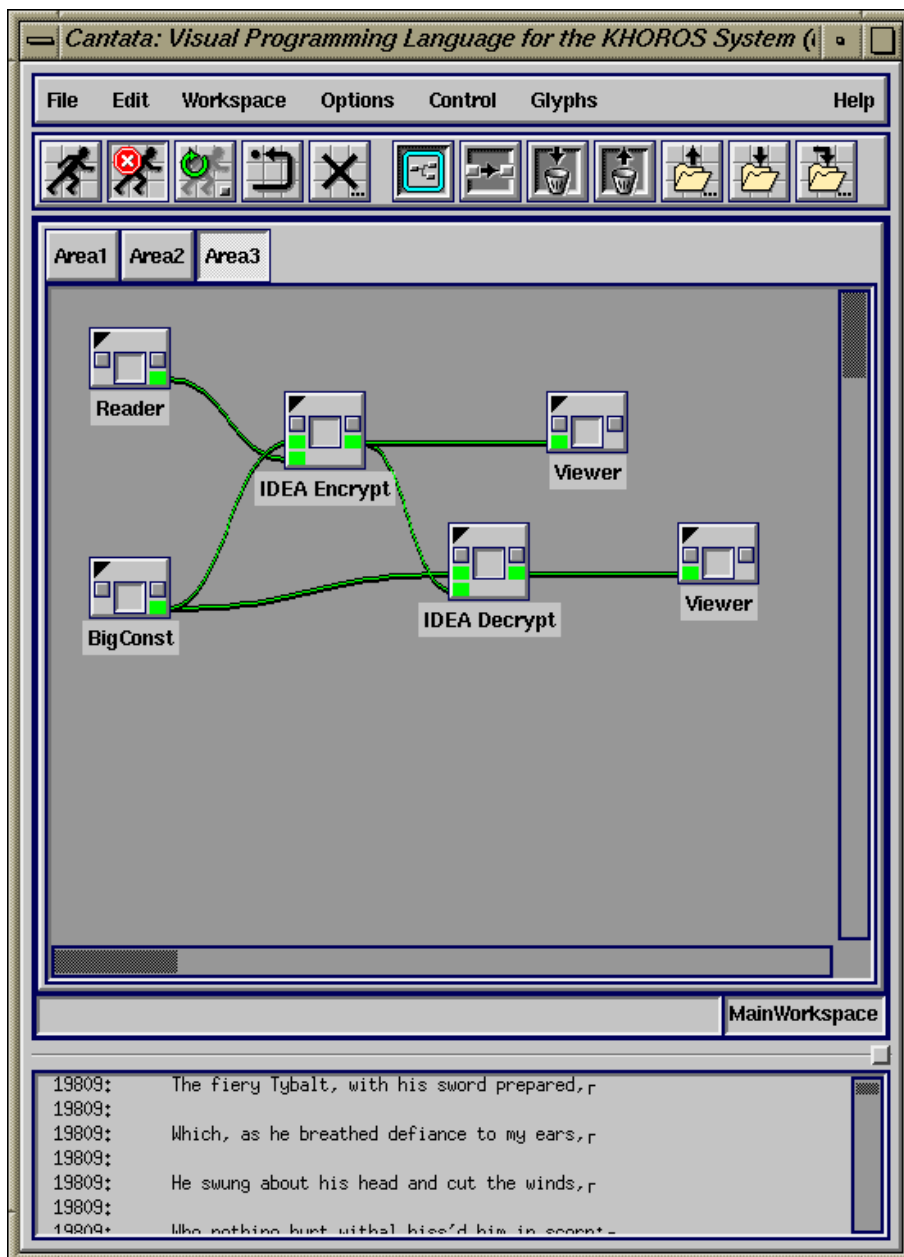


Figure 4: *IDEA in Cantata*

³International Data Encryption Algorithm

4.1 Security of IDEA

4.1 Security of IDEA

Since IDEA uses 128 bit long keys a brute force attack would have to check 2^{128} keys in the worst case. So we can do the same calculation as with DES:

$$\frac{2^{128}}{1 \times 10^{-9\frac{1}{s}}} = 3.403 \times 10^{47} s$$

This is approximately $1.079 * 10^{40}$ years to check all possible keys assuming we can check one key per nanosecond.

5 Public-Key Encryption

Figure 5 shows the model of a RSA encryption and decryption in Cantata. The “Generate RSA-Key”-glyph in the upper left corner produces three numbers p , s and n where $\langle p, n \rangle$ is the public key and $\langle s, n \rangle$ is the private key. $\langle p, n \rangle$ is needed for the decryption and $\langle s, n \rangle$ for the encryption.

