IF4020 kriptografi

07- Kriptografi Modern dan Stream Cipher

Oleh: Rinaldi M

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2026

Pendahuluan

- Kriptografi modern adalah era kriptografi setelah penemuan komputer digital (awal tahun 1970-an).

- Perkembangan teknologi komputer digital membuat hampir semua bidang ilmu pengetahuan berkebang pesat, termasuk kriptografi.

- Komputer digital merepresentasikan data dan informasi dalam biner.

- Algoritma kriptografi modern beroperasi dalam mode bit (bandingkan dengan algoritma kriptografi klasik beroperasi dalam mode karakter)

  → kunci, plainteks, cipherteks, diproses dalam rangkaian bit

  → operasi **xor** paling banyak digunakan di dalam algoritmanya

- Meskipun disebut kriptografi modern, namun algoritmanya tetap menggunakan dua teknik dasar dasar di dalam kriptografi klasik: **teknik substitusi** dan **teknik transposisi (permutasi)**,

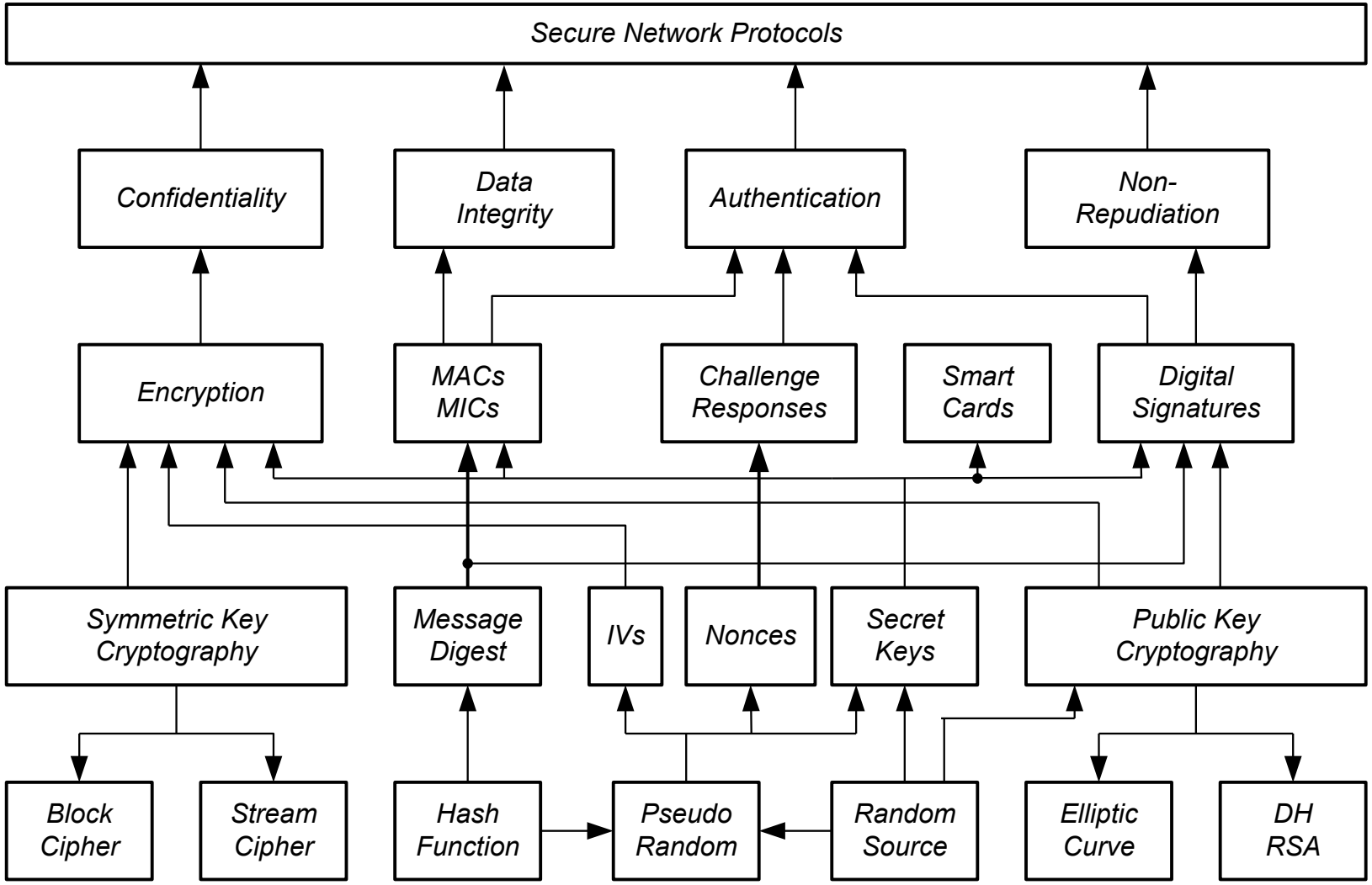
- tetapi operasinya dibuat lebih kompleks, tidak sesederhana *cipher* klasik.

  Tujuannya: agar *cipher* modern lebih sulit dikriptanalisis

- Selain kedua teknik dasar tersebut, juga digunakan teknik lain seperti rotasi, kompresi, ekspansi, penjumlahan modulo, dan lain-lain.

- Kriptografi modern melahirkan konsep-konsep baru seperti algoritma kriptografi kunci-publik, fungsi *hash*, protokol kriptografi, tanda-tangan digital, pembangkit bilangan acak, skema pembagian kunci, dsb.

Diagram Blok Kriptografi Modern



Bit, Byte, dan Kode Heksadesimal

- Pesan di dalam *cipher* modern dienkripsi bit-per-bit atau *byte*-per-*byte*, atau dalam kelompok bit (*byte*).

  1 byte = 8 bit

- Pada beberapa algoritma kriptografi, pesan dikodekan ke dalam kode heksadesimal (**Hex**).

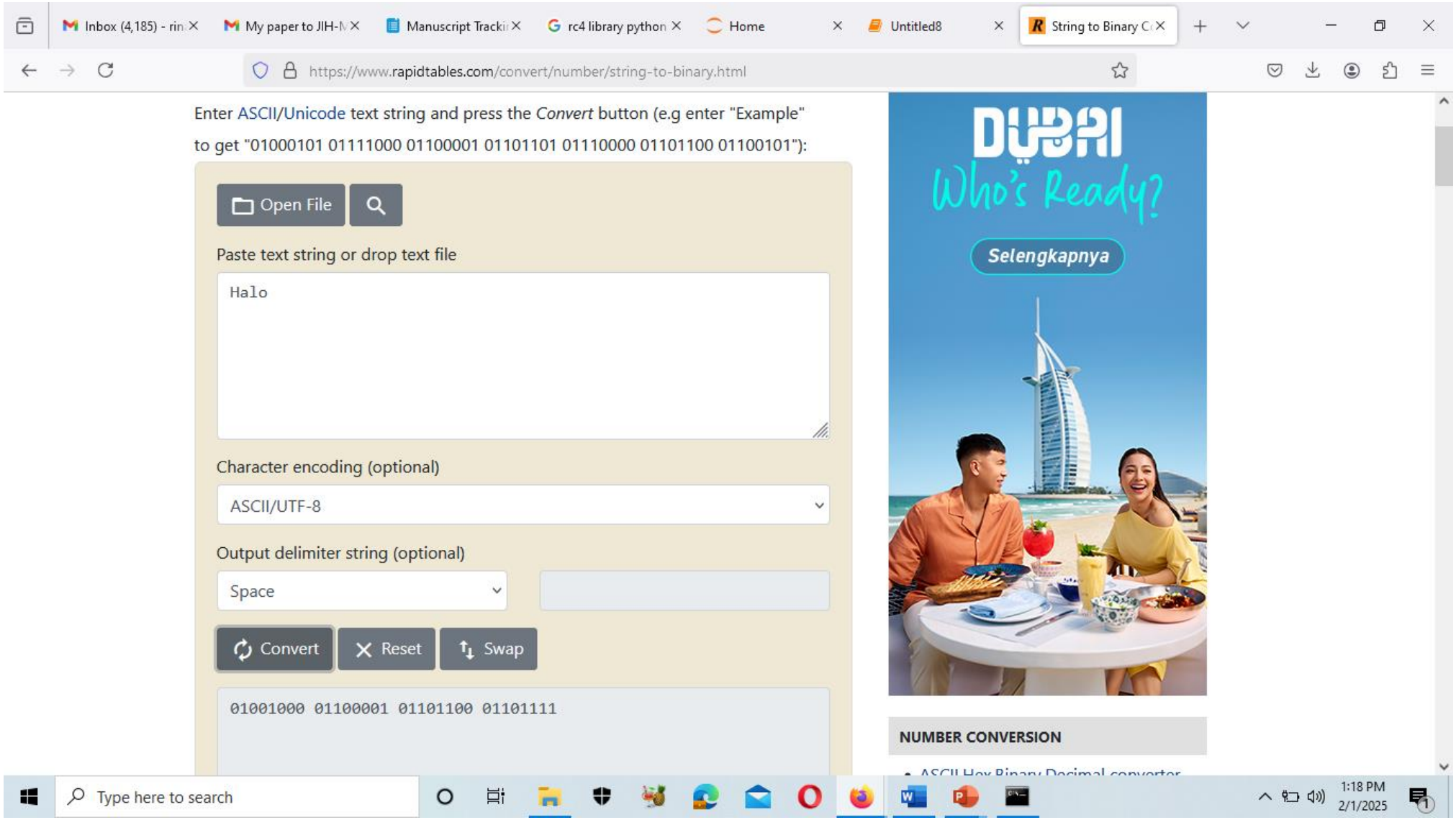
  1 kode hex = 4 bit

0000 = 0	0001 = 1	0010 = 2	0011 = 3
0100 = 4	0101 = 5	0110 = 6	0111 = 7
1000 = 8	1001 = 9	1010 = A	1011 = B
1100 = C	1101 = D	1110 = E	1111 = F

- Contoh: Pesan 1001110101101010 dalam kode Hex dengan cara membagi pesan menjadi blok 4-bit:

  1001 1101 0110 1010 = 9D6A

- Konversi teks ke biner: https://www.rapidtables.com/convert/number/string-to-binary.html

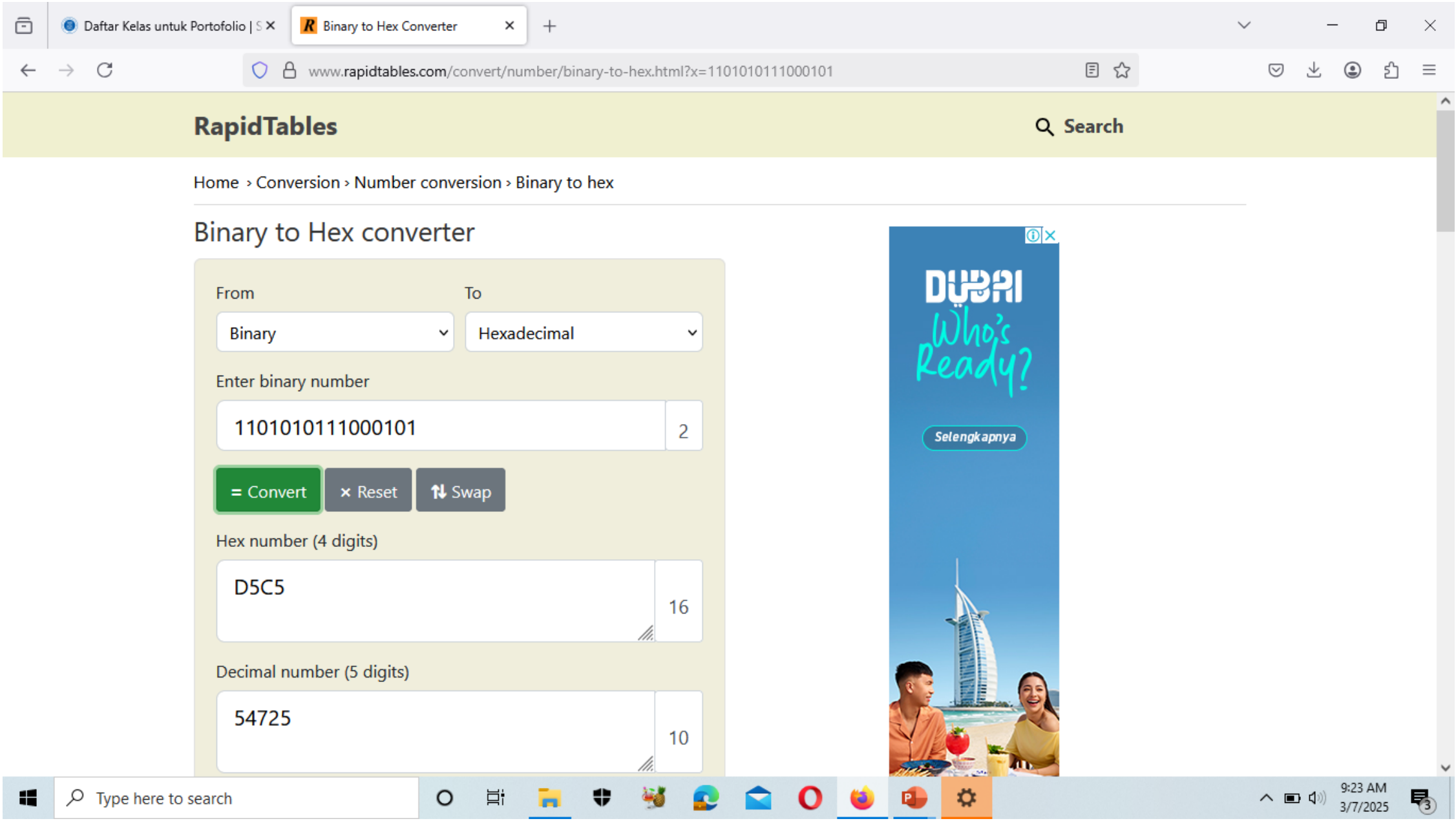


Contoh:

Halo → 01001000 01100001 01101100 01101111

Indonesia emas 2045 →01001001 01101110 01100100 01101111 01101110 01100101 01110011 01101001 01100001 00100000 01100101 01101101 01100001 01110011 00100000 00110010 00110000 00110100 00110101

Konversi biner ke hex:



Operasi *XOR*

- Di dalam *cipher* modern, operasi XOR adalah operasi yang paling sering digunakan
- Notasi: $\oplus$
- Operasi:

$0 \oplus 0 = 0$ $\quad\quad$ $0 \oplus 1 = 1$

$1 \oplus 0 = 1$ $\quad\quad$ $1 \oplus 1 = 0$

a	b	a ⊕ b
0	0	0
0	1	1
1	0	1
1	1	0

- Sifat-sifat operasi XOR:

  (i) $a \oplus a = 0$

  (ii) $a \oplus b = b \oplus a$

  (iii) $a \oplus (b \oplus c) = (a \oplus b) \oplus c$

Contoh:

(i) $1 \oplus 1 = 0$

(ii) $1 \oplus 0 = 0 \oplus 1 = 1$

(iii) $1 \oplus (0 \oplus 1) = (1 \oplus 0) \oplus 1 = 0$

Operasi *Bitwise* XOR

- Jika dua rangkaian dioperasikan dengan *XOR*, maka operasinya dilakukan dengan meng-*XOR*-kan setiap bit yang berkoresponden dari kedua rangkaian bit tersebut.

