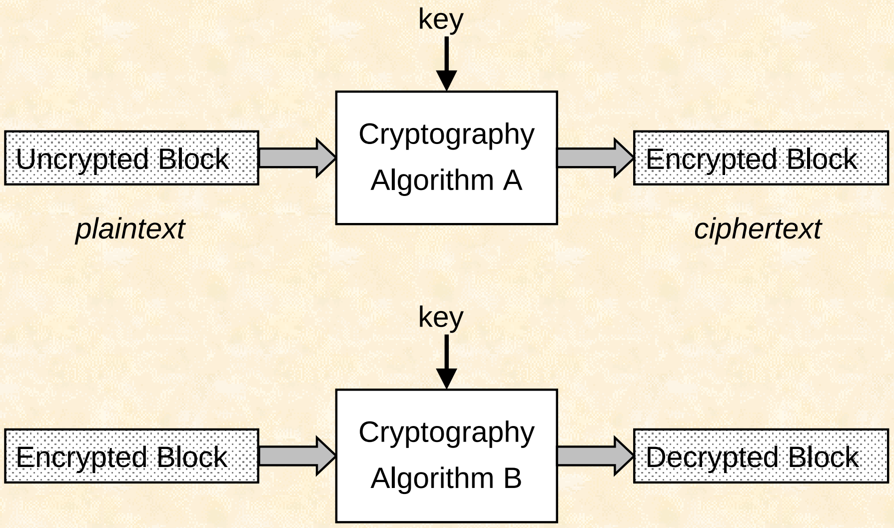
An Algorithm for Symmetric Cryptography with a wide range of scalability

By Klaus Felten

Contents

- Basic Concept for Symmetric Cryptography
- Block diagram: Encrypting a message
- Block diagram: Decrypting a message
- Algorithms for pseudo random-number generators
- Byte Exchange
- Scalability of the algorithm
- Goal of the warm-up-cycles
- Expense of cracking the key
- Features of the presented algorithm
- Possible weak-points
- Future activities
- References

Basic Concept for Symmetric Cryptography



Characteristic:

The same key is used for encryption and decryption

Basic Rules for efficient Cryptography are defined by Shannon:

Confusion:

Goal: Hiding the relationship between message and encrypted message

Good: Any character in the plaintext-block is replaced by another character, but not always the same character

Bad: Any character in the plaintext-block is replaced by one corresponding character

Diffusion:

Goal: Distribution of changes over the complete encrypted message

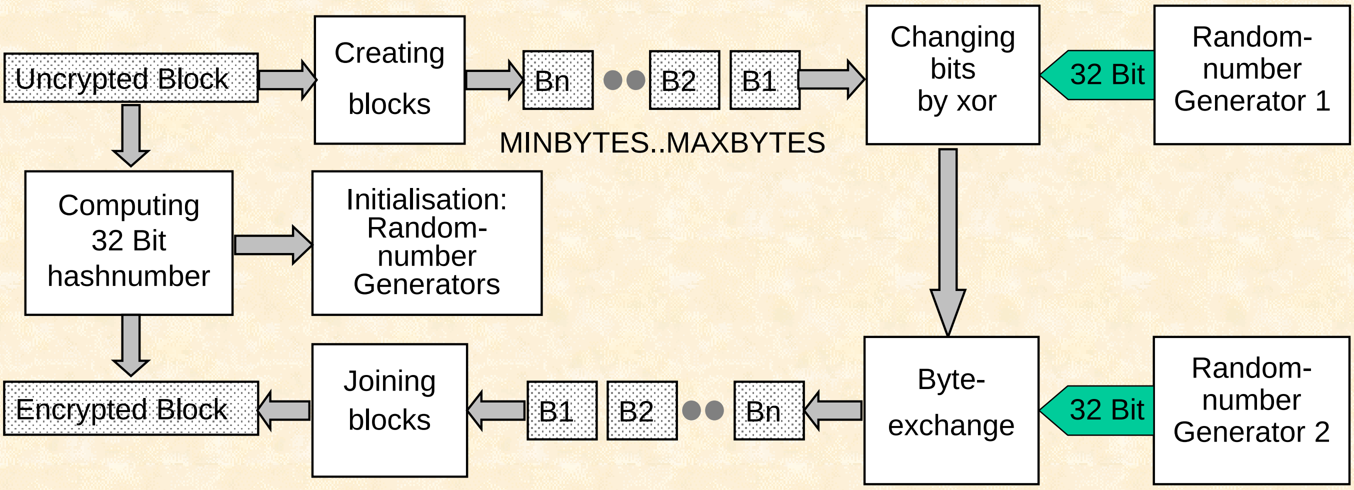
Good: If we change one bit of the *message*, all bits in the *encrypted message* may change