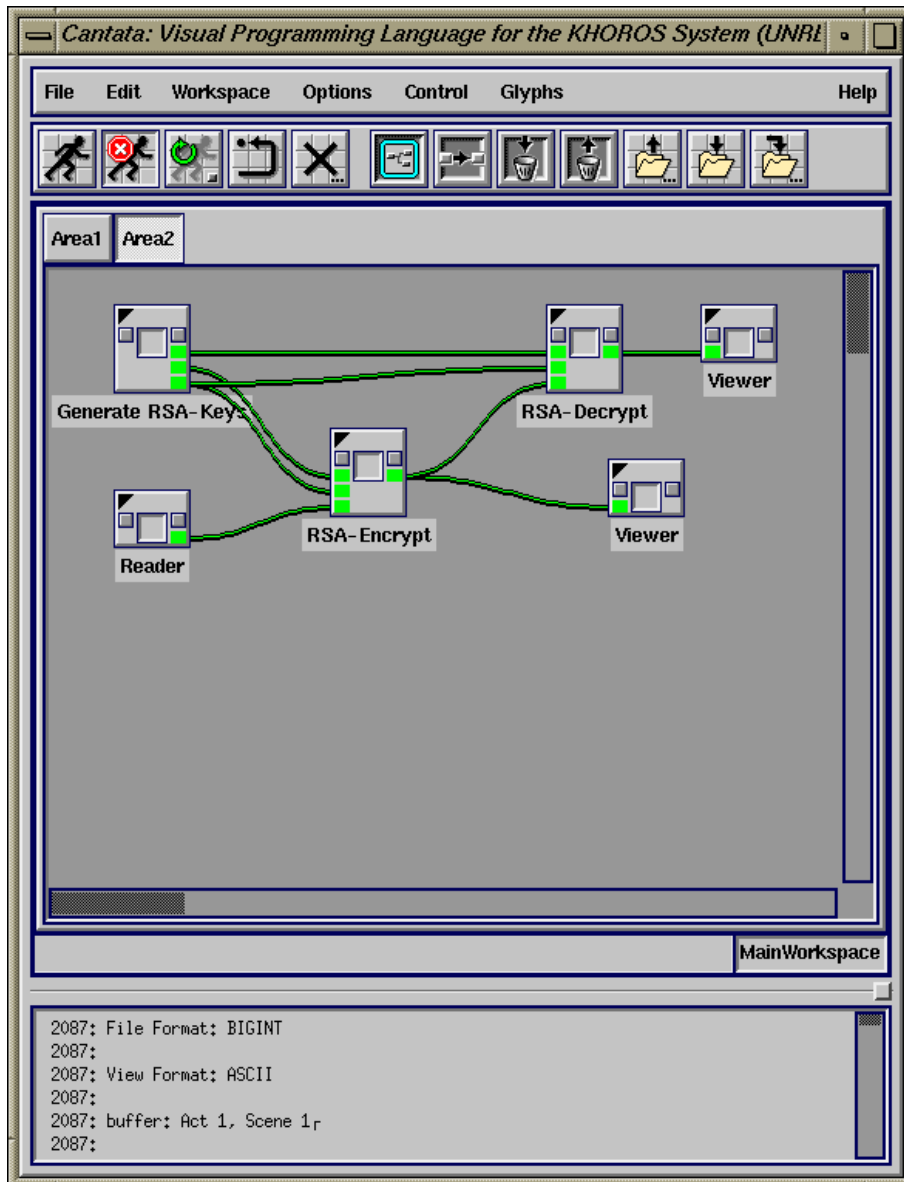


Figure 5: *RSA in Cantata*

5.1 Proof of RSA

1. Randomly select two large primes q and r . Each approximately 512 bits long for a 1024 bit key.

5.1 Proof of RSA

2. Let $n = q \times r$.
3. Randomly choose an integer number p which is coprime to $\phi(n)$, where $\phi(n) = (q - 1)(r - 1)$.
4. Compute s such that $ps \bmod \phi(n) = 1$.

Publish $\langle p, n \rangle$ as your public key. Record $\langle s, n \rangle$ as your private key. The modulus n always has to be greater than your message m you want to encrypt. If $n = 1024$ your message can be $m = 1023$. That is 9 bits, but since you are limited to bytes you can only transmit one byte with this modulus.

If RSA works the following equation has to be fulfilled. If it really works m' has to be equal to the original message m .

$$m' = (m^p \bmod n)^s \bmod n = m^{ps} \bmod n$$

using the Euler equation:

$$\begin{aligned} p \times s &= 1 + \nu\phi(n), \nu \in \mathfrak{S} \\ m' &= m^{ps} \bmod n \\ m' &= m^{1+\nu\phi(n)} \bmod n \\ m' &= m \times (m^{\phi(n)})^\nu \bmod n \end{aligned}$$

At this point you have to differentiate between two cases:

1. The two numbers m and n have no common factors.
Assuming $m < n$. In this case the Euler equation can be used.

$$\begin{aligned} m' &= m \bmod n \times (m^{\phi(n)})^\nu \bmod n \\ m' &= m \times 1^\nu \\ m' &= m \end{aligned}$$

2. If the numbers m and n have common factors. Either m is a multiple of p or it is a multiple of q .

If m is a multiple of p , that is $m = t \times p$ and $t < q$ Using Fermat's Theorem:

$$\begin{aligned} t^{q-1} \bmod q &= 1 \\ p^{p-q} \bmod q &= 1 \end{aligned}$$

and:

$$\begin{aligned} (t^{p-1})^{\nu(q-1)} \bmod q &= 1 \\ (p^{q-1})^{\nu(p-1)} \bmod q &= 1 \end{aligned}$$

5.2 RSA Encryption

If you multiply both equations you get:

$$\begin{aligned}
 (t^{(p-1)\times\nu\times(q-1)} \bmod q) \times (p^{(q-1)\times\nu\times(p-1)} \bmod q) &= 1 \\
 (tp)^{(p-1)\times\nu\times(q-1)} \bmod q &= 1 \\
 t \times t^{(p-1)\times\nu\times(q-1)} \bmod q &= t \\
 pt \times (t^{(p-1)\nu\times(q-1)} \bmod (pq)) &= pt \\
 (tp)^{1+\nu\times(p-1)\times(q-1)} \bmod (pq) &= pt \\
 (tp)^{1+\nu\times\phi(n)} \bmod n &= pt \\
 m^{ps} \bmod n &= pt = m
 \end{aligned}$$

Now it is proven that RSA **REALLY** works.

5.2 RSA Encryption

In this section I will answer question 4 of the task description.