  Contoh: 10011 ⊕ 11001 = 01010
  yang dalam hal ini, hasilnya diperoleh sebagai berikut:

$$
\begin{array}{ccccccccccc}
1 & & 0 & & 0 & & 1 & & 1 & & \\
1 & & 1 & & 0 & & 0 & & 1 & \oplus & \\
\hline
1 \oplus 1 & & 0 \oplus 1 & & 0 \oplus 0 & & 1 \oplus 0 & & 1 \oplus 1 & & \\
0 & & 1 & & 0 & & 1 & & 0 & &
\end{array}
$$

Cipher Sederhana dengan operasi XOR

- Sama seperti *Vigenere Cipher*, tetapi dalam mode bit
- Setiap bit plainteks di-*XOR*-kan dengan setiap bit kunci.

Enkripsi: $C = P \oplus K$

Dekripsi: $P = C \oplus K$

Contoh:	plainteks	01100101		(karakter ‘e’)
	kunci	00110101	⊕	(karakter ‘5’)
	cipherteks	01010000		(karakter ‘P’)
	kunci	00110101	⊕	(karakter ‘5’)
	plainteks	01100101		(karakter ‘e’)

- Jika panjang bit-bit kunci lebih pendek daripada panjang bit-bit pesan, maka bit-bit kunci diulang penggunaannya secara periodik (seperti halnya pada Vigenere Cipher)

- Contoh:

  Plainteks : 10010010101110101010001110001

  Kunci : 11011011011011011011011011011

  Cipherteks: 01001001110101110001010101010

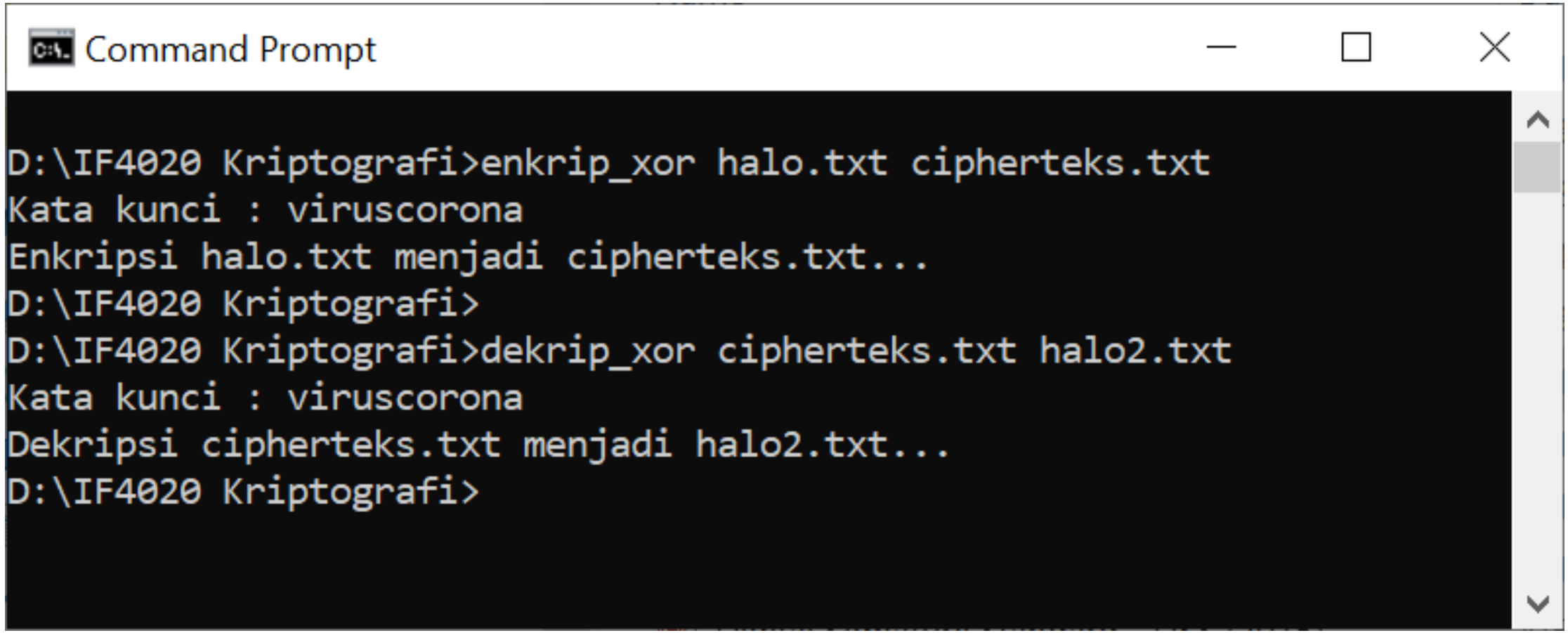
Program C++ untuk enkripsi-dekripsi file dengan cipher XOR sederhana

```
// Enkripsi sembarang berkas dengan
// algoritma XOR sederhana.
#include <iostream>
#include <string.h>
#include <fstream>
#include <stdlib.h>
using namespace std;

main(int argc, char *argv[])
{
 FILE *Fin, *Fout;
 char p, c;
 string K;
 int i;

 Fin = fopen(argv[1], "rb");
 if (Fin == NULL) {
   cout << "Berkas " << argv[1] <<"
tidak ada" << endl;
   exit(0);
 }

 Fout = fopen(argv[2], "wb");

 cout << "Kata kunci : "; cin >> K;
 cout <<"Enkripsi " << argv[1] << "
menjadi " << argv[2] << "...";
 i = 0;
 while (!feof(Fin)) {
   p = getc(Fin);
   c = p ^ K[i];  // operasi XOR
   putc(c, Fout);
   i = (i + 1) % K.length();
 }
 fclose(Fin);
 fclose(Fout);
}
```

(a) enkrip_xor.cpp

```
// Dekripsi sembarang berkas dengan
// algoritma XOR sederhana.
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <fstream>
using namespace std;

main(int argc, char *argv[])
{
 FILE *Fin, *Fout;
 char p, c;
 string K;
 int i;

 Fin = fopen(argv[1], "rb");
 if (Fin == NULL){
   cout << "Berkas " << argv[1] <<"
tidak ada" << endl;
   exit(0);
 }

 Fout = fopen(argv[2], "wb");

 cout << "Kata kunci : "; cin >> K;
 cout <<"Dekripsi " << argv[1] << "
menjadi " << argv[2] << "...";
 i = 0;
 while (!feof(Fin)) {
   c = getc(Fin);
   p = c ^ K[i];  // operasi XOR
   putc(p, Fout);
   i = (i + 1) % K.length();
 }
 fclose(Fin);
 fclose(Fout);
}
```

(b) dekrip_xor.cpp


Command Prompt
D:\IF4020 Kriptografi>enkrip_xor halo.txt cipherteks.txt
Kata kunci : viruscorona
Enkripsi halo.txt menjadi cipherteks.txt...
D:\IF4020 Kriptografi>
D:\IF4020 Kriptografi>dekrip_xor cipherteks.txt halo2.txt
Kata kunci : viruscorona
Dekripsi cipherteks.txt menjadi halo2.txt...
D:\IF4020 Kriptografi>

Program Python untuk enkripsi-dekripsi file dengan cipher XOR sederhana

```python
import sys
```

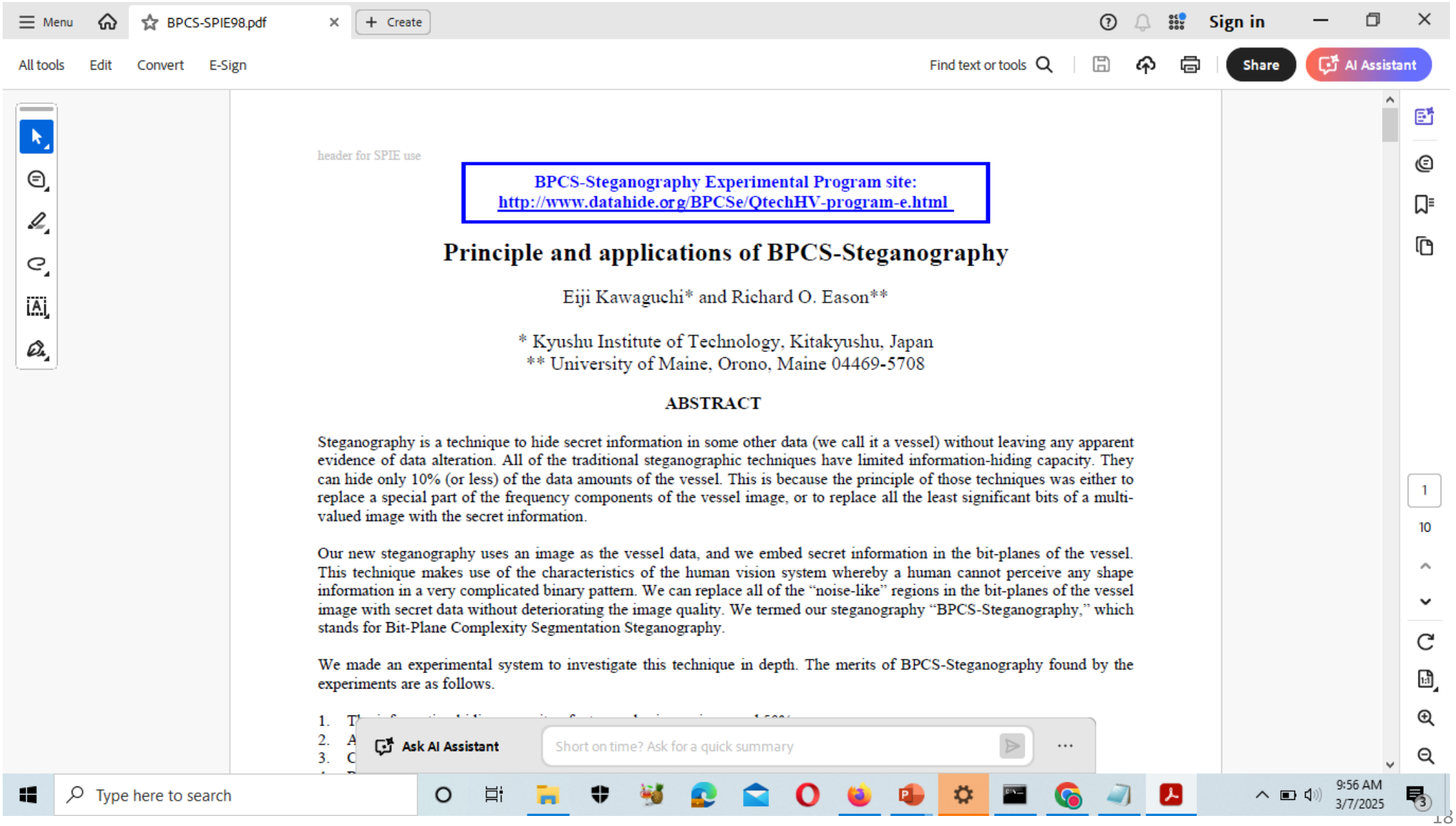
```python
def xor_cipher_encrypt(input_file, output_file, kunci):
    with open(input_file, 'rb') as file:
        plainteks = file.read()

    cipherteks = bytearray()
    byte_kunci = bytearray(kunci, 'utf-8')
    n = len(kunci)
    indeks_kunci = 0
    for byte in plainteks:
        c = byte ^ byte_kunci[indeks_kunci]
        cipherteks.append(c)
        indeks_kunci = (indeks_kunci + 1) % n

    with open(output_file, 'wb') as file:
        file.write(cipherteks)
```

```python
def xor_cipher_decrypt(input_file, output_file, kunci):
    with open(input_file, 'rb') as file:
        cipherteks = file.read()

    plainteks = bytearray()
    byte_kunci = bytearray(kunci, 'utf-8')
    n = len(kunci)
    indeks_kunci = 0
    for byte in cipherteks:
        p = byte ^ byte_kunci[indeks_kunci]
        plainteks.append(p)
        indeks_kunci = (indeks_kunci + 1) % n

    with open(output_file, 'wb') as file:
        file.write(plainteks)
```