Bad: If we change one bit of the *message*, only one bit in the *encrypted message* will change

Good: If we change one bit of the *key*, all bits in the *encrypted message* may change

Bad: If we change one bit of the *key*, only one bit in the *encrypted message* will change

Block diagram: Encrypting a message



Uncrypted Block

Creating blocks

Bn •• B2 B1

MINBYTES..MAXBYTES

Changing bits by xor

32 Bit

Random-number Generator 1

Computing 32 Bit hashnumber

Initialisation: Random-number Generators

Encrypted Block

Joining blocks

B1 B2 •• Bn

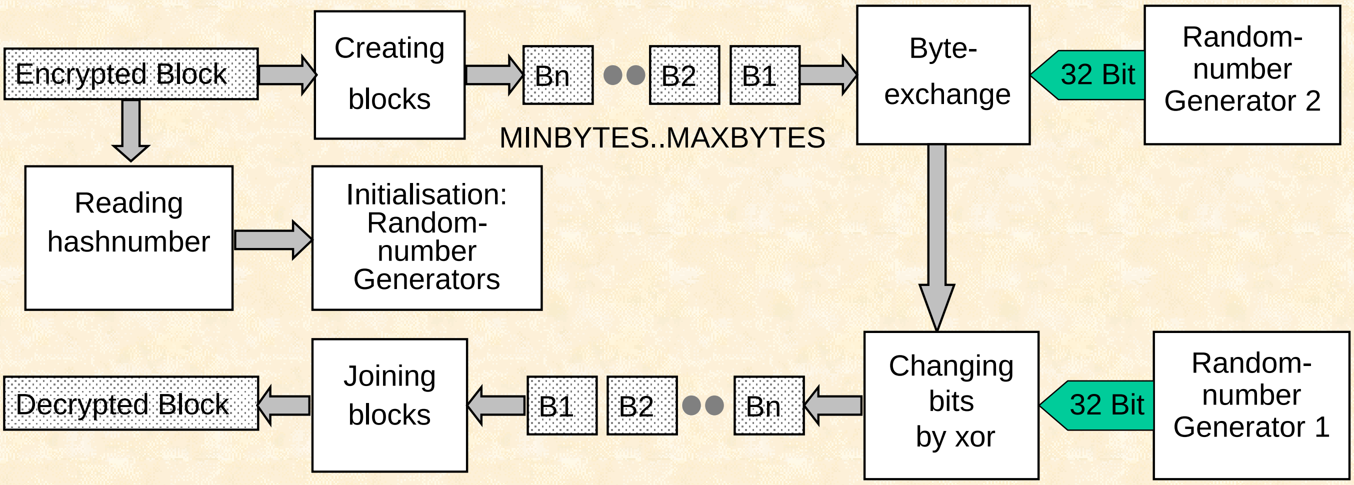
Byte-exchange

32 Bit

Random-number Generator 2

Actions:
1. Computing a 32 bit hashnumber, writing it to the head of the encrypted block and initialising the random-number generators 1 and 2 with this hashnumber
2. Creating blocks of length MINBYTES to MAXBYTES
3. Changing blocks by xor-operation with 32 bit pseudo-random-numbers from generator 1
4. Byte exchange within blocks, dependent on pseudo-random-numbers from generator 2
5. Joining all blocks to the encrypted block with length n+4