Assume $q = 8101$ and $r = 7951$. The public key value $p = 2047$. The first step is to calculate $\phi(n)$ and n :

$$\phi(n) = (q - 1)(r - 1) = 64395000$$

$$ps \bmod \phi(n) = 1$$

$$n = r \times q = 64411051$$

5.2.1 Creating the Secret Key

To create a secret key you need to calculate a modular inverse. Since the keys have to fulfill the equation $ps \bmod \phi(n) = 1$ you can calculate s since you know all other values.

example with the following values:

$$p = 11, q = 7, r = 13$$

$$\Rightarrow \phi(n) = 27$$

The left most column tracks the result of the Euler equation. The first two rows are to initialize the algorithm (compare to the source code of results are saved (In column 3). The result of the Euler equation has to be related to the key which was tried (Column 4). Every further iteration leads to a lower result of the Euler equation.

The calculation of $11 - 5$ in row three is the same as calculating $66 \times 11 \bmod 72$. If the Euler equation result is 1 the result of the second column shows the secret key.

5.2 RSA Encryption

Rest of Euler	Calculation	Rest	Secret Key
$7 \times 11 \bmod 72 = 5$		5	7
$1 \times 11 \bmod 72 = 11$		11	1
$11 - 5 = 6$	$(1 - 7) \bmod 72 = 66$	6	66
$6 - 5 = 1$	$(66 - 7) \bmod 72 = 59$	1	59

Table 1: Calculation of the secret key

5.2.2 Encrypting and Decrypting the Message

To encrypt the message $m = 1234$ the following calculation has to be performed:

$$c = m^p \bmod n$$

$$c = 1234^{2047} \bmod (8101 \times 7951)$$

Since the numbers get very large during this operation, I chose to implement a program which does this job. At first the key has to be divided into smaller numbers to prevent an overflow.

If we do a binary decomposition of the key we get :

$$2047 = 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 512 + 1024$$

So we could calculate : $1234^{2047} = 1234^{1024} \times 1234^{512} \times \dots \times 1234^1$

A good point to start the search for the secret key is the next odd number greater than $\phi(n)/p$.

Now insert your guess in the Euler equation and save the result as your rest:

But 1234^{1024} is still a very large number. But we could calculate the term on the far right 1234^1 . Under the modulus, it's still 1234. But knowing the result of the term on the right, we can compute the one next to it: Doubling the powers by squaring the numbers, and applying the modulus to intermediate results. This is how my algorithm works. You can find it implemented in the C-function `crypt()` (see ??). Using this approach we get a vector of numbers where. The products of those numbers are equal to 1234^{2047} :

$$1234^{2047} \bmod 2047 = (a \times b \times c \times \dots) \bmod 2047$$

From this result we can do the calculation of the final result by applying the modulus to the intermediate results of the following calculation:

$$\begin{aligned} a \times b \bmod 2047 &= result_0 \\ result_0 \times c \bmod 2047 &= result_1 \\ result_1 \times d \dots & \end{aligned}$$

5.3 The Guts of RSA

The final result is the result of $m^p \bmod n$.

The algorithm described above is implemented in the program `crypt`. It takes three parameters. The message to encrypt/decrypt the power to which the message has to be taken (the key) and the modulo – n .

If you call `crypt` like this: `./crypt 1234 2047 64411051`
You will see a lot of numbers showing the calculations described above.
The last shows the result: `The result is : c = 20479463`

If you do a `./crypt 20479463 49043383 64411051` you get the original message back.

A key like 2048 does not work since it is no prime number. The keys have to be prime numbers since they would not fulfill the requirement of the Euler equation $ps \bmod \phi(n) = 1$

5.3 The Guts of RSA

Figure 6 shows a model of the RSA algorithm using only very basic glyphs of Cantata. With the two BigPrime-glyphs in the upper left corner two big prime numbers are generated (q and r). From these two primes $n = q \times r$ and $\phi(n) = (q - 1)(r - q)$ can be calculated. The BigConst-glyph provides the public key. The reader in the lower left provides a plain text message.

To create a secret key the equation $ps \bmod \phi(n) = 1$ has to be solved by calculating the modular inverse.

To encrypt the message, the BigPow-glyph has to perform the operation $c = m^e$. Where c is the cipher text, m is the plain text and e is the public key.

To decrypt the message, the operation $m = c^s$ has to be performed by the BigPow-glyph on the right. The viewer on the right side should now display the plain text again.

5.4 Breaking RSA

For breaking RSA theoretically you only need the modulus n and the public key p . From this you can calculate the secret key s . The hardest problem is factoring n into its prime factors q and r . If n is long enough – e.g. 1024 bits long or even longer – you will get into serious trouble if you try to find the two prime numbers. For the “small” numbers in Question 5 it is not that long. On my 400MHz CPU it takes 20 to 30 milliseconds to find the primes and to calculate the corresponding secret key. The time depends heavily on q and r . If they are both very close at