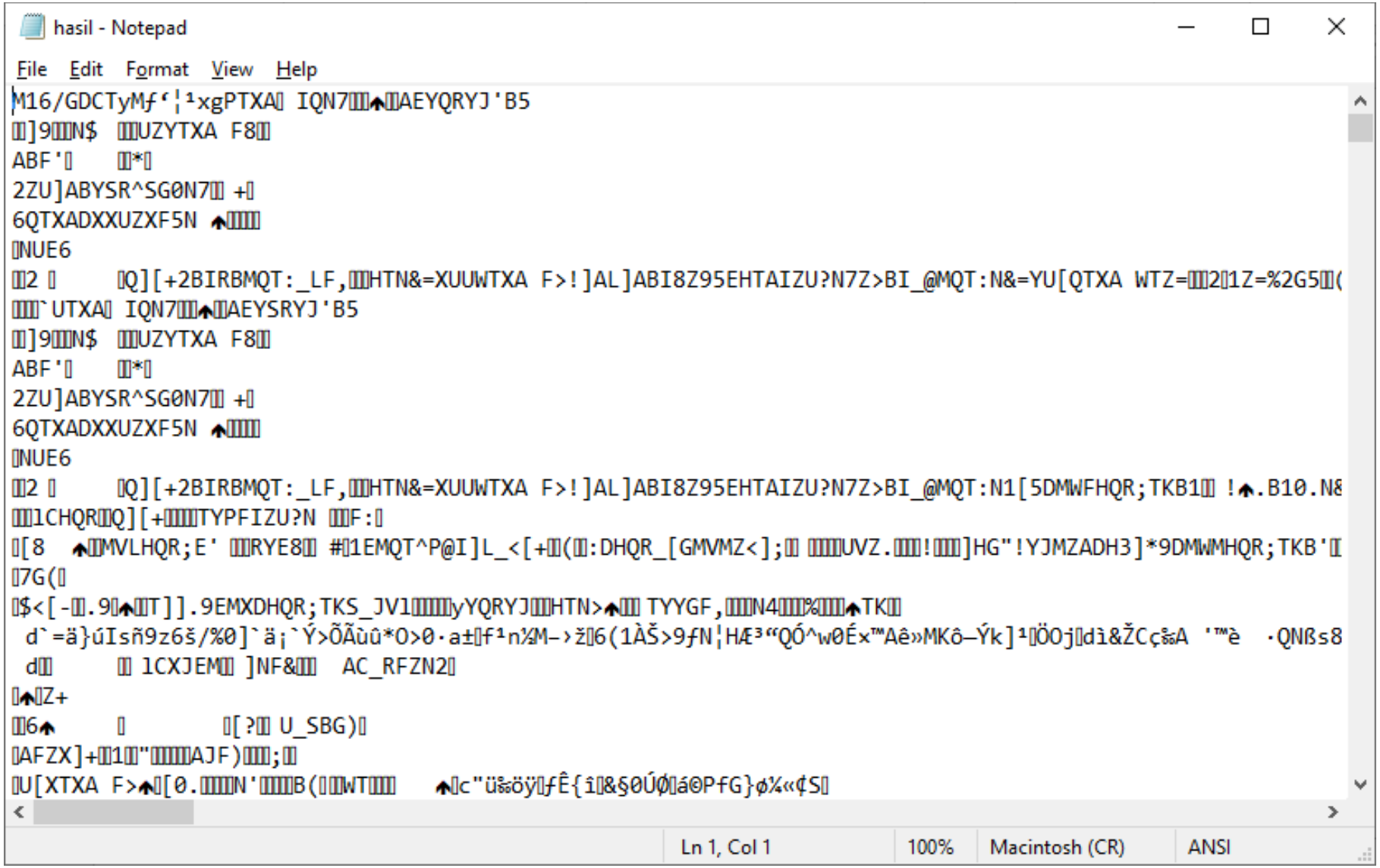
Contoh penggunaan:

```
[4]: input_file = input("Masukkan nama file plainteks:")
     Masukkan nama file plainteks: E:BPCS-SPIE98.pdf

[5]: output_file = input("Masukkan nama file untuk penyimpanan cipherteks: ")
     Masukkan nama file untuk penyimpanan cipherteks:  E:hasil.enc

[6]: kunci = input("Masukkan kata kunci: ")
     Masukkan kata kunci:  harijumat

[7]: xor_cipher_encrypt(input_file, output_file, kunci)

[8]: input_file = input("Masukkan nama file cipherteks:")
     Masukkan nama file cipherteks: E:hasil.enc

[9]: output_file = input("Masukkan nama file untuk menyimpan plainteks: ")
     Masukkan nama file untuk menyimpan plainteks:  E:BPCS-SPIE98-decrypt.pdf

[10]: kunci = input("Masukkan kata kunci: ")
      Masukkan kata kunci:  harijumat

[11]: xor_cipher_decrypt(input_file, output_file, kunci)
```

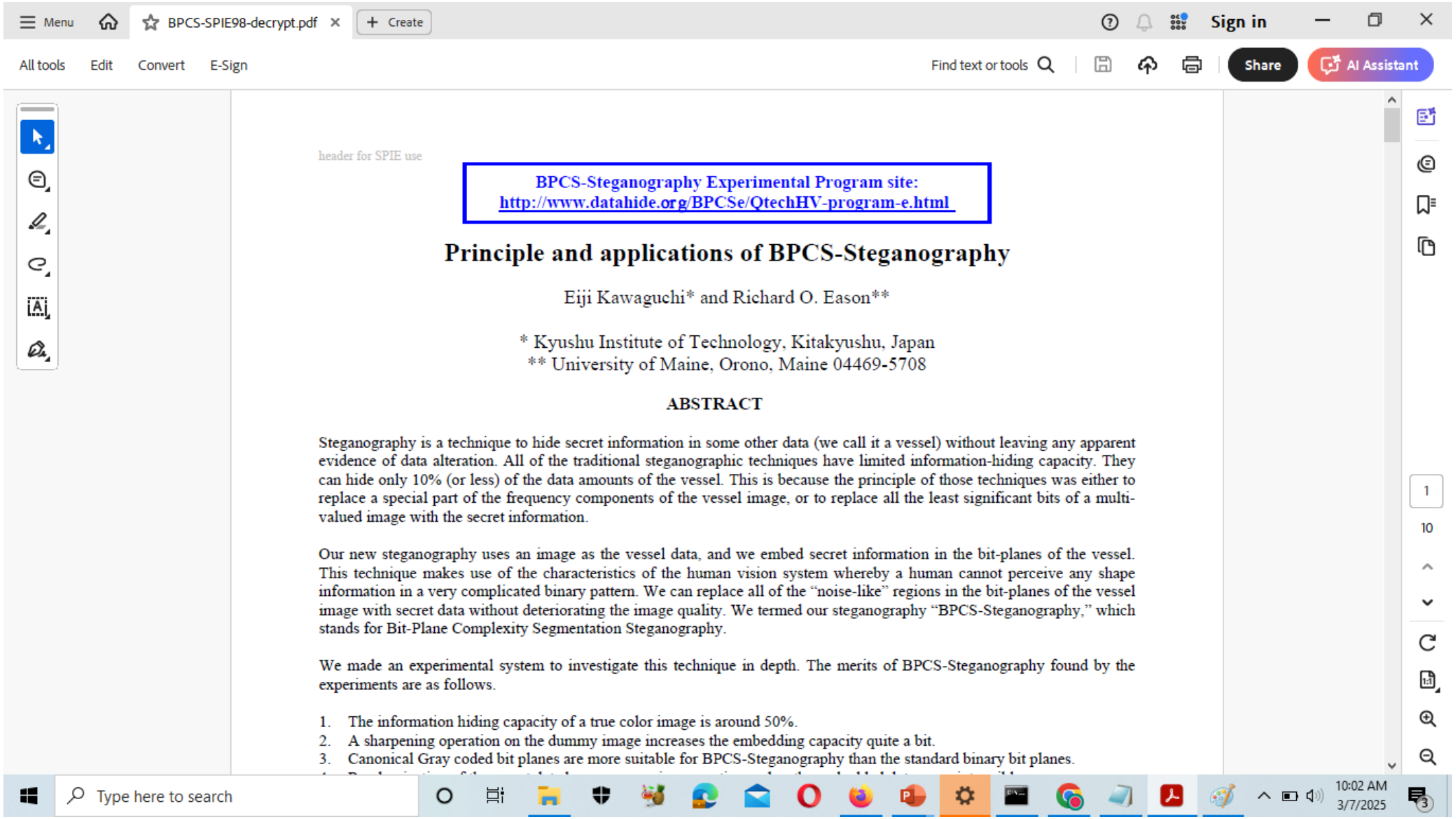
Contoh file plainteks: BPCS-SPIE98.pdf



Contoh hasil file cipherteks: hasil.enc



Hasil dekripsi: BPCS-SPIE98-decrypt.pdf



- *Cipher* XOR sederhana tidak aman, karena mudah dikriptanalisis dengan metode yang sama seperti metode Kasiski

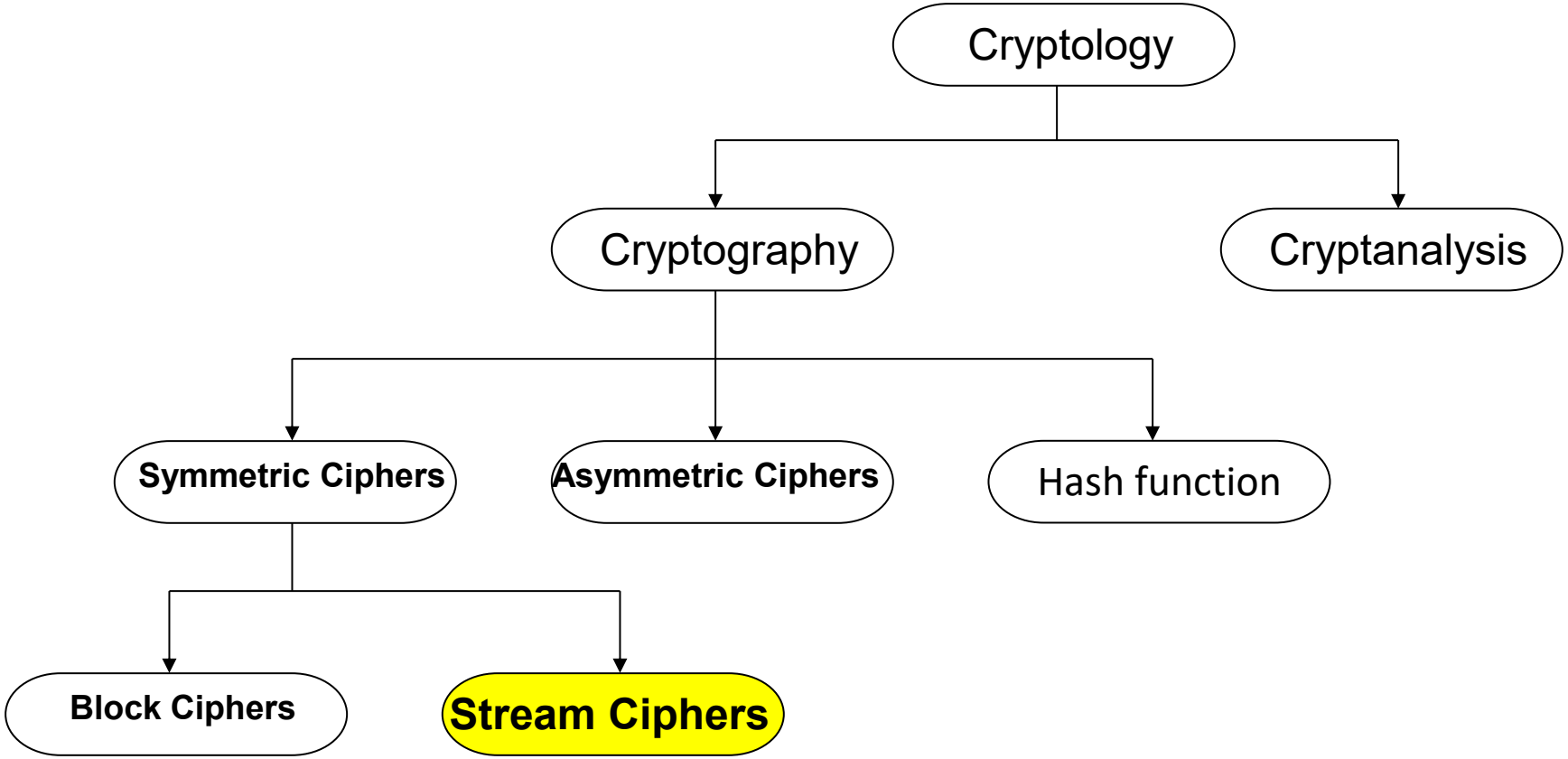
Kategori *cipher* berbasis bit

1. ***Cipher* Alir (*Stream Cipher*)**

   - beroperasi pada bit (atau *byte*) secara individual
   - enkripsi/dekripsi pesan secara bit per bit (atau *byte* per *byte*) hanya dengan operasi XOR saja

2. ***Cipher* Blok (*Block Cipher*)**

   - beroperasi pada blok-blok bit (sekumpulan bit) (contoh: 128-bit/blok = 16 byte/blok)
   - enkripsi/dekripsi pesan secara blok per blok bit dengan komputasi yang lebih kompleks dan dilakukan berulang kali dalam sejumlah putaran.


Cryptology
Cryptography
Cryptanalysis
Symmetric Ciphers
Asymmetric Ciphers
Hash function
Block Ciphers
Stream Ciphers

A. Stream Cipher

Cipher Alir (*Stream Cipher*)

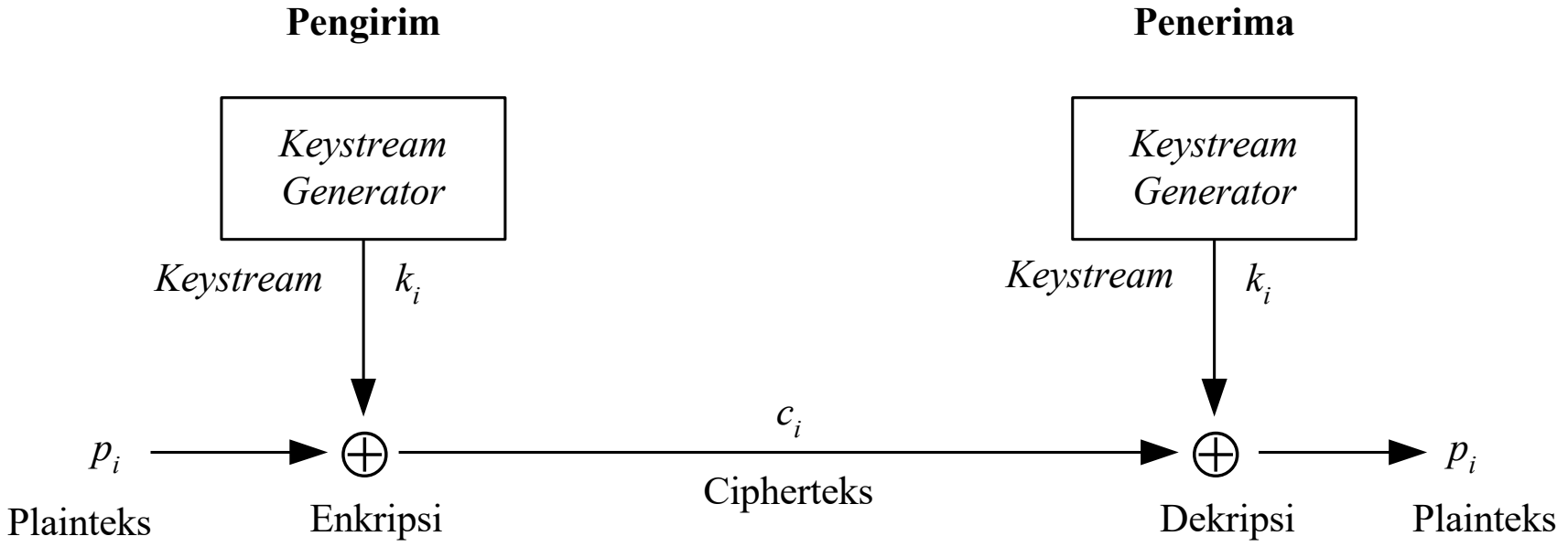
- Mengenkripsi plainteks menjadi chiperteks dengan cara:
  - 1 bit plainteks $\oplus$ 1 bit kunci, atau
  - 1 *byte* plainteks $\oplus$ 1 *byte* skunci, atau
  - *n* bit plainteks $\oplus$ *n* bit kunci

  setiap kali enkripsi

- Dekripsi dilakukan dengan cara:
  - 1 bit cipherteks $\oplus$ 1 bit kunci, atau
  - 1 *byte* cipherteks $\oplus$ 1 *byte* skunci, atau
  - *n* bit cipherteks $\oplus$ *n* bit kunci

  setiap kali dekripsi.