Block diagram: Decrypting a message



Encrypted Block

Creating blocks

Bn •• B2 B1

MINBYTES..MAXBYTES

Byte-exchange

32 Bit

Random-number Generator 2

Reading hashnumber

Initialisation: Random-number Generators

Decrypted Block

Joining blocks

B1 B2 •• Bn

Changing bits by xor

32 Bit

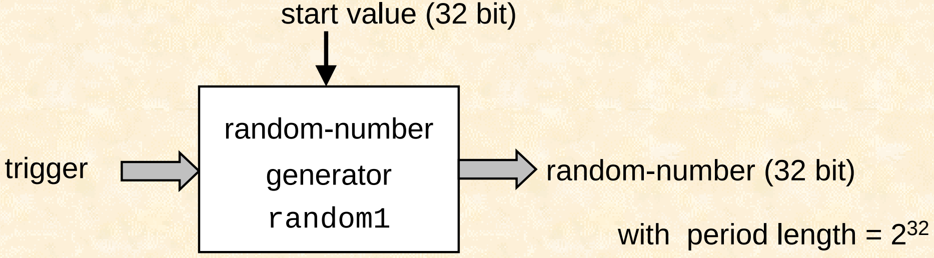
Random-number Generator 1

Actions:
1. Reading 32 Bit hashnumber and initialising random-number generators 1 and 2
2. Creating blocks of length MINBYTES to MAXBYTES
3. Byte exchange within blocks, dependent on pseudo-random-numbers from generator 2
4. Changing blocks by xor-operation with 32 bit pseudo-random-numbers from generator 1
5. Joining all blocks to the decrypted block

This algorithm needs 2 random-number generators with a large period length

This has been realised by two generators-types:

generator-type `random1`



start value (32 bit)

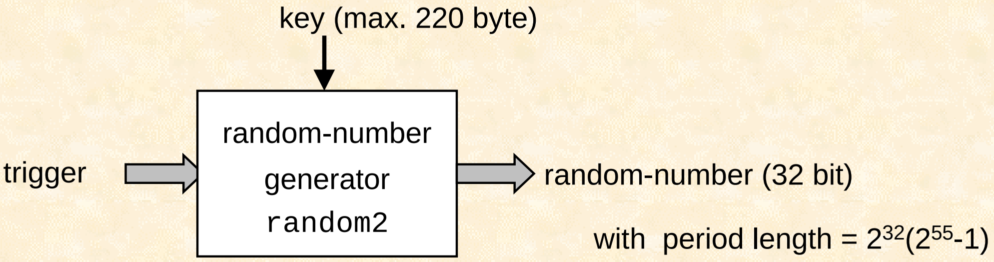
trigger

random-number generator `random1`

random-number (32 bit)

with period length = 2^{32}

generator-type `random2`



key (max. 220 byte)

trigger

random-number generator `random2`

random-number (32 bit)

with period length = $2^{32}(2^{55}-1)$

Algorithm of the pseudo random-number generator-type `random1`

Published by Donald E. Knuth: The Art of Computer Programming, Volume 2

$$X(n+1) = (aX(n) + c) \bmod m$$

where

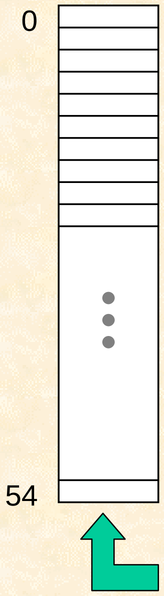
a, c are large prime numbers

$a \neq c, a < m$

Suitable constants for a 32 bit random-number generator are:

```
a = 4294967279  = 2^32 -17
c =  715827883
m = 4294967296  = 2^32
```

Algorithm of the pseudo random-number generator-type `random2` called ***additive number generator***