5.4 Breaking RSA

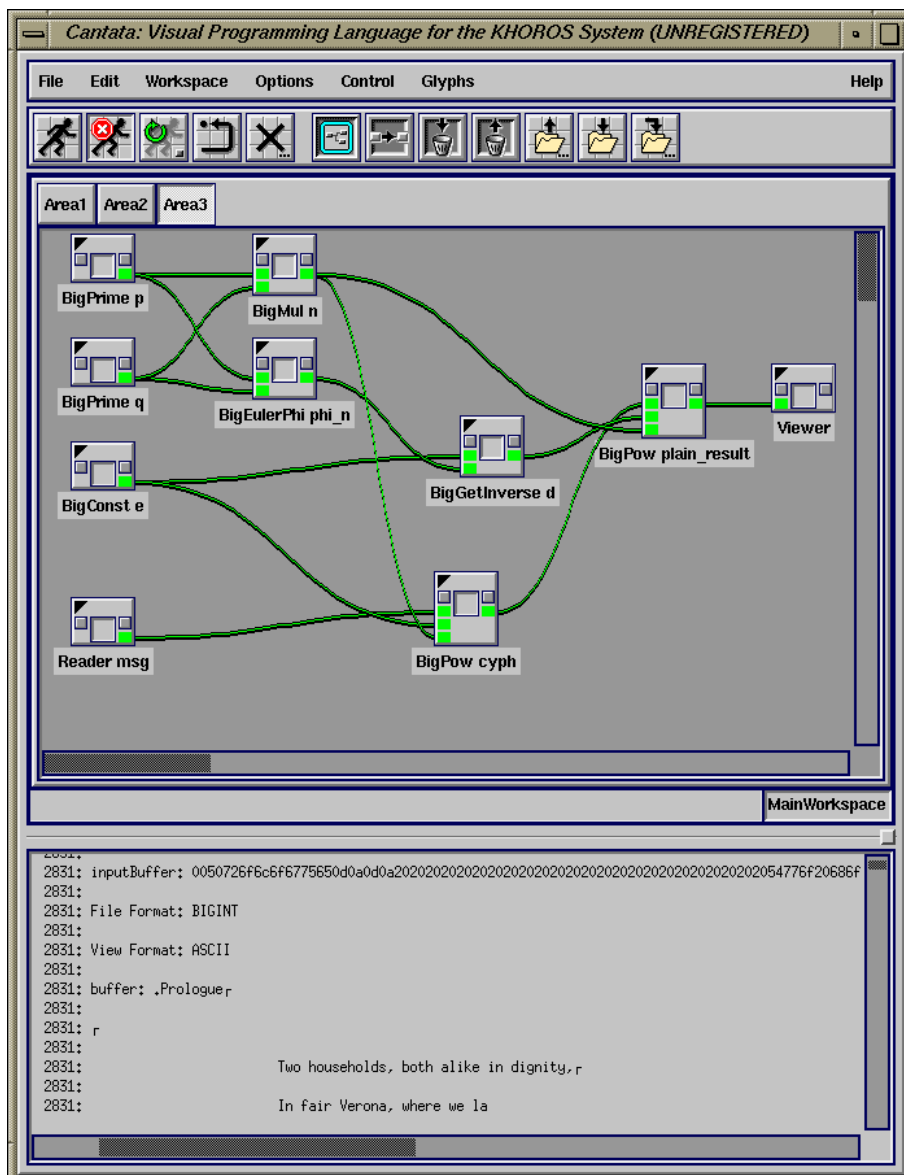


Figure 6: *RSA in Cantata*

the square root of n the algorithm is very fast.

6 Public-Key based Protocols – Shamir’s No-Key Protocol

Figure 7 shows the No-Key protocol which is based on a public key system. It works like this:

1. The two parties (Bob and Alice) agree on a prime p , which is used as the modul for encryption/decryption.
2. Bob and Alice each choose two numbers which have the property:

$$\begin{aligned} e_{Alice} \times d_{Alice} \bmod \phi(p) &= 1 \\ e_{Bob} \times d_{Bob} \bmod \phi(p) &= 1 \end{aligned}$$

The two numbers are secret. On the left side of figure 7 you can see the BigConst-glyphs “e_A” and “e_B” which provide e_{Alice} and e_{Bob} . d_{Alice} and d_{Bob} are created via the two BigGetInverse-Glyphs.

3. Alice encrypts a message using $c = m^{e_{Alice}} \bmod p$ and sends the encrypted message to Bob (The first BigPow-Glyph in figure 7).
4. Bob encrypts the received message again using $c' = c^{e_{Bob}} \bmod p$ and sends c' back to Alice (The second BigPow-Glyph in figure 7).
5. Alice decrypts the received message using $m' = (c')^{d_{Alice}} \bmod p$ and sends the result back to Bob (The third BigPow-Glyph in figure 7).
6. Bob decrypts the received message using $m = (m')^{d_{Bob}} \bmod p$ (The fourth BigPow-Glyph in figure 7). Now Bob can read the message.

That Shamir’s No-Key protocol works can be shown using the Euler-formula:

$$(e \times d) \bmod \phi(p) = 1$$

$$\begin{aligned} &\Rightarrow e \times d = 1 + \nu \times \phi(p) \\ m^{e \times d} \bmod p &= m^{\nu \times \phi(p)} \bmod p \\ m^{e \times d} \bmod p &= (m \bmod p) \times (m^{\nu \times \phi(p)} \bmod p) \\ m^{e \times d} \bmod p &= m \times (m^{\phi(p)})^\nu \bmod p \\ m^{e \times d} \bmod p &= m \times (1)^\nu \bmod p \\ m^{e \times d} \bmod p &= m \end{aligned}$$