Diagram Umum *Cipher* alir



- Bit-bit aliran kunci untuk enkripsi/dekripsi disebut *keystream*
- *Keystream* dibangkitkan oleh *keystream generator* berdasarkan umpan (*seed*) U.
- Enkripsi dan dekripsi dengan *cipher* alir komputasinya sederhana dan cepat
- *Keystream* c_1, c_2, … di-XOR-kan dengan aliran bit-bit plainteks, p_1, p_2, …, menghasilkan aliran bit-bit cipherteks c_1, c_2, …:

$$c_i = p_i \oplus k_i$$

- Di sisi penerima dibangkitkan *keystream* yang sama untuk mendekripsi aliran bit-bit cipherteks c_1, c_2, … menjadi plainteks, p_1, p_2, …:

$$p_i = c_i \oplus k_i$$

Keystream Generator

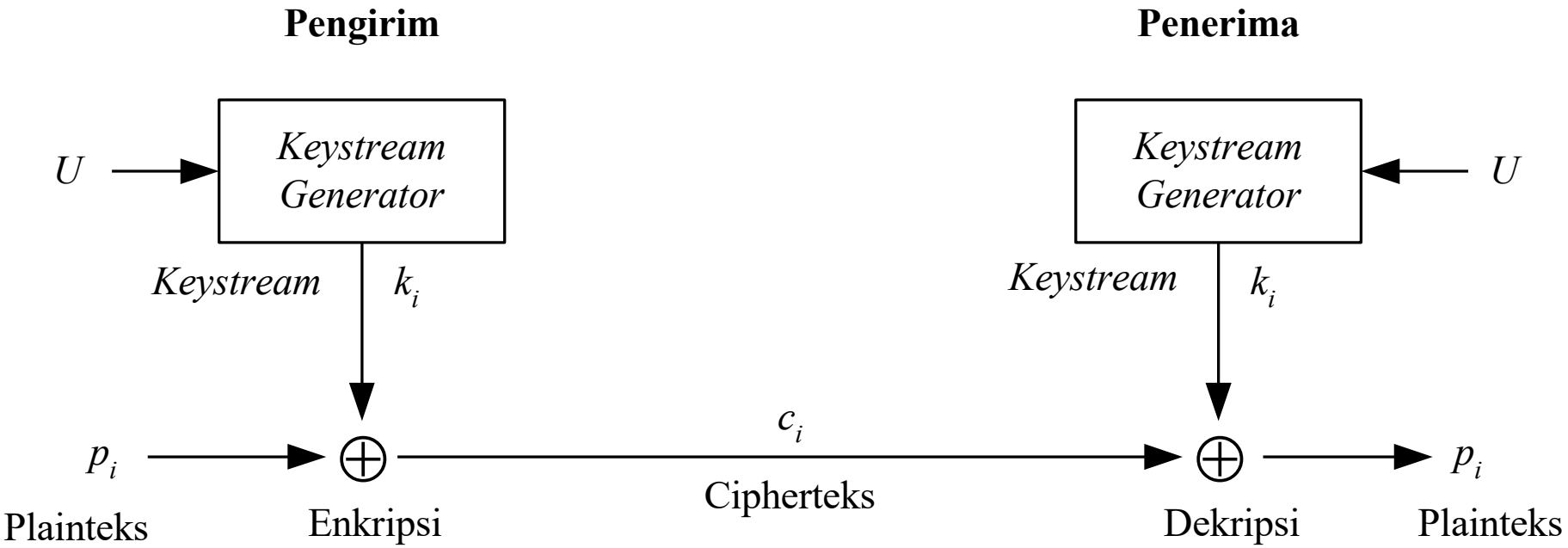
- *Keystream generator* diimplementasikan sebagai prosedur yang sama baik di sisi pengirim maupun di sisi penerima pesan.

- *Keystream generator* dapat membangkitkan *keystream* dalam satuan bit, atau *byte*, atau blok bit.

- Bit-bit *keystream* adalah bit-bit acak, namun bukan *true random*, tetapi *pseudo-random*, karena bit-bit *keystream* dapat dibangkitkan ulang baik di sisi pengirim maupun di sisi penerima pesan asalkan umpan (*seed*) U yang digunakan sama.
- Bit-bit *pseudo-random* memiliki periode (dapat terulang kembali setelah beberapa kali iterasi)

- *Keystream generator* menerima masukan sebuah umpan (*seed*) *U*. Luaran dari prosedur merupakan fungsi dari *U* (lihat Gambar 2). Pengirim dan penerima harus memiliki umpan *U* yang sama. Umpan *U* ini harus dijaga kerahasiaanya. Umpan *U* dapat dianggap sebagai kunci eksternal.

- *Keystream generator* menghasilkan bit-bit kunci yang di-XOR-kan dengan bit plainteks.



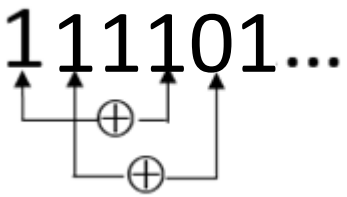
Gambar 2 *Cipher* aliran dengan pembangkit bit kunci-alir yang bergantung pada kunci *U* [MEY82].

- Contoh: Misalkan *U* = 1111

  Algoritma sederhana memperoleh *keystream di dalam keystream* generator misalnya adalah sebagai berikut:

  XOR-kan bit ke-1 dengan bit ke-4 dari empat bit sebelumnya:

  1111010110010001111101011...



  dan akan berulang setiap 15 bit.

- Secara umum, jika panjang *U* adalah *n* bit, maka bit-bit kunci tidak akan berulang sampai $2^n - 1$ bit.

  1 1 1 1 0 1 0 1 1 0 0 1 0 0 0

- Misalkan plainteks adalah:

  P: 0101 0111 0001 1001 0110 0100… (571964 dalam Hexa)

  Keystream yang dibangkitkan:

  K: 11110101100100011101011…

- Enkripsi: C= P ⊕ K

  01010111000110010110010 0…

  11110101100100011101011… ⊕

  1010 0010 1000 1000 1000 1111… (biner)

  A 2 8 8 8 F … (Hex)

- Dekripsi: P = C ⊕ K

  10100010100010001111…

  11110101100100011101011… ⊕

  01010111000110010110010 0… (571964 dalam Hexa)

- Karakteristik *cipher* alir adalah melokalisir kesalahan enkripsi/dekripsi hanya pada bit yang eror saja.
- Maksudnya, kesalahan 1-bit pada cipherteks hanya menghasilkan kesalahan 1-bit pada plainteks hasil dekripsi.
- Misalkan cipherteks 1010 0010 1000 1000 1000 1111 (Hex: A2888F) mengalami eror pada bit ke-5 dari semula 0 menjadi 1: 1010 1010 1000 1000 1000 1111
- Ketika didekripsi:

C: 10101010100010001111…

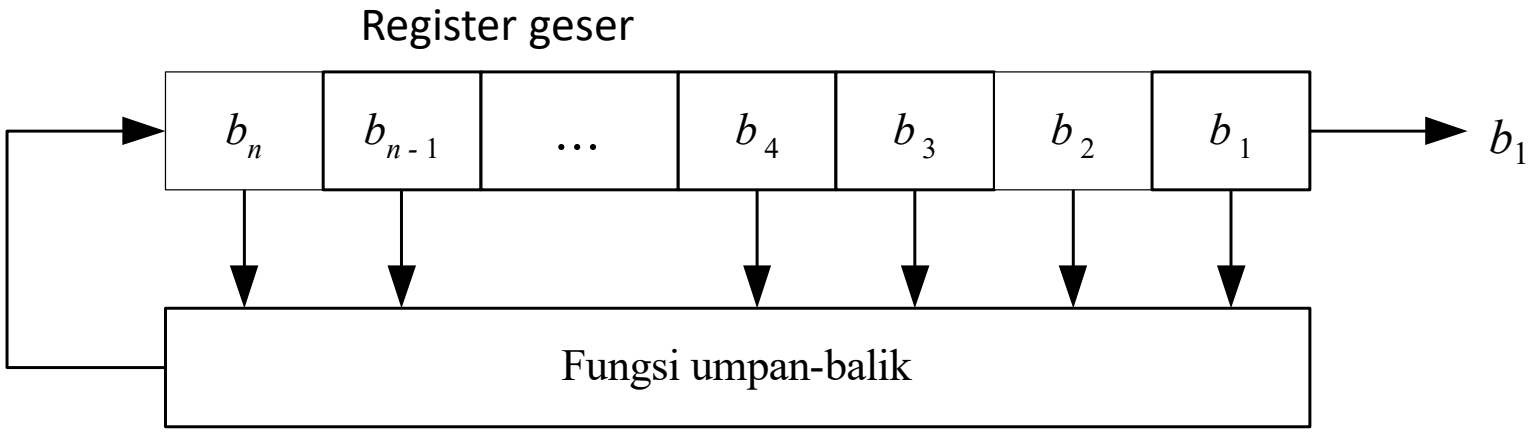
K: 11110101100100011110101 1… ⊕

01011111100011001011001 00… (5F1964 dalam Hexa)

(Seharusnya: 571964 dalam Hexa)

Feedback Shift Register (FSR)

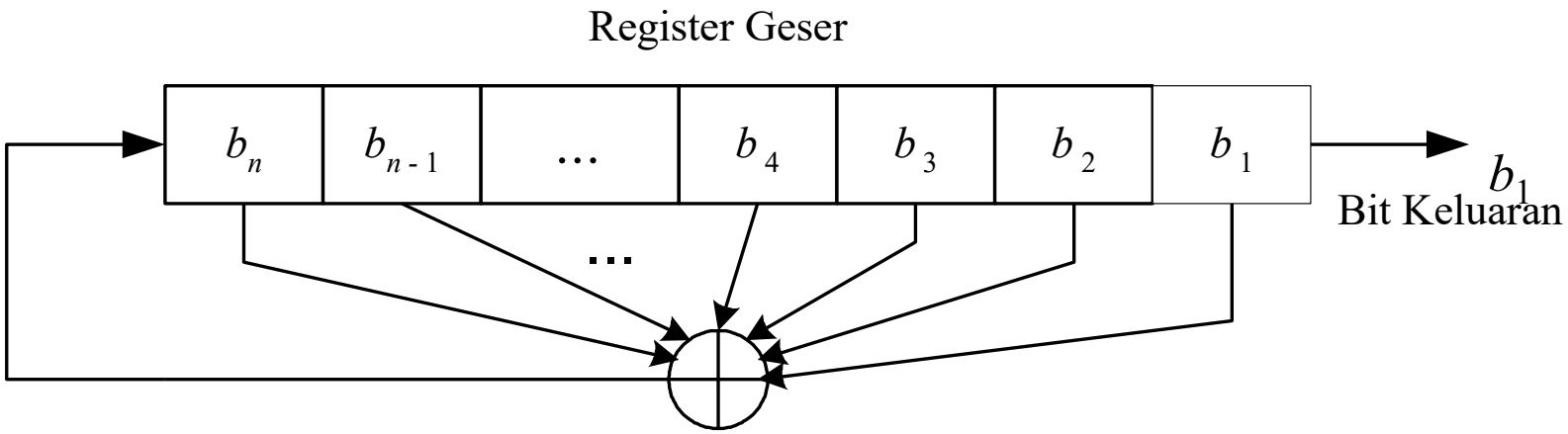
- *FSR* adalah contoh sebuah *keystream generator*.
- Umumnya diimplementasikan sebagai *hardware*.
- *FSR* terdiri dari dua bagian: register geser (n bit) dan fungsi umpan balik



Fungsi umpan baik : $f(b_n, b_{n-1}, b_2, b_1)$

$b_n = f(b_n, b_{n-1}, b_2, b_1)$

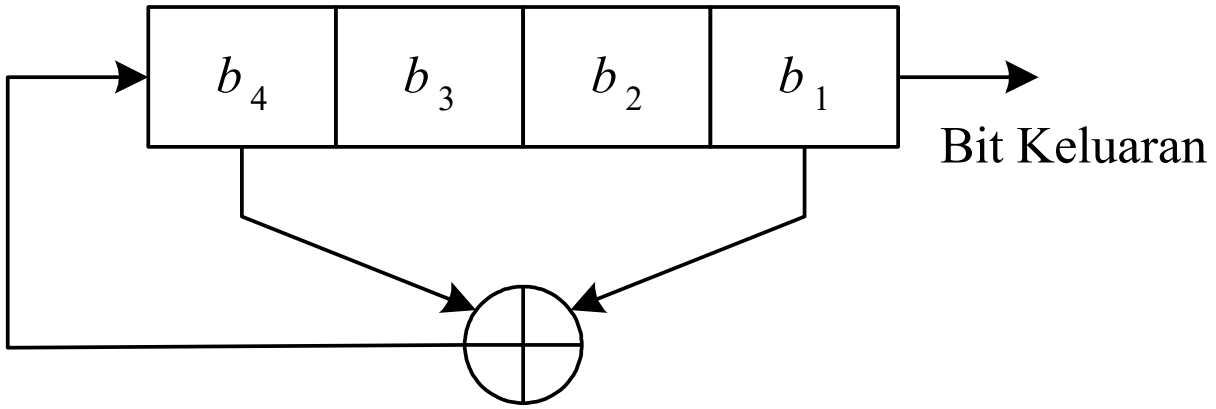
- Contoh FSR adalah LFSR (*Linear Feedback Shift Register*)



$$b_n = b_n \oplus b_{n-1} \oplus \ldots \oplus b_2 \oplus b_1$$

- Bit-bit di dalam register digeser satu bit ke kanan setiap kali pembangkitan bit luaran (= bit kunci alir atau *keystream*)
- Bit b_n digeser ke tempat b_{n-1}, b_{n-1} digeser ke b_{n-2}, dst, b_2 digeser ke b_1, b_1 terlempar ke luar
- Bit yang terlempar keluar (b_1) menjadi bit luaran LFSR
- Selanjutnya dihitung b_n yang baru, yaitu $b_n = b_n \oplus b_{n-1} \oplus \ldots \oplus b_2 \oplus b_1$

- Contoh LFSR 4-bit



- Fungsi umpan balik:

$$b_4 = f(b_1, b_4) = b_1 \oplus b_4$$

- Contoh: jika LFSR 4-bit diinisialisasi dengan U = 1111

i	Isi Register	Bit Keluaran
0	1 1 1 1	
1	0 1 1 1	1
2	1 0 1 1	1
3	0 1 0 1	1
4	1 0 1 0	1
5	1 1 0 1	0
6	0 1 1 0	1
7	0 0 1 1	0
8	1 0 0 1	1
9	0 1 0 0	1
10	0 0 1 0	0
11	0 0 0 1	0
12	1 0 0 0	1
13	1 1 0 0	0
14	1 1 1 0	0
15	1 1 1 1	0

- Barisan bit acak: 1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 …
- Periode LFSR n-bit: $2^n - 1$

Aplikasi *Cipher* Alir

- *Cipher* alir cocok untuk mengenkripsi aliran data yang kontinu melalui saluran komunikasi, misalnya:
  1. Mengenkripsi data pada saluran yang menghubungkan antara dua buah komputer.
  2. Mengenkripsi suara pada jaringan telepon *mobile* GSM.