Published by Donald E. Knuth: The Art of Computer Programming, Volume 2

Originally developed by G.J. Mitchell and D.P. Moore

$$X(n) = (X(n-24) + X(n-55)) \bmod m$$

where

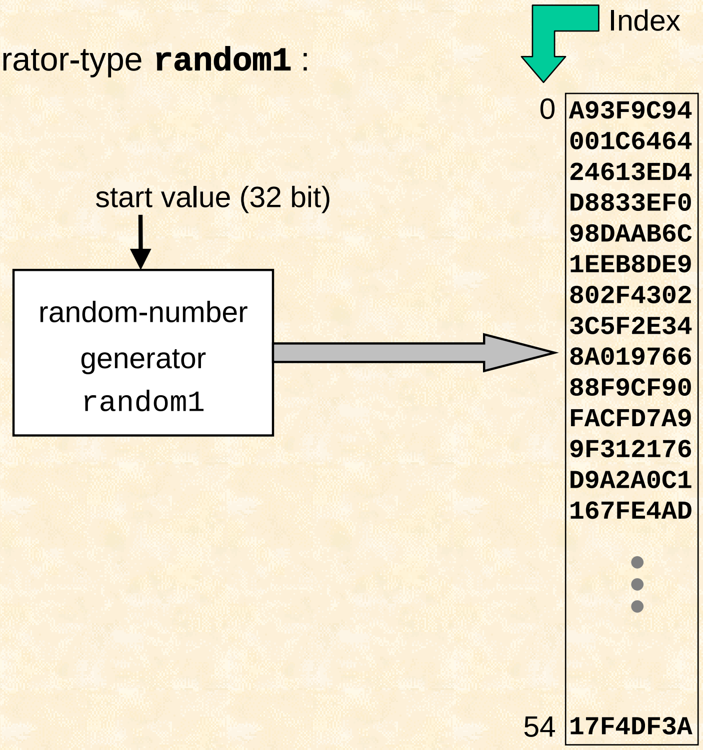
$X(0),...X(54)$ are arbitrary integers not all even

$n >= 55$

Array with integer numbers, values from 0 to m-1

Initialization of the array with 32 bit integer numbers

This is realised by generator-type **`random1`** :



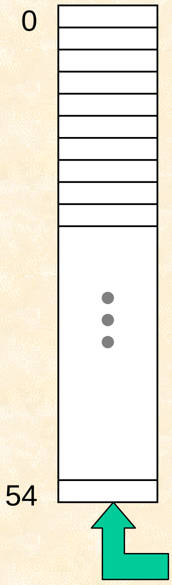
A93F9C94
001C6464
24613ED4
D8833EF0
98DAAB6C
1EEB8DE9
802F4302
3C5F2E34
8A019766
88F9CF90
FACFD7A9
9F312176
D9A2A0C1
167FE4AD
17F4DF3A

Implementation of the pseudo random-number generator `random2` (additive number generator) as a C++ program

```
// Constants
#define KK 55
```

```
class randomnumber{
   // private members
   int j, k;
   u_long y[KK];// Content of array determines the
                // sequence of 32 Bit random-numbers
```

```
   u_long randomnumber::random2( void )
   {
   u_long i1;
      i1 = y[k] + y[j]; // Modulo 2^32 by overflow
      y[k] = i1;
      j--; k--;
      if ( k < 0 ) k = KK-1;
      if ( j < 0 ) j = KK-1;
      return i1;
   }
```



32 bit numbers (u_short), values from 0 to 2^{32} -1

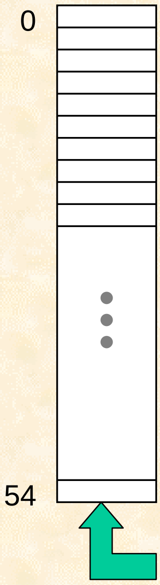
Initialisation of the ***additive number generator***

1. Filling the array "y" with pseudo random-numbers

```
void init_random1( u_long ix )
{
   int i;
      j=KJ-1; k=KK-1;
      //Initializiation of the array with random-numbers
      for ( i=0; i < KK; i++){
         ix = random1( ix );
         y[i] = ix;
      }
}
```

Note: The hashnumber determined by the input-block is used as the start-value for the initialisation procedure.