7 Message Authentication

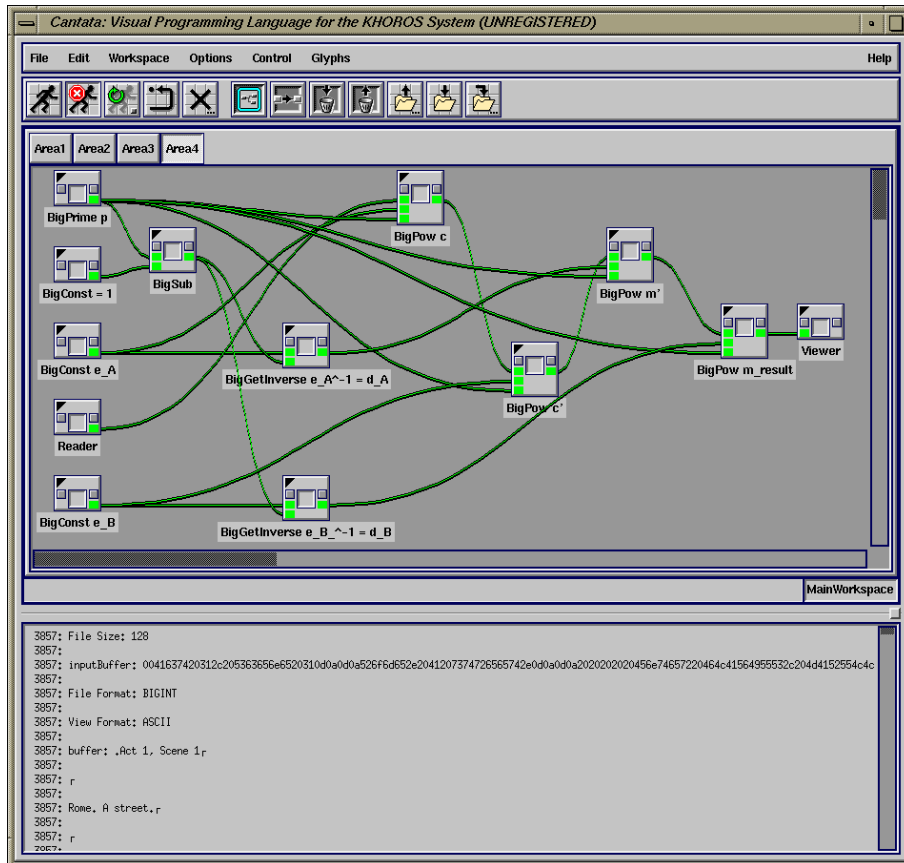


Figure 7: *Shamir's No-Key Protocol in Cantata*

7 Message Authentication

Figure 8 shows the Cantata model of a message authentication based on the RSA algorithm. A MD5 check sum is calculated over a message with the Hash-glyph. This check sum is encrypted and sent to the receiver of the message. The receiver can decrypt the check sum with the public key of the sender. If the decrypted check sum is equal to the check sum calculated over the message everything is OK.

This method could also be used as a digital signature since a change to a contract would be easy to detect with the check sum. A protocol for digital signatures for contracts could work like this:

1. Bob sends a contract to Alice. The contract is secured by a check sum which is encrypted with Bob's secret key.
2. Alice receives the contract and is able to check via the encrypted check sum that ...
 - (a) the message is from Bob
 - (b) the contract was not altered during transmission