- Alasan: jika bit cipherteks yang diterima mengandung kesalahan, maka hal ini hanya menghasilkan satu bit kesalahan pada waktu dekripsi, karena tiap bit plainteks ditentukan hanya oleh satu bit cipherteks.

- Selain itu, alasannya karena *cipher* alir itu komputasinya cepat (dibutuhkan untuk *real-time processing*)

Beberapa cipher alir dan tahun pembuatan

- RC4 (1987)
- A5 (1989)
- Rabbit (2003)
- Trivium (2004)
- SEAL (1997)
- Salsa20 (2004)
- Scream (2002)
- SOBER-128 (2003)
- WAKE (2003)
- Turing (2000-2003)
- Scream (2002)
- VEST (2005)
- SNOW (2003)
- CryptMT (2005)
- FISH (1993)
- Grain (2004)
- Isaac (1995)
- MICKEY (2004)

Stream cipher	Creation date	Speed (cycles per byte)	(bits)			Attack	
			Effective key-length	Initialization vector	Internal state	Best known	Computational complexity
A5/1	1989	?	54 or 64 (in 2G)	22 (in 2G)	64	Active KPA OR KPA time–memory tradeoff	~ 2 seconds OR $2^{39.91}$
A5/2	1989	?	54	114	64?	Active	4.6 milliseconds
Achterbahn-128/80	2006	1 (hardware)	80/128	80/128	297/351	Brute force for frame lengths $L \leq 2^{44}$. Correlation attack for $L \geq 2^{48}$.	2^{80} resp. 2^{128} for $L \leq 2^{44}$.
CryptMT	2005	?	Variable	up to 19968	19968	— (2008)	— (2008)
Crypto-1	Pre-1994	?	48	16	48	Active KPA (2008)	40 ms OR 2^{48} (2008)[2]
E0 (cipher)	Pre-1999	?	Variable (usually 128)	4	132	KPA (2005)	2^{38} (2005)[3]
FISH	1993	?	Variable	?	?	Known-plaintext attack	2^{11}
Grain	Pre-2004	?	80	64	160	Key derivation	2^{43}
HC-256	Pre-2004	4 (W_{P4})	256	256	65536	?	?
ISAAC	1996	2.375 ($W_{64\text{-bit}}$) – 4.6875 ($W_{32\text{-bit}}$)	8–8288 (usually 40–256)	—	8288	(2006) First-round weak-internal-state-derivation	4.67×10^{1240} (2001)
MICKEY	Pre-2004	?	80	Variable (0 to 80)	200	Differential Fault Attack (2013)	$2^{32.5}$ (2013)[4]
MUGI	1998–2002	?	128	128	1216	— (2002)	~ 2^{82}

PANAMA	1998	2	256	128?	1216?	Hash collisions (2001)	2^{82}
Phelix	Pre-2004	up to 8 (W_{x86})	256 + a 128-bit nonce	128?	?	Differential (2006)	2^{37}
Pike	1994	?	Variable	?	?	— (2004)	— (2004)
Py	Pre-2004	2.6	8–2048? (usually 40–256?)	64	8320	Cryptanalytic theory (2006)	2^{75}
Rabbit	2003-Feb	3.7(W_{P3}) – 9.7(W_{ARM7})	128	64	512	— (2006)	— (2006)
RC4	1987	7 W_{P5}[5]	8–2048 (usually 40–256)	RC4 does not take an IV. If one desires an IV, it must be mixed into the key somehow.	2064	Shamir initial-bytes key-derivation OR KPA	2^{13} OR 2^{33}
Salsa20	Pre-2004	4.24 (W_{G4}) – 11.84 (W_{P4})	256	a 64-bit nonce + a 64-bit stream position	512	Probabilistic neutral bits method	2^{251} for 8 rounds (2007)
Scream	2002	4–5 (W_{soft})	128 + a 128-bit nonce	32?	64-bit round function	?	?
SEAL	1997	?	?	32?	?	?	?
SNOW	Pre-2003	?	128 or 256	32	?	?	?
SOBER-128	2003	?	up to 128	?	?	Message forge	2^{-6}
SOSEMANUK	Pre-2004	?	128	128	?	?	?
Trivium	Pre-2004	4 (W_{x86}) – 8 (W_{LG})	80	80	288	Brute force attack (2006)	2^{135}
Turing	2000–2003	5.5 (W_{x86})	?	160	?	?	?
VEST	2005	42 (W_{ASIC}) – 64 (W_{FPGA})	Variable (usually 80–256)	Variable (usually 80–256)	256–800	— (2006)	— (2006)
WAKE	1993	?	?	?	8192	CPA & CCA	Vulnerable

(Sumber: https://en.wikipedia.org/wiki/Stream_cipher

1. RC4

- *Cipher* alir yang paling popular, namun sekarang sudah tidak aman
- Dibuat oleh Ronald Rivest (1987) dari Laboratorium *RSA*
- *RC* adalah singkatan dari *Ron's Code*. Versi lain mengatakan *Rivest Cipher* .
- Digunakan di dalam beberapa sistem keamanan seperti:

  - protokol *SSL* (*Secure Socket Layer*) dan TLS (*Transport Layer Security*)

  - Standard IEEE 802.11 *wireless* LAN: WEP (*Wired Equivalent Privacy*)

  - WPA (*Wi-fi Protocol Access*) untuk nirkabel

Ron Rivest

36 languages

Article Talk Read Edit View history

From Wikipedia, the free encyclopedia

"Rivest" redirects here. For other people with the same name, see Rivest (surname).

Ronald Linn Rivest (/rɪˈvɛst/;[3][4] born May 6, 1947) is an American cryptographer and computer scientist whose work has spanned the fields of algorithms and combinatorics, cryptography, machine learning, and election integrity. He is an Institute Professor at the Massachusetts Institute of Technology (MIT),[5] and a member of MIT's Department of Electrical Engineering and Computer Science and its Computer Science and Artificial Intelligence Laboratory.

Along with Adi Shamir and Len Adleman, Rivest is one of the inventors of the RSA algorithm, for which they won the 2002 ACM Turing Award. He is also the inventor of the symmetric key encryption algorithms RC2, RC4, and RC5, and co-inventor of RC6. He also devised the MD2, MD4, MD5 and MD6 cryptographic hash functions.

Ron Rivest

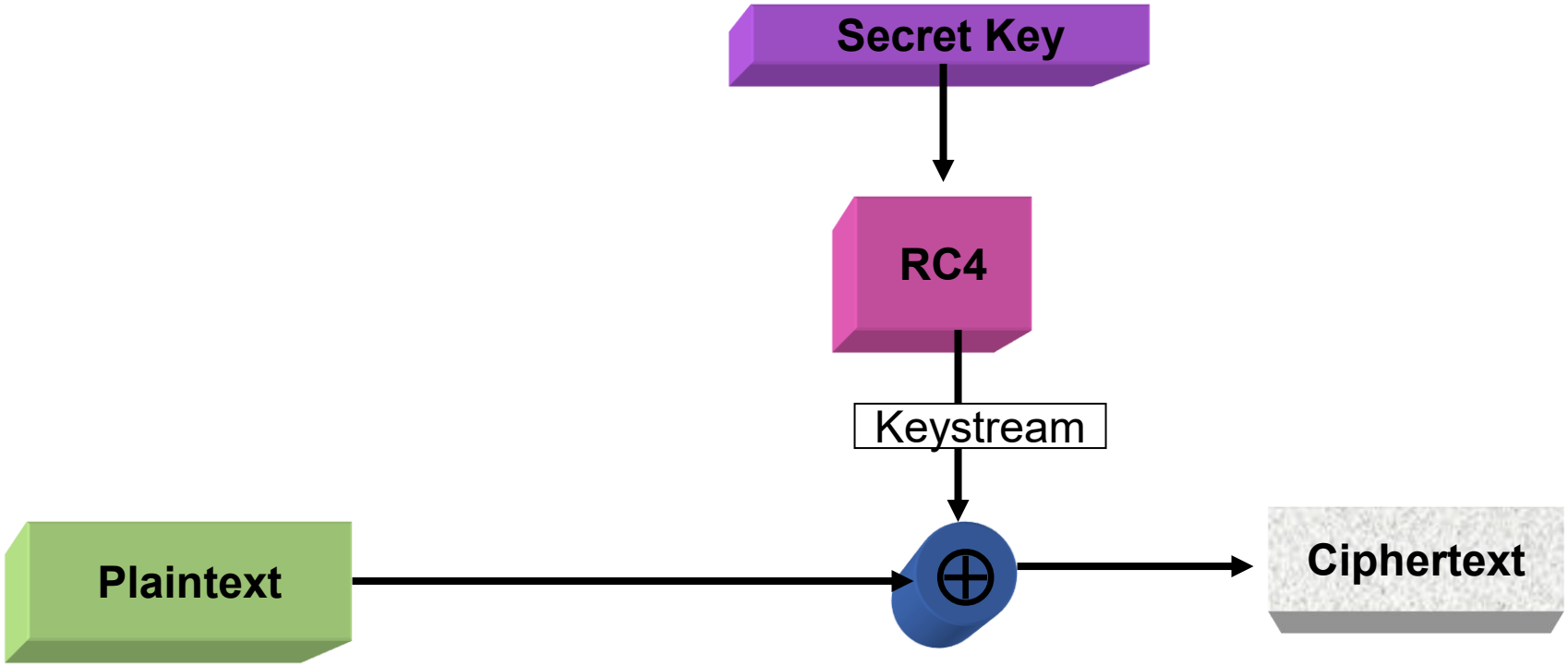
Rivest in 2012

Born	May 6, 1947 (age 79) Schenectady, New York, U.S.
Education	Yale University (BA) Stanford University (MS, PhD)
Known for	Public-key

Education [edit]

Rivest earned a bachelor's degree in mathematics from Yale

Diagram Blok RC4:



- Kunci rahasia (*secret key*) memiliki panjang maksimal 256 karakter (1 karakter = 1*byte*). Jika panjang kunci kurang dari 256 *byte* maka karakter-karakter kunci diulang secara periodik.
- RC4 menghasilkan luaran berupa kunci-alir (*keystream*) dengan panjang tak terbatas

- *RC4* membangkitkan kunci-alir (*keystream*) dalam satuan *byte* setiap kalinya, yang kemudian di-XOR-kan dengan *byte* plainteks (operasi *bitwise* XOR)

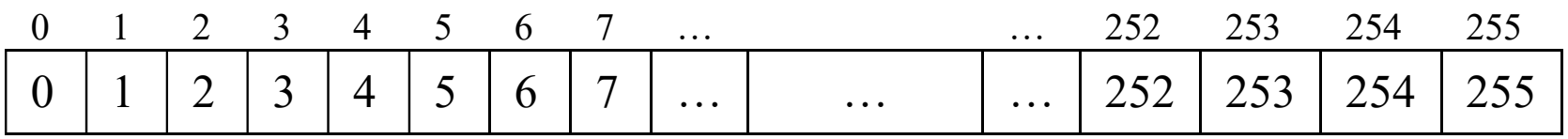
- Untuk membangkitkan kunci-alir, *cipher* menggunakan status internal yang terdiri dari:
  - Larik S_0, S_1, …, S_{255}. Elemen-elemen di dalam larik S dipermutasi berdasarkan kunci rahasia *K* dengan panjang variabel.
  - Dua buah pencacah indeks, *i* dan *j*

- RC4 terdiri dari dua sub-proses:
  1. *Key-Scheduling Algorithm* (*KSA*)
  2. *Pseudo-random generation algorithm* (*PRGA*).

Key-Scheduling Algorithm (*KSA*)

1. Inisialisasi larik S: $S_0 = 0$, $S_1 = 1$, …, $S_{255} = 255$

```
for i ← 0 to 255 do
  S[i] ← i
end
```



3. Lakukan pengacakan (permutasi) nilai-nilai di dalam larik *S* berdasarkan kunci rahasia K (kunci eksternal dari pengguna):

```
j ← 0
for i ← 0 to 255 do
  j ← (j + S[i] + K[i  mod Length(K)]) mod 256
  swap(S[i], S[j]) {* Pertukarkan S[i] & S[j] *}
end
```

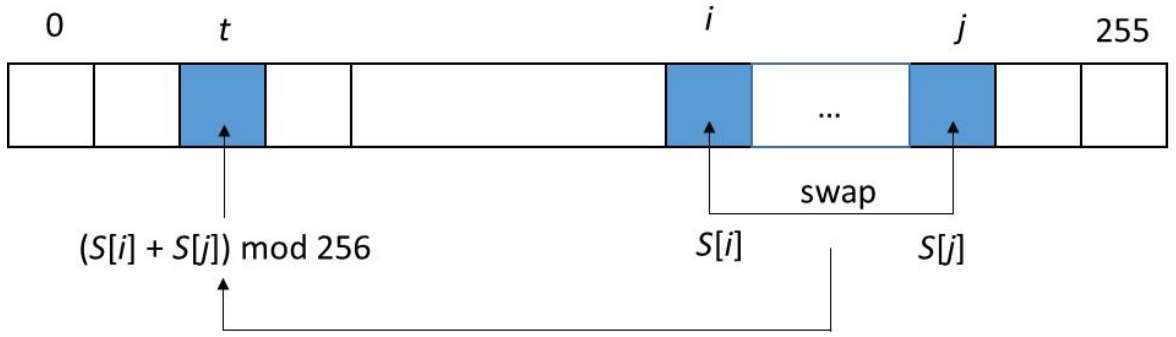
- Permutasi ini menyebabkan elemen-elemen di dalam larik *S* teracak.