Result: If we change one bit of the *message*, all bits in the *encrypted message* may change.



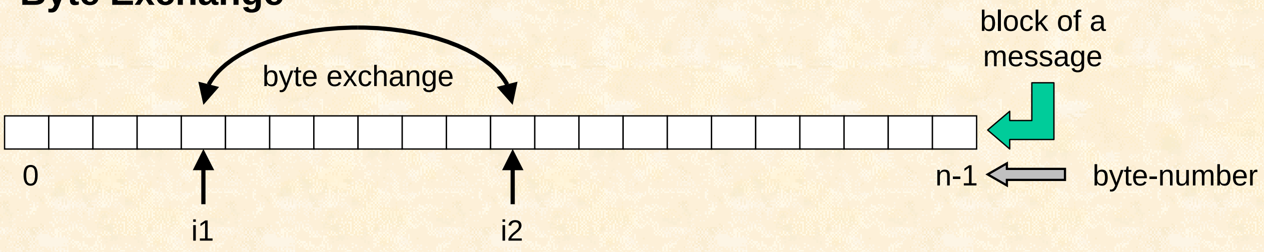
array "y"

Initialisation of the ***additive number generator*** (2)

2. Adding the `key`, byte by byte, to the array elements

```
void init_random2( char sx[] )
{
int i, j;
u_char *py;
   py = (u_char*)y;
   for ( i=0, j=0; i < KK; py++, i++, j++ ){
      if ( sx[j] == '\0' ){
         j = -1;
         continue;
      }
      // Add key[j] to y-Buffer Modulo 2^8
      *py = *py + sx[j];
   }
}
```

Byte Exchange



n = current blocklength = MINBYTES..MAXBYTES

Rules

- ❖ The two bytes for exchange are determined by the random-number generator random2.
- ❖ n/2 bytes are exchanged; n/2 bytes stays unchanged; therefore n/4 exchange-operations are performed.
- ❖ Any byte will be exchanged only one times.

Program code (principle)

```
while (current_number_of_exchanges < n/4){
   i1 = random_number();// next random-number
   i2 = random_number();// next random-number
   byte_exchange(i1, i2);
}
```

Scalability of the algorithm

- ❖ key-length from 4 to 220 bytes
- ❖ security levels from 1 to 7

The security level determines the value of `startloops`, used for initialization of the random-number generators 1 and 2:

security level	startloops	duration
1	200	10µs
2	2.000	100µs
3	20.000	1ms
4	200.000	10ms
5	2.000.000	100ms
6	20.000.000	1s
7	200.000.000	10s

The random-numbers generated during the warm-up-cycles are ignored.

Goal of the warm-up-cycles (startloops)

Every warm-up-cycle needs at least:

- ❖ 2 integer additions of array elements
- ❖ 2 integer decrement operations
- ❖ 2 integer comparisons

The warm-up-cycles must be finished before encryption or decryption may start.

Only in this case the right pseudo random-number sequences from random-number generator 1 and 2 are available.

Result: This forced time-consumption makes it more difficult to break the key.

There are some requirements for security:

- ❖ There is no way to get the right pseudo random-number sequence without performing `startloops` warm-up-cycles
- ❖ Special hardware does not have a significantly higher speed than a CPU from Intel or AMD.

Expense of cracking the key (1)

We assume that the attacker

- knows the exact algorithm for encryption/decryption
- is able to cause a known message to be encrypted with the searched key (called: chosen-plaintext-attack)
- tries to get the right key by attempting any possible key (called: brute-force-attack)

Further we assume that

- decryption of a (short) chosen message needs `dt` seconds
- warm-up-run needs `wt` seconds

Therefore any trial needs at least `tt` seconds.

```
with tt = dt + wt
```

Expense of cracking the key (2)

Let us consider the worst case:

- key-length = 4 Byte, using only printable characters (95 characters)
- $dt = 10^{-5}$ s; $wt = 10^{-5}$ s; $tt = 2*10^{-5}$ s

On the average $95^4/2$ trials are needed to determine the right key.