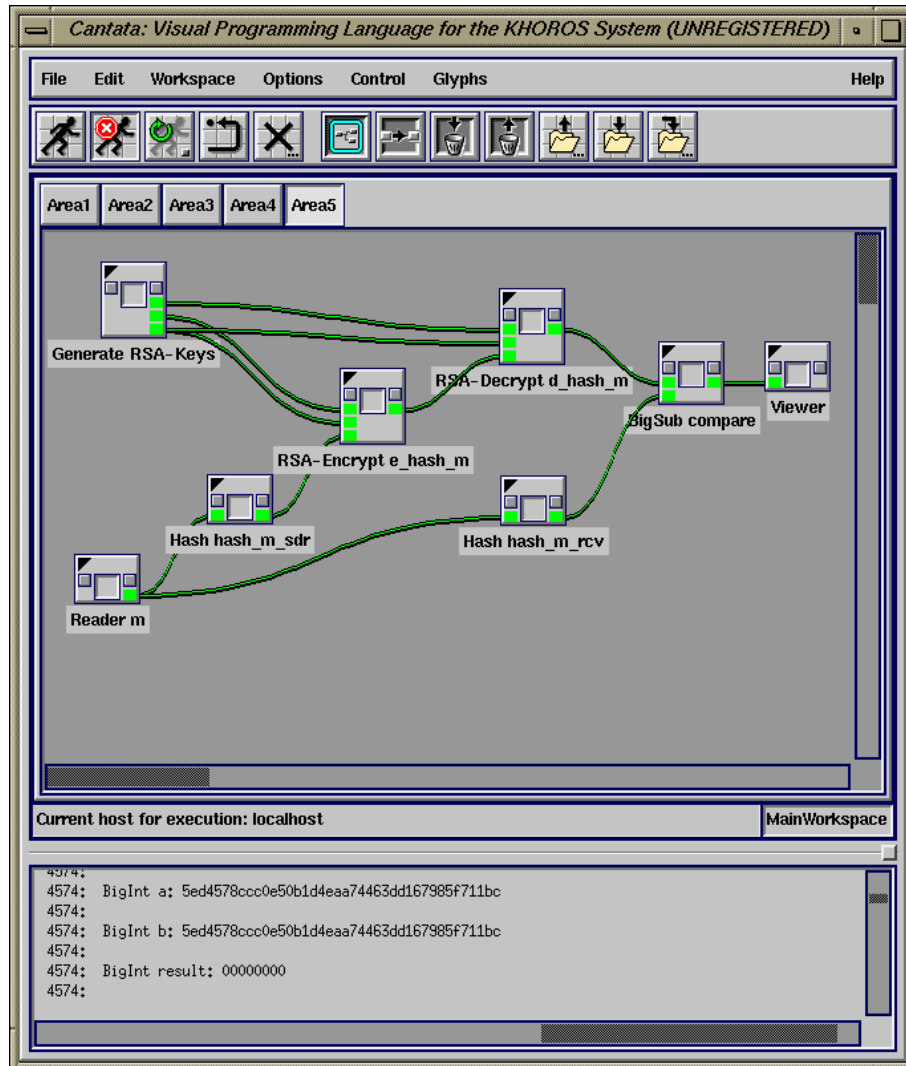
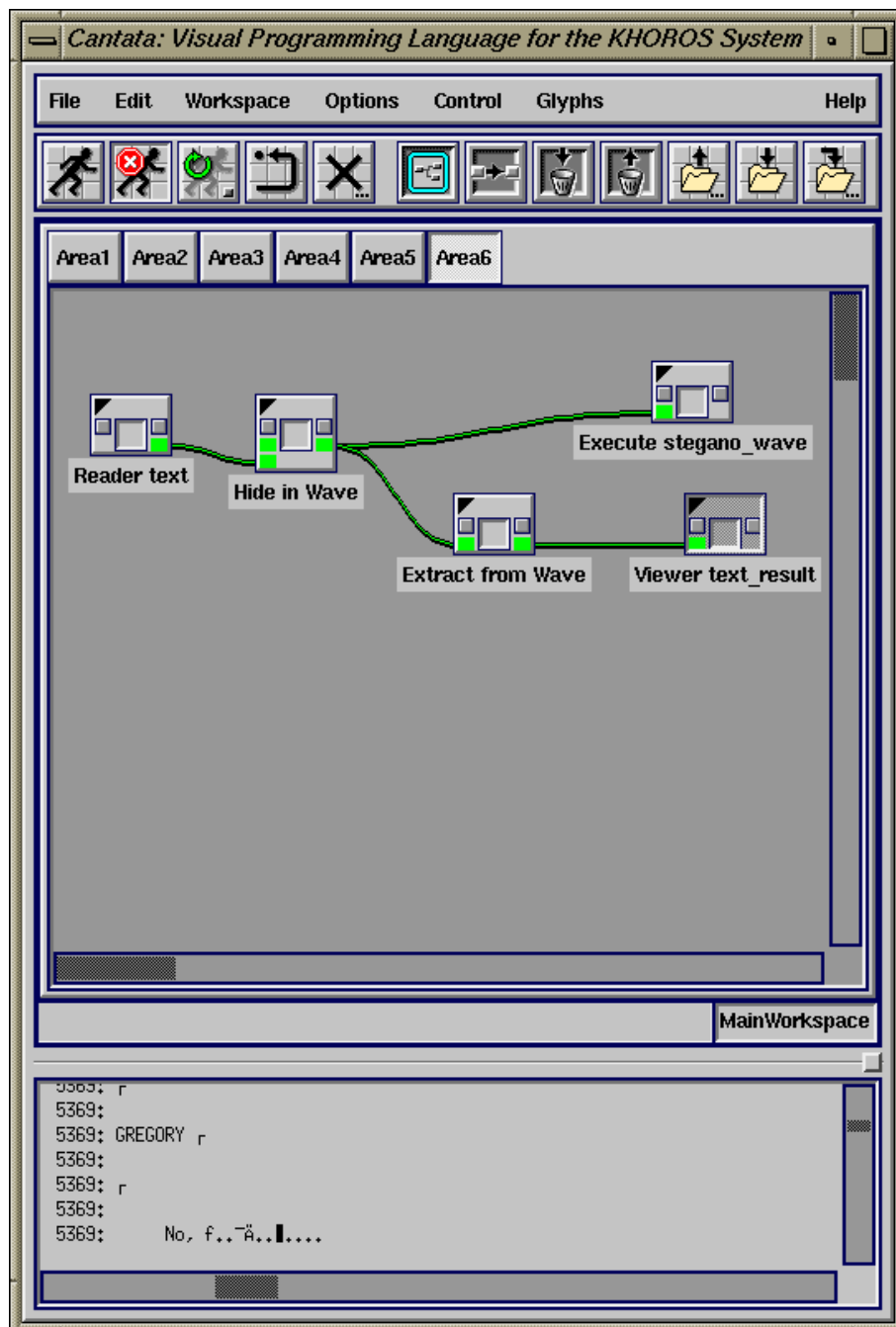


Figure 8: *Message Authentication Cantata*

3. If Alice agrees to the contract she calculates a new check sum over the contract **AND** Bob's check sum. She sends the contract and her own check sum back to Bob.
4. Bob checks if the contract was altered by comparing a check sum of his original to a check sum of the contract. If the contract was not illegally altered he can decrypt Alice's check sum and calculate a new check sum over the contract plus his own check sum and compares it to Alice's check sum. If they are equal the contract is OK.

8 Steganography

In figure 9 you can see a Cantata-model which reads a message and hides it in a wave-file.

Figure 9: *Steganography in Cantata*

With Steganography a message could be hidden in a picture, or a video-stream. It does not matter in what kind of data you hide your message but this data is preferable multimedia-data where the corruption of some bits does not have noticeable effects on the original picture, video or sound data.

This approach could be made even more secure by hiding encrypted messages and not plain text.

9 Conclusions

Different encryption algorithms are presented in this assignment. The hardest thing was the implementation of the RSA-cracking program. This was the most time consuming exercise.

I think the exercise to copy the formulas to proof that RSA works from the script of Mr Schmidt was complete nonsense. In the master course there are a lot of very bright people but it guess none of them is able to do this proof. So what do they do ? They go to a book or a script from a Prof and copy the one and a half pages of equations. I don't think that such an exercise makes much sense.