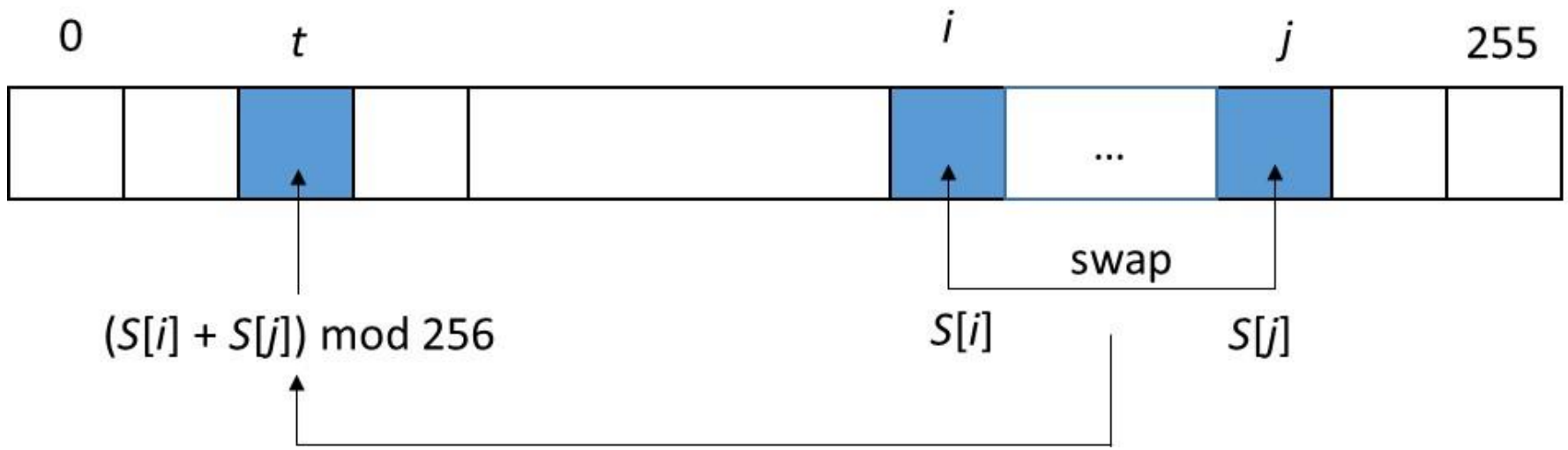
- `K[i mod Length(K)]` menyatakan karakter-karakter kunci diulang secara periodik jika panjangnya kurang dari 256

Pseudo-random generation algorithm (*PRGA*)

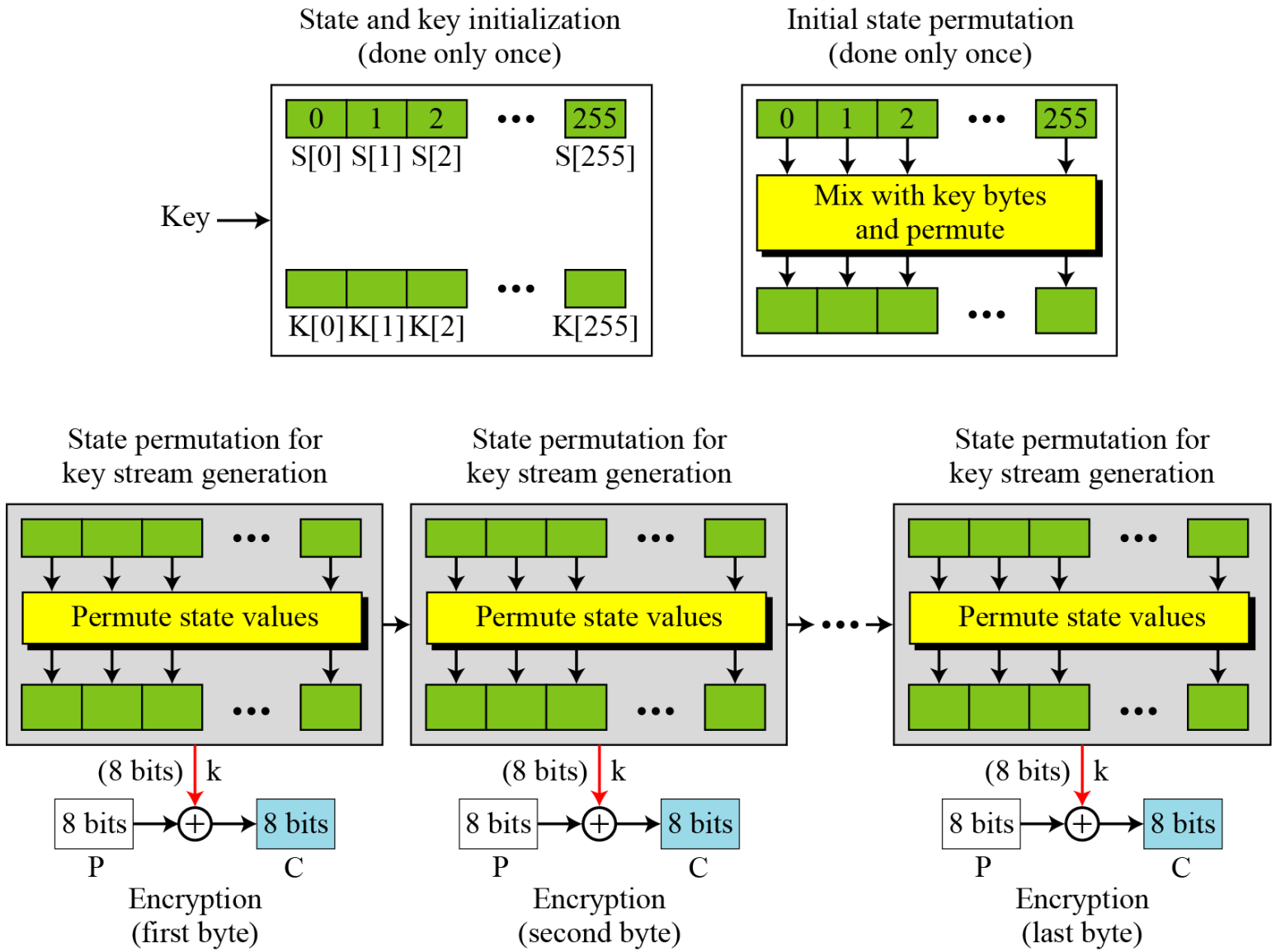
- *PRGA* membangkitkan kunci-alir (*keystream*) dengan cara mengambil nilai *S*[*i*] dan *S*[*j*], mempertukarkannya, lalu menjumlahkan keduanya dalam modulus 256.
- Kunci alir tersebut kemudian di-XOR-kan dengan sebuah karakter plainteks.

```
i ← 0
j ← 0
for idx ← 0 to Length(P) - 1 do
    i ← (i + 1) mod 256
    j ← (j + S[i]) mod 256
    swap(S[i], S[j])    {* Pertukarkan nilai S[i] dan S[j] *}
    t ← (S[i] + S[j]) mod 256
    u ← S[t]            (* keystream *)
    c ← u ⊕ P[idx]      { enkripsi }
end
```




0
t
i
j
255
...
swap
(S[i] + S[j]) mod 256
S[i]
S[j]

- Operasi RC4 secara keseluruhan:



Keamanan RC4

- RC4 memiliki kelemahan, yaitu *byte-byte* pada *keystream* awal pembangkitan memiliki korelasi yang tinggi dengan beberapa *byte* awal kunci
- Oleh karena itu direkomendasikan membuang 256 sampai 512 *byte keystream* awal
- RC4 adalah *cipher* alir, maka ia tidak kuat terhadap serangan seperti *flip-bit attack* maupun serangan-serangan *stream attack* lainnya.
- Saat ini keamanan RC4 sudah berhasil dipecahkan dalam hitungan jam atau hari.
- Pada bulan Februari 2015, RC4 dilarang penggunaannya di dalam *Transport Layer Security* (TLS) seperti disebutkan di dalam RFC 7465 (https://tools.ietf.org/html/rfc7465).
- Beberapa varian RC4 telah dibuat untuk mengatasi kelemahannya, yaitu Spritz, RC4A, VMPC, dan RC4+.

Kode Program RC4 (dalam Bahasa C++)

- Enkripsi

```
// Enkripsi sembarang berkas dengan algoritma RC4.
#include <iostream>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
using namespace std;

main(int argc, char *argv[])
{
 FILE *Fin, *Fout;
 char p, c, u;
 string K;
 int S[256];
 int i, j, t;
```

```
Fin = fopen(argv[1], "rb");
 if (Fin == NULL) {
   cout << "Berkas " << argv[1] <<" tidak ada" << endl;
   exit(0);
 }

 Fout = fopen(argv[2], "wb");

 cout << "Kata kunci : "; cin >> K;
 cout <<"Enkripsi " << argv[1] << " menjadi " << argv[2] << "...";

 for (i = 0; i<256; i++) {
  S[i] = i;
 }
```

```
 j = 0;
 for (i=0; i<256; i++) {
   j = (j + S[i] + K[i % K.length()]) % 256;
   swap(S[i], S[j]); // Pertukarkan nilai S[i] dan S[j]
 }

 i = 0;   j = 0;
while (!feof(Fin)) {
   p = fgetc(Fin);
   i = (i + 1) % 256;
   j = (j + S[i]) % 256;
   swap(S[i], S[j]);    // Pertukarkan nilai S[i] dan S[j]
   t = (S[i] + S[j]) % 256;
   u = S[t];   // keystream
   c = u ^ p;
   fputc(c, Fout);
 }
 fclose(Fin);  fclose(Fout);
}
```

- Dekripsi

```
//Dekripsi sembarang berkas dengan algoritma RC4
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
using namespace std;

main(int argc, char *argv[])
{
 FILE *Fin, *Fout;
 char p, c, u;
 string K;
 int S[256];
 int i, j, t;
```

```
Fin = fopen(argv[1], "rb");
 if (Fin == NULL) {
   cout << "Berkas " << argv[1] <<" tidak ada" << endl;
   exit(0);
 }

 Fout = fopen(argv[2], "wb");

 cout << "Kata kunci : "; cin >> K;
 cout <<"Dekripsi " << argv[1] << " menjadi " << argv[2] << "...";

 for (i = 0; i<256; i++) {
  S[i] = i;
 }
```

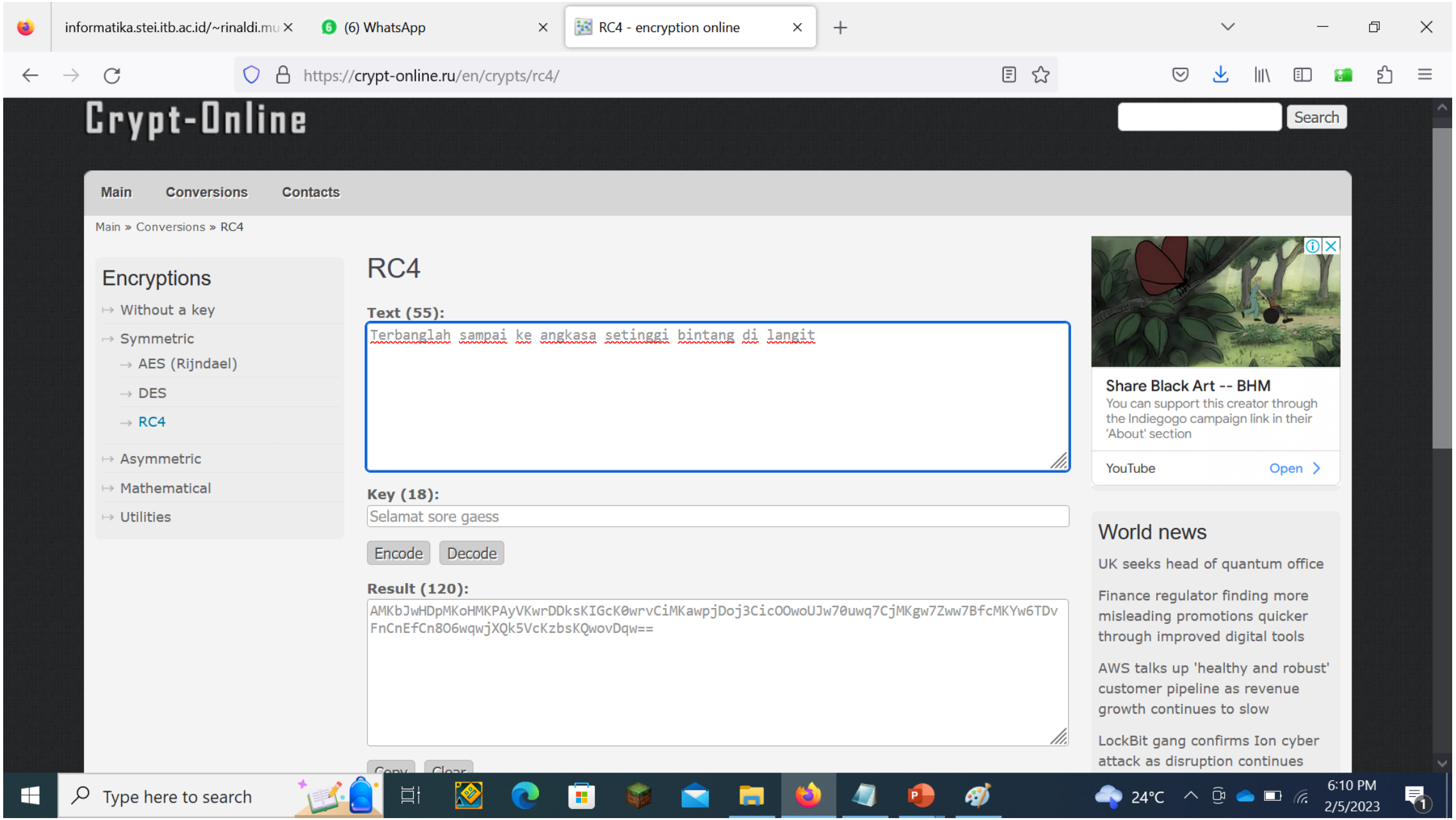
```
j = 0;
 for (i=0; i<256; i++) {
    j = (j + S[i] + K[i % K.length()]) % 256;
    swap(S[i], S[j]); // Pertukarkan nilai S[i] dan S[j]
 }

 i = 0;   j = 0;
while (!feof(Fin)) {
    c = fgetc(Fin);
    i = (i + 1) % 256;
    j = (j + S[i]) % 256;
    swap(S[i], S[j]);    // Pertukarkan nilai S[i] dan S[j]
    t = (S[i] + S[j]) % 256;
    u = S[t];   // keystream
    p = u ^ c;
    fputc(p, Fout);
 }
 fclose(Fin);  fclose(Fout);
}
```

Demo RC4 Online

1. https://crypt-online.ru/en/crypts/rc4/
2. https://www.lddgo.net/en/encrypt/rc4

https://crypt-online.ru/en/crypts/rc4/

Enkripsi



Dekripsi

informatika.stei.itb.ac.id/~rinaldi.mu | (6) WhatsApp | RC4 - encryption online

https://crypt-online.ru/en/crypts/rc4/

Crypt-Online

Search

Main | Conversions | Contacts

Main » Conversions » RC4

Encryptions

- Without a key
- Symmetric
  - AES (Rijndael)
  - DES
  - RC4
- Asymmetric
- Mathematical
- Utilities