The required time is:

$t = 95^4/2 * 2*10^{-5}$ s

$t = 815$ s

Result: Under these conditions it is easy to crack the key.

Expense of cracking the key (3)

Let us consider a further case:

❖ key-length = 4 Byte, using only printable characters (95 characters)

❖ $dt = 10^{-5}$ s; wt = 1s; tt ≈ 1s

On the average $95^4/2$ trials are needed to determine the right key.

The required time is:

$t = 95^4/2 * 1s$

t = 40725313s

t = 11313h (1000 PCs woud need 11.3h)

Result:
Under these conditions it is very difficult to crack even such a short key.

Expense of cracking the key under normal conditions

❖ key-length = 6 Byte, using only printable characters (95 characters)

❖ security level from 1 to 7

❖ brute-force-attack

❖ single PC with CPU AMD Athlon 1700+

security level	startloops	crack time (average)
1	200	85 days
2	2.000	468 days
3	20.000	4.296 days
4	200.000	117 years
5	2.000.000	1.166 years
6	20.000.000	11.654 years
7	200.000.000	116.548 years

Features of the presented algorithm

- key-length from 4 to 220 bytes
- block-length from 16 to FILE_LENGTH
- security levels from 1 to 7 by changing only one parameter
    - 1 = quick: encryption/decryption requires 10µs
    - 7 = slow: encryption/decryption requires 10s
- easy to understand
- no hidden features
- easy to implement on PC or workstation
- Encryption/Decryption speed;
  CPU=AMD Athlon 1700+; input- and output-file on harddisk :
    - Encryption: 2 MB/s
    - Decryption: 3 MB/s

Note: A large key-length makes it possible to choose a key which is easy to remember such as "My neighbors to the left are Frank&Mary"

Possible weak-points of the algorithm

- A chosen plaintext attack may give sufficient information about the internal status of the pseudo random-number generator.

  ⇨ Result:
  The attacker may be able to predict the random-number sequence.

- The hashnumber stored in the encrypted block may give too much information about the plaintext.

Future activities

- Examination of the weak-points.
- Finishing the C++ program
- Publishing the source code

References

❖ Knuth, D. E.: The Art of Computer Programming.
Volume 2. Seminumerical Algorithms. Addison-Wesley 1998

❖ Schneier, Bruce: Applied Cryptography, Addison-Wesley 1996

❖ Schneier, Bruce: Why Cryptography is Harder Than it Looks. 1997;
www.counterplane.com/whycrypto.html

❖ Schneier, Bruce: A Self-Study Course in Block-Cipher Cryptoanalysis. 1997;
www.counterplane.com/self-study.html

❖ Krzyzanowski, Paul: Lectures on distributed systems:
Cryptographic communication and authentification. 1997-2001
www.pk.org/rutgers/notes/pdf/crypto.pdf

WWW Documents

❖ K. Felten: An Algorithm for Symmetric Cryptography with a wide range of scalability. www.e-technik.fh-kiel.de/~felten/iws2003/crypto_01.pdf.

❖ K. Felten: Abstract: An Algorithm for Symmetric Cryptography ...
www.e-technik.fh-kiel.de/~felten/iws2003/abstract.pdf.

❖ Test-program crypt.exe for PC with Intel-CPU:
www.e-technik.fh-kiel.de/~felten/iws2003/crypt.exe

❖ C++ Source code and header-file for random-number generators `random1` and `random2`:
www.e-technik.fh-kiel.de/~felten/iws2003/ranclass3.cpp
www.e-technik.fh-kiel.de/~felten/iws2003/ranclass3.h

WWW Links

❖ Felten's Homepage:
www.e-technik.fh-kiel.de/~felten/

⇨ **current hints** ⇨ **Cryptography Documents**