A The program get_key

get_key.h

```
#ifndef _GET_KEY_H_
#define _GET_KEY_H_

#ifdef __cplusplus
extern "C" {
#include <stdlib.h>
#include <math.h>
}
#endif

#include <map>
#include <vector>
//#define N 56480441318443LL
//#define ROOT 7515348LL
//#define N 299LL
//#define ROOT 17LL
//#define N 58578673415881LL
//#define ROOT 7653670LL
//#define N 53894924919989LL
//#define ROOT 7341346LL
//#define N 56913547008931LL
//#define ROOT 7544106LL
#define N 64411051 /*8101 * 7951*/
#define ROOT 8026 /* --> s_key = 49043383 (checked in \
calculator)*/
//#define N 143 /*13*11*/
//#define ROOT 10
//#define N 19601063 /*2393*8191*/
```

A The program get_key

```
//#define ROOT 4428
//#define N 253LL
//#define ROOT 14LL

#define PUBLIC_KEY 2047
//#define PUBLIC_KEY 17

typedef long long int ullong;

void get_key(ullong p_key, ullong phi_n);
#endif
```

get_key.cc

```
#include "get_key.h"

void get_key(ullong p_key, ullong phi_n){
    map<ullong, ullong> results;

    ullong rest, s_key, guess;
    ullong ar [3], ar_tmp [3], mult;
    results [p_key]=1;

    guess = (ullong)phi_n/p_key;
    if ( guess % 2 == 0) guess++;

    rest = (guess*p_key)%phi_n;
    results [rest] = guess;
    ar[2]= rest;

    rest = (1 * p_key) % phi_n;
    results [rest] = 1;

    cout << "└guess└*└p_key└mod└phi(n)└=└" << ar[2] << endl\
        << endl;

    cout << rest << "└-->└" << 1 << endl;
    cout << ar [2] << "└-->└" << guess << endl;

    if ( rest < ar [2]) {
        ar[0]=ar [2];
        ar[1]=rest;
        ar[2]=rest;
    }
    else {
        ar[0]=rest;
    }
}
```

A The program get_key

```
    ar[1]=ar [2];
    ar[2]=ar [2];
}
while (rest>1) {
    ullong small, big, mid;

    small = ar [0];
    big = ar [0];
    mid = ar[0];
    /*
     * sort array ar
     * ar[0] - second smallest element
     * ar[1] - smallest element
     * ar[2] - result of ar[2] - ar [1]
     */
    for (int i=1; i<3; i++){
        if (small > ar[i] ) {
            small=ar[i];
        }
        if (big < ar[i] ) {
            big=ar[i];
        }
    }

    for (int i=0; i<3; i++){
        if ( ( ar[i] < big)&&(ar[i] > small) ) mid=ar[i];
    }
    ar [1] = small;
    ar [0] = mid;

    mult = (ullong) mid / small;

    if (mult<1) mult = 1 ;

    ar [2] = ar [0] - (mult * ar [1]) ;
    rest = ar [2];
    small = results [ar [0]];
    mid  = results [ar [1]];
    s_key = (results [ar [0]] - (mult * results [ar [1]]) ) % phi_n;
    if (s_key<0) s_key = phi_n + s_key;

    results [ar [2]] = s_key;
}
if (s_key==0) {
    cout << "phi(n) had common factor with p" << endl;
} else {
```

B The program crypt

```
        cout << "rest=" << ar[2] << " --> secret_key=" << \
            s_key << endl;
    }
}
```

get_key_main.cc

```
#include "get_key.h"

int main (int argc, char* argv[]){
    ullong p;
    ullong phi_n;

    if (argc<4) {
        cout << "Usage:" << endl;
        cout << argv[0] << " public_key(phi(n)|q|r)" \
            << endl;
        exit(-1);
    }
    /*for p = 11 q=13 r=7 --> s=59*/
    if (argc==4) {
        p = atoi(argv[1]);
        phi_n = (atoi(argv[2])-1)*(atoi(argv[3])-1);
        cout << "public_key=" << p << endl;
        cout << "phi_n=" << phi_n << endl;
        get_key( p, phi_n );
    }
    if (argc==3) {
        p = atoi(argv[1]);
        phi_n = atoi(argv[2]);
        cout << "public_key=" << p << endl;
        cout << "phi_n=" << phi_n << endl;
        get_key( p, phi_n );
    }

    return(1);
}
```

B The program crypt

This little application can be used to crypt and encrypt little messages.

crypt.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
typedef long long unsigned int ullong;

typedef struct ullong_struct{
    ullong val;
    struct ullong_struct *next;
} ullong_list_t ;

ullong crypt(ullong msg, ullong key, ullong modul) {
    ullong_list_t *rest_ptr ;
    ullong_list_t *tmp_rest_ptr;
    ullong_list_t *pot_ptr;
    ullong_list_t *tmp_pot_ptr;

    ullong pot;
    ullong rest ;

    ullong mask, size_of_vector , result ;
    int i , end;

    pot_ptr = malloc (sizeof( ullong_list_t ));
    rest_ptr = malloc (sizeof( ullong_list_t ));
    pot_ptr -> val = 0;
    pot_ptr -> next = NULL;
    rest_ptr -> val = 0;
    rest_ptr -> next = NULL;

    tmp_pot_ptr = pot_ptr;
    tmp_rest_ptr = rest_ptr;

    mask = 1;
    size_of_vector = sizeof(mask)*8;