RC4

Text (120):

AMKbJwHDpMKoHMKPAyVKwrDDksKIGcK0wrvCiMKawpjDoj3CicOOwoUJw70uwq7CjMKgw7Zww7BfcMKYw6TDv
FnCnEfCn8O6wqwjXQk5VcKzbsKQwovDqw==

Key (18):

Selamat sore gaess

Encode | Decode

Result (55):

Terbanglah sampai ke angkasa setinggi bintang di langit

Share Black Art -- BHM

You can support this creator through the Indiegogo campaign link in their 'About' section

YouTube Open

World news

UK seeks head of quantum office

Finance regulator finding more misleading promotions quicker through improved digital tools

AWS talks up 'healthy and robust' customer pipeline as revenue growth continues to slow

LockBit gang confirms Ion cyber attack as disruption continues

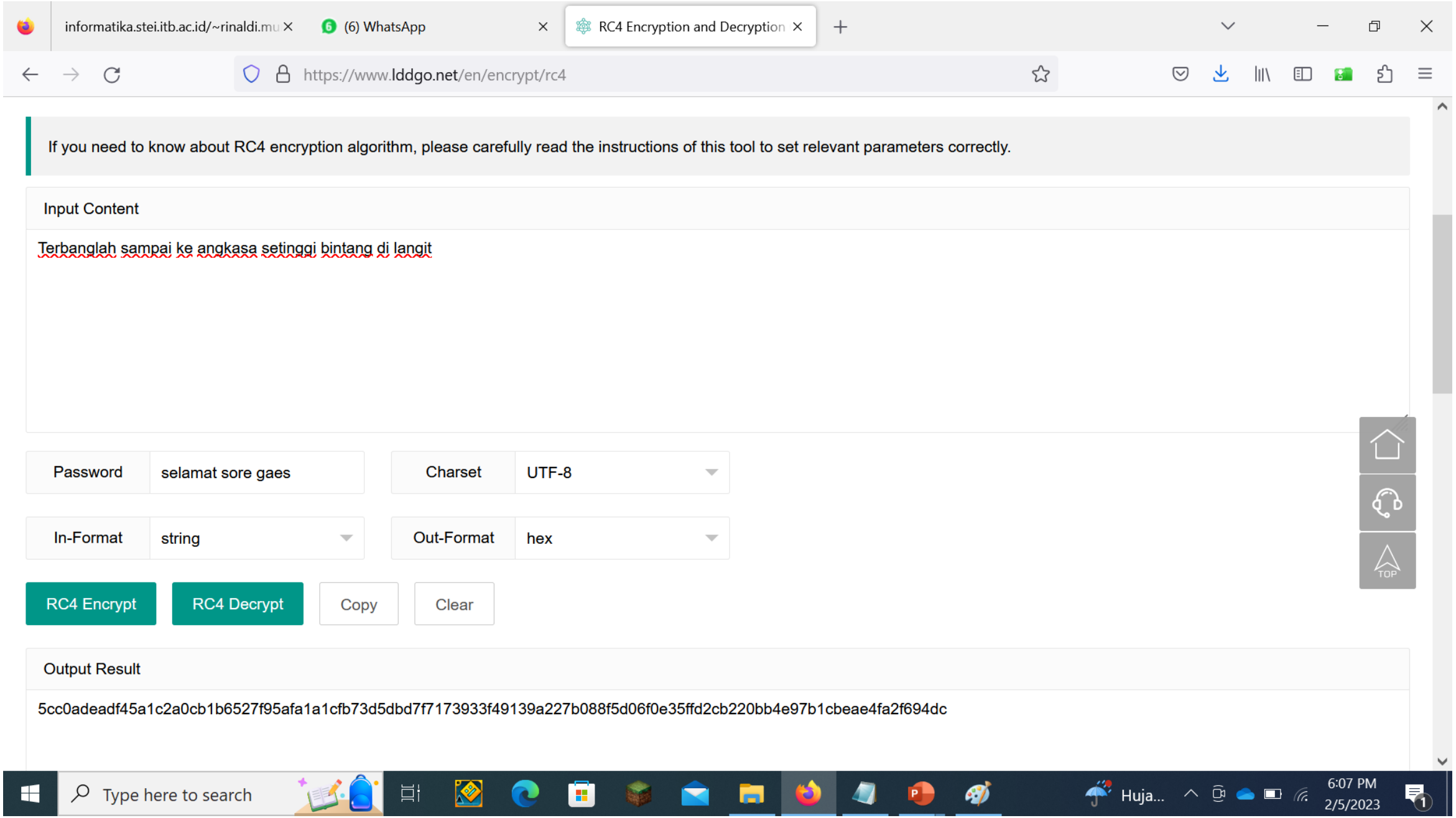
Type here to search

Senja

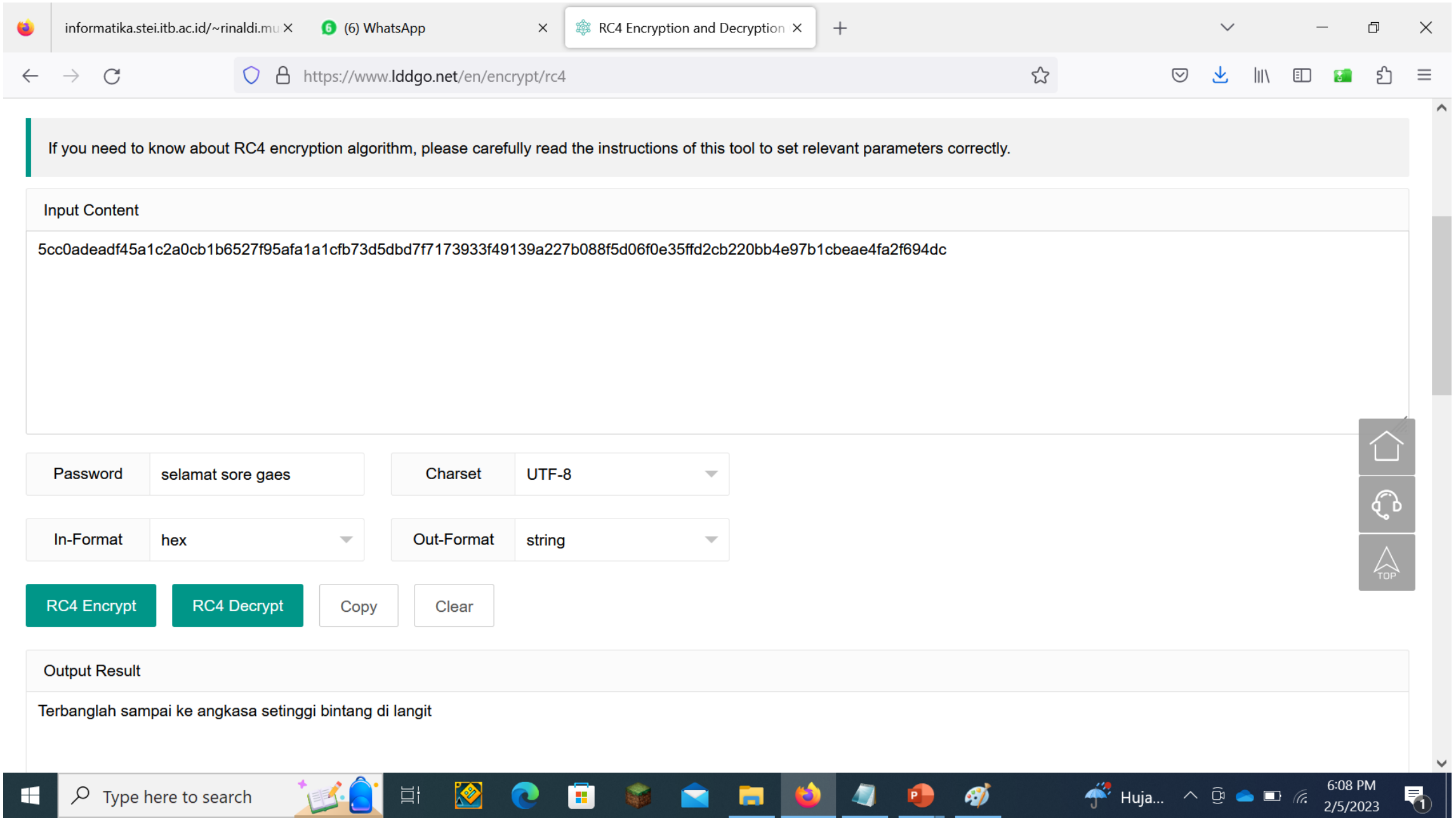
6:12 PM
2/5/2023

https://www.lddgo.net/en/encrypt/rc4

Enkripsi



Dekripsi


informatika.stei.itb.ac.id/~rinaldi.mu
(6) WhatsApp
RC4 Encryption and Decryption
https://www.lddgo.net/en/encrypt/rc4
If you need to know about RC4 encryption algorithm, please carefully read the instructions of this tool to set relevant parameters correctly.
Input Content
5cc0adeadf45a1c2a0cb1b6527f95afa1a1cfb73d5dbd7f7173933f49139a227b088f5d06f0e35ffd2cb220bb4e97b1cbeae4fa2f694dc
Password
selamat sore gaes
Charset
UTF-8
In-Format
hex
Out-Format
string
RC4 Encrypt
RC4 Decrypt
Copy
Clear
Output Result
Terbanglah sampai ke angkasa setinggi bintang di langit
TOP
Type here to search
Huja...
6:08 PM
2/5/2023

Library Crypto.Cipher

- Pustaka `Crypto.Cipher` adalah *package cipher* di dalam Python.

  https://www.pycrypto.org/api/2.6/Crypto.Cipher-module.html

- Di dalamnya terdapat beberapa modul beberapa cipher, salah satunya RC4 dengan nama modul `Crypto.Cipher.ARC4`
- Cara memanggil Pustaka `Crypto.Cipher`

```
>> from Crypto.Cipher import ARC4
```

  dengan asumsi sudah meng-intall *pycryptodome* dengan menggunakan perintah *pip*:

```
>> pip install pycryptodome
```

- Contoh program Python untuk enkripsi pesan teks dengan RC4 pada slide berikut

Kode Program RC4 (dalam Bahasa Python)

```
from Crypto.Cipher import ARC4
import base64

def RC4_encrypt(plainteks : str, kunci : bytes) -> str:
    cipher = ARC4.new(kunci)
    cipherteks = cipher.encrypt(plainteks.encode())
    return base64.b64encode(cipherteks).decode()

def RC4_decrypt(cipherteks : str, kunci : bytes) -> str:
    cipher = ARC4.new(kunci)
    cipherteks_decoded = base64.b64decode(cipherteks)
    return cipher.decrypt(cipherteks_decoded)

pesan = input("Ketikkan pesan rahasia:")
kunci = input("Ketikkan kata kunci: ")
kunci = bytearray(kunci, 'utf-8')
ciphertext = RC4_encrypt(pesan, kunci)
print(f"Cipherteks: {ciphertext}")
plaintext = RC4_decrypt(ciphertext, kunci)
print(f"Plainteks semula: {plaintext}")
```

Latihan

- Ubahlah program di atas sehingga dapat digunakan untuk enkripsi dan dekripsi file sembarang

2. A5

- *A5* adalah *cipher* alir yang digunakan untuk mengenkripsi transmisi sinyal percakapan dari standard sinyal telepon seluler *GSM* (*Group Special Mobile*).

- Sinyal *GSM* dikirim sebagai barisan *frame*. Satu *frame* panjangnya 228 bit dan dikirim setiap 4,6 milidetik.

- *A5* menghasilkan kunci-alir (*keystream*) sepanjang 228-bit yang kemudian bit-bitnya di-*XOR*-kan dengan bit-bit *frame* pesan sepanjang 228 bit. Kunci eksternal (*session key*) panjangnya 64 bit.

- GSM merupakan standard telepon seluler Eropa. A5 dikembangkan oleh Perancis

- Tidak semua operator GSM mengimplementasikan enkripsi, bergantung regulasi (seperti di Indonesia)

- A5 ada dua versi:
    1. A5/1 : versi kuat A5, digunakan di Eropa
    2. A5/2 (Kasumi) : versi ekspor, lebih lemah

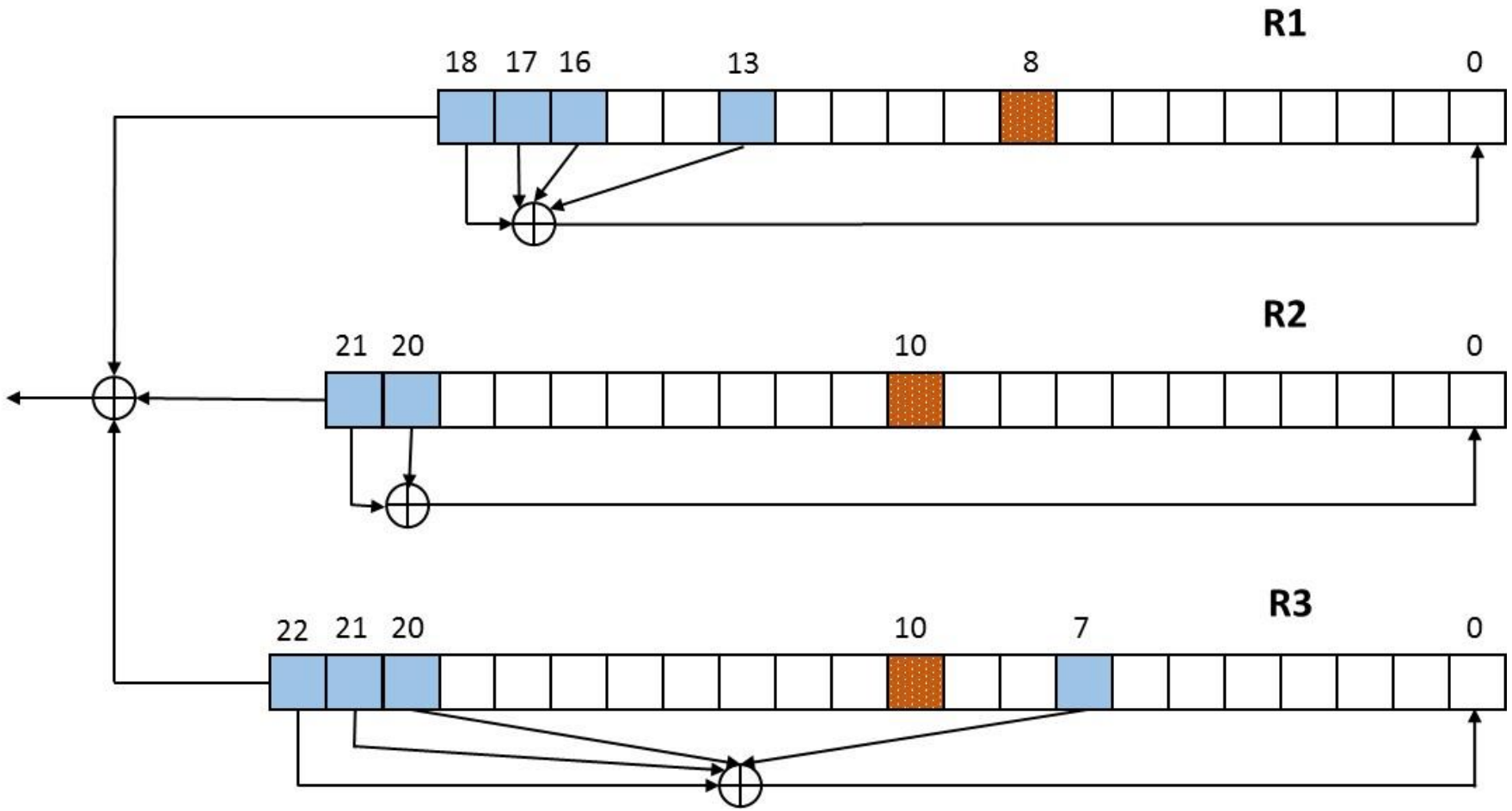
- Algoritma A5/1 pada awalnya rahasia, tetapi pada tahun 1994 melalui *reverse engineering*, algoritmanya terbongkar.