    /* do binary decomposition of key */
    for (i=0; i<size_of_vector ; i++){
        /* if bit set ...*/
        if ( (mask << i) == (key & (mask << i) ) ){
            tmp_pot_ptr->val = mask << i;
            tmp_pot_ptr->next = malloc (sizeof(ullong_list_t));
            if (tmp_pot_ptr->next==NULL) {
                perror("Error allocating list element");
                exit (-1);
            }
            tmp_pot_ptr = tmp_pot_ptr->next;
            tmp_pot_ptr -> next = NULL;
            tmp_pot_ptr -> val = 0;
        }
    }
}
```

```
    }
}

printf("%11d_=", key);
for (tmp_pot_ptr=pot_ptr; tmp_pot_ptr->next!=NULL; \
     tmp_pot_ptr=tmp_pot_ptr->next){
    printf("%11d", tmp_pot_ptr->val);
    if (tmp_pot_ptr->next->next != NULL) printf("_+");
}
puts("");

pot      = 1;
rest     = msg;
tmp_pot_ptr = pot_ptr;
end      = 0;

while ( (tmp_pot_ptr -> next != NULL) && (!end) ){
    while ( (tmp_pot_ptr -> val < pot) && (!end) ){
        tmp_pot_ptr = tmp_pot_ptr -> next;
        if (tmp_pot_ptr -> next == NULL) end=1;
    }

    if (!end) {
        printf("%11d_mod_%11d_=", rest, modul);
        rest = rest % modul;
        printf("%11d\n", rest);

        /*if the actual pot equals a pot contained in the key ...*/
        if (pot == tmp_pot_ptr->val) {
            tmp_rest_ptr->val = rest;
            tmp_rest_ptr->next = malloc(sizeof(ulong_list_t));
            tmp_rest_ptr      = tmp_rest_ptr->next;
            tmp_rest_ptr->next = NULL;
            tmp_rest_ptr->val = 0;
        }
        //if (pot==1) printf ("%11d_mod_%11d_=%11d\n", rest, \
            modul, rest);
        //else printf ("%11d_mod_%11d_=%11d\n", rest*rest, modul\
            , rest);

        rest *= rest;
        pot*=2;
    }
}
```

```
for (tmp_rest_ptr=rest_ptr; tmp_rest_ptr->next!=NULL; \
     tmp_rest_ptr=tmp_rest_ptr->next){
    printf ("%11d", tmp_rest_ptr->val);
    if (tmp_rest_ptr->next->next != NULL) printf("_");
}
puts("");

end      = 0;
tmp_rest_ptr = rest_ptr;
printf ("%11d_x%11d=_", tmp_rest_ptr -> val, tmp_rest_ptr -> \
        next -> val);
rest = tmp_rest_ptr -> val * tmp_rest_ptr -> next -> val;
printf ("%11d;\t%11d_mod%11d=_", rest, rest, modul);
rest = rest % modul;
printf ("%11d\n",rest);

tmp_rest_ptr = tmp_rest_ptr -> next -> next;
while ( tmp_rest_ptr -> next != NULL ){
    printf ("%11d_x%11d=_",rest, tmp_rest_ptr -> val);
    rest *= tmp_rest_ptr -> val;
    printf ("%11d;\t%11d_mod%11d=_", rest, rest, modul);
    rest = rest % modul;
    printf ("%11d\n",rest);
    tmp_rest_ptr = tmp_rest_ptr -> next;
}
printf ("RESULT:%11d\n", rest);
result=rest;

/* Do memory cleanup */
for (; pot_ptr->next!=NULL;) {
    tmp_pot_ptr=pot_ptr;
    pot_ptr=tmp_pot_ptr->next;
    free (tmp_pot_ptr);
}
free (pot_ptr);

for (; rest_ptr->next!=NULL;) {
    tmp_rest_ptr=rest_ptr;
    rest_ptr=tmp_rest_ptr->next;
    free (tmp_rest_ptr);
}
free (rest_ptr);

return result;
```

C The program prim

```
}

int main (int argc, char* argv[]){
    ullong message, power, modulo, result;

    if (argc<4) {
        puts("Usage:");
        printf ("%s_message_power_modulo\n\n", argv[0]);
        puts("performs_message^power_mod_modulo");
        exit(-1);
    }
    if (argc==4) {
        message = strtol(argv [1], NULL, 10);
        power   = strtol(argv [2], NULL, 10);
        modulo  = strtol(argv [3], NULL, 10);

        result = crypt (message, power, modulo);
        printf ("The_result_is:_c_=%lld\n", result);
    }
    return(1);
}
```

C The program prim

main.cc

```
#include "get_key.h"

int main (void){

    unsigned long long int upper_limit_r=ROOT+3;
    unsigned long long q=ROOT-3;
    unsigned long long r;
    unsigned int found=0;
    unsigned long long q_times_r=(N+2);
    unsigned long long phi_n;
    unsigned long long secret_key;
    unsigned long long public_key;
    unsigned long long u, rest, x, m, n;
    u=0;

    r = upper_limit_r;
    q_times_r = q*r;
    for (; q<=N;) {
```

C The program prim

```
q=q+2;
q_times_r = q_times_r + r + r;
if (q_times_r==N) {
    found=1;
    cout << "q=□" << q << "\tr=□" << r << endl;
    break;
}
//printf("q=%lld\tr=%lld\tq*r=%lld\n",q,r,q_times_r);
for (; q_times_r>=N;){
    r=r-2;
    q_times_r = q_times_r - q - q;
    if (q_times_r==N) {
        found=1;
        cout << "q=□" << q << "\tr=□" << r << endl;
        break;
    }
}
}
}

phi_n=(q-1)*(r-1);
cout << "phi(n)=□" << phi_n << endl;

// get the secret key
get_key(PUBLIC_KEY, phi_n);
}
```

References

- [1] Thomas Owens (Script) “Coding for Compression and Data Security”