- *A5* terdiri dari 3 buah *LFSR* , masing-masing panjangnya 19, 22, dan 23 bit (total = 19 + 22 + 23 = 64).

- Bit-bit di dalam register diindeks dimana bit paling tidak penting (*LSB*) diindeks dengan 0 (elemen paling kanan).

- Luaran (*output*) dari *A5* adalah hasil *XOR* dari ketiga buah *LFSR* ini.

- *A5* menggunakan tiga buah kendali detak (*clock*) yang variabel:
  - Bit ke-8 pada register 1. Bit-bit detak pada bit 13, 16, 17, dan 18
  - Bit ke-10 pada register 2. Bit-bit detaknya adalah pada bit 20 dan 21
  - Bit ke-10 pada register 3. Bit-bit detaknya adalah pada bit 7, 20, 21, dan 22


R1
18
17
16
13
8
0
R2
21
20
10
0
R3
22
21
20
10
7
0

- Setiap register didetak dalam fase *stop/go* dengan menggunakan kaidah mayoritas:

  "Pada setiap putaran (*i*=1..64), bit-bit tengah setiap register diperiksa dan bit mayoritasnya (50% + 1) ditentukan. Jika bit tengah sebuah register sama dengan bit mayoritas, maka register tersebut didetak"

.

- Biasanya pada setiap putaran dua atau tiga buah register didetak. Peluang sebuah register didetak pada setiap putaran adalah ¾.

- Ketika sebuah register didetak, semua bit detaknya di-XOR-kan dan hasilnya diletakkan pada posisi LSB (posisi ke-0) dengan mekanisme pergeseran bit-bit ke kiri.

- Bit paling kiri (MSB) terlempar keluar. Bit yang terlempar dari setiap register di-XOR-kan bersama, bit inilah yang menjadi luaran dari ketiga buah register tadi.

Proses pembangkitan bit-bit acak di dalam A5/1 berdasarkan kunci sesi *K* adalah sebagai berikut:

1. Ketiga register pada awalnya diinisialisasi seluruh bitnya dengan 0. Selanjutnya dilakukan 64 putaran pertama, yaitu 64 bit kunci sesi *K* dicampur dengan bit register berdasarkan skema berikut:

   Pada putaran $0 \leq i < 64$,

   - bit $K[i]$ ditambahkan ke bit *LSB* dari setiap register *R* dengan menggunakan operasi *XOR*:

     $R[0] = R[0] \oplus K[i]$

   - tiap register kemudian didetak

2. Selanjutnya, ketiga register didetak selama 22 putaran tambahan. Selama 22 putaran tersebut, 22-bit dari nomor *frame* (F_n) dicampur dengan bit register berdasarkan skema berikut:

   Pada putaran $0 \leq i < 22$,

   - bit $F_n[i]$ ditambahkan ke bit *LSB* dari setiap register *R* dengan menggunakan operasi *XOR*:

     $R[0] = R[0] \oplus F_n[i]$

   - tiap register kemudian didetak

Isi register pada akhir putaran menyatakan kondisi awal untuk pembangkitan *frame* sepanjang 228-bit.

3. Ketiga register didetak selama 100 putaran dalam fase *stop/go* dengan menggunakan kaidah mayoritas, namun bit-bit luarannya dibuang (tidak dipakai).

4. Selanjutnya, ketiga register didetak selama 228 putaran dalam fase *stop/go* menggunakan kaidah mayoritas untuk menghasilkan bit-bit kunci-alir sepanjang 228 bit.

   Pada setiap putaran dihasilkan 1 bit yang merupakan hasil peng-XOR-an bit-bit MSB yang terlempar. Total bit luaran adalah 228 bit.

   Kunci-alir 228-bit inilah yang kemudian digunakan untuk mengenkripsi *frame* pesan sepanjang 228 bit.

- Algoritma A5/1 telah digunakan untuk mengenkripsi semua percakapan dan komunikasi data melalui telepon seluler GSM di Eropa.

- Program A5/1 ditanam di dalam *chip* pada kartu *Simcard*.

- Di negara-negara di mana operator telepon seluler dilarang melakukan enkripsi percakapan, seperti di Indonesia, maka program algoritma A5 tidak diaktifkan (*disabled*), sehingga semua sinyal terkirim dalam bentuk plainteks.

Keamanan A5

- Keamanan A5/1 terletak pada pembangkitan bit-bit acak yang dihasilkan oleh tiga buah LFSR di atas.

- Tujuan kriptanalisis A5/1 adalah untuk menemukan kunci sesi yang digunakan dalam pembangkitan bit-bit acak.

- Dalam serangan tersebut diasumsikan penyerang mengetahui luaran A5/1 pada periode awal komunikasi, untuk selanjutnya menemukan kunci sesi yang digunakan untuk mendekripsi sisa komunikasi selanjutnya.

- Serangan yang dilakukan terhadap A5/1 adalah *known-plaintext attack*.

- Serangan semacam ini pertama kali dilakukan oleh Ross Anderson pada tahun 1994.

- Ross Anderson mencoba menerka 42 bit isi register R1 dan R2, dan menurunkan 23 bit R3 dari 42 bit tersebut. Pekerjaan menemukan kunci tersebut dilakukan pada komputer PC dan membutuhkan waktu komputasi lebih dari sebulan.

- Pada tahun-tahun selanjutnya, para kriptanalis berhasil menemukan kunci hanya dalam waktu beberapa menit saja.

- Secara umum, A5/1 sudah berhasil dikriptanalisis.

Interactive Online Simulation of the A5/1 Cipher (https://733amir.github.io/a51-cipher-simulator/)

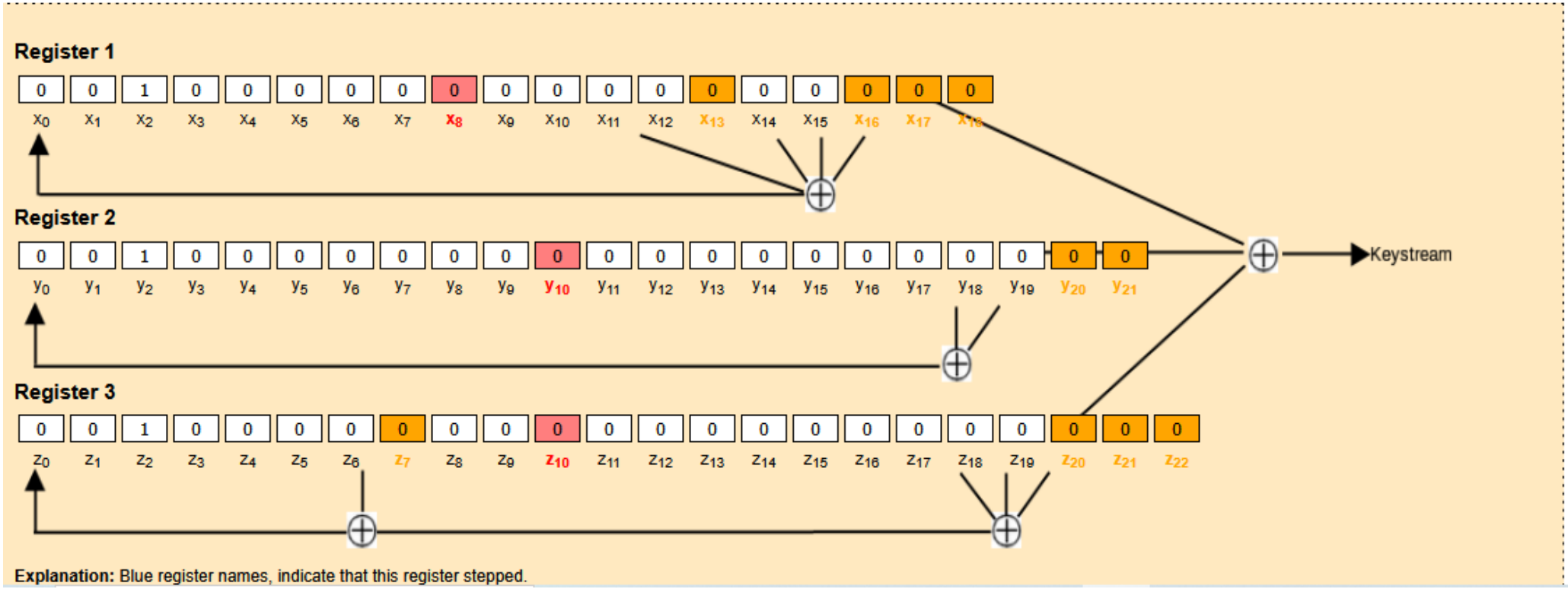
The Key (64 bit)

0100 1110 0010 1111 0100 1101 0111 1100 0001 1110 1011 1000 1000 1011 0011 1010

(1) Reading Key » | (2) Registers Initialization » | 0/100 | (3) Generate Keystream »

$maj(x_8, y_{10}, z_{10}) = maj(0, 0, 0) = 0$

Keystream (0 bits generated)



Explanation: Blue register names, indicate that this register stepped.

3. Trivium

- Trivium adalah *cipher-alir* modern, merupakan salah satu *cipher* alir yang di-submit pada kompetisi *stream cipher* eSTREAM pada tahun 2004 - 2008.

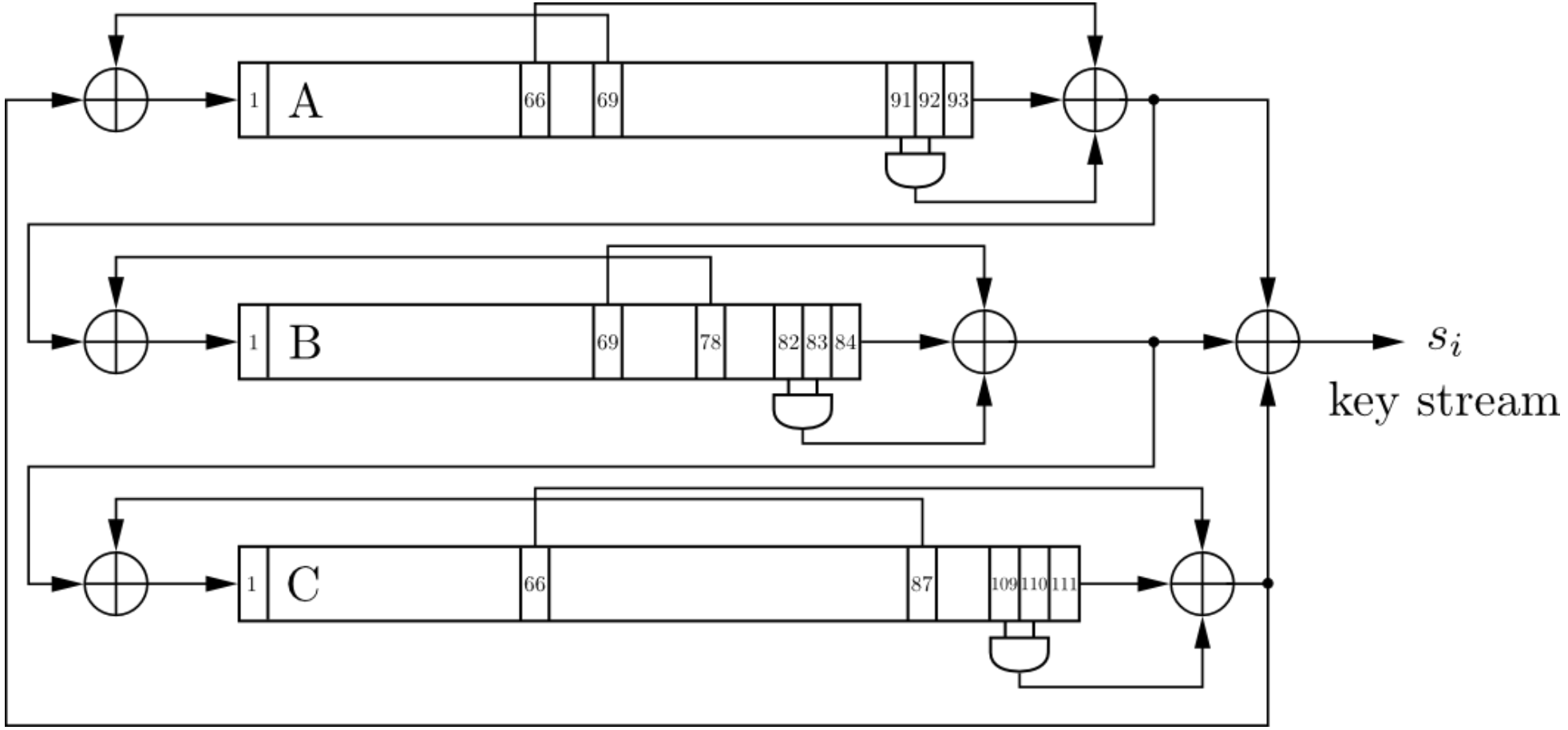
- Dibuat oleh Christophe De Cannière dan Bart Preneel.

- Menggunakan tiga buah *nonlinear LFSRs* (NLFSR) dengan panjang masing-masing 93, 84, dan 111 bit.
- Melakukan XOR terhadap tiga buah luaran NLFSR untuk membangkitkan bit kunci-alir.
- Minimalis jika diimplementasikan pada *hardware*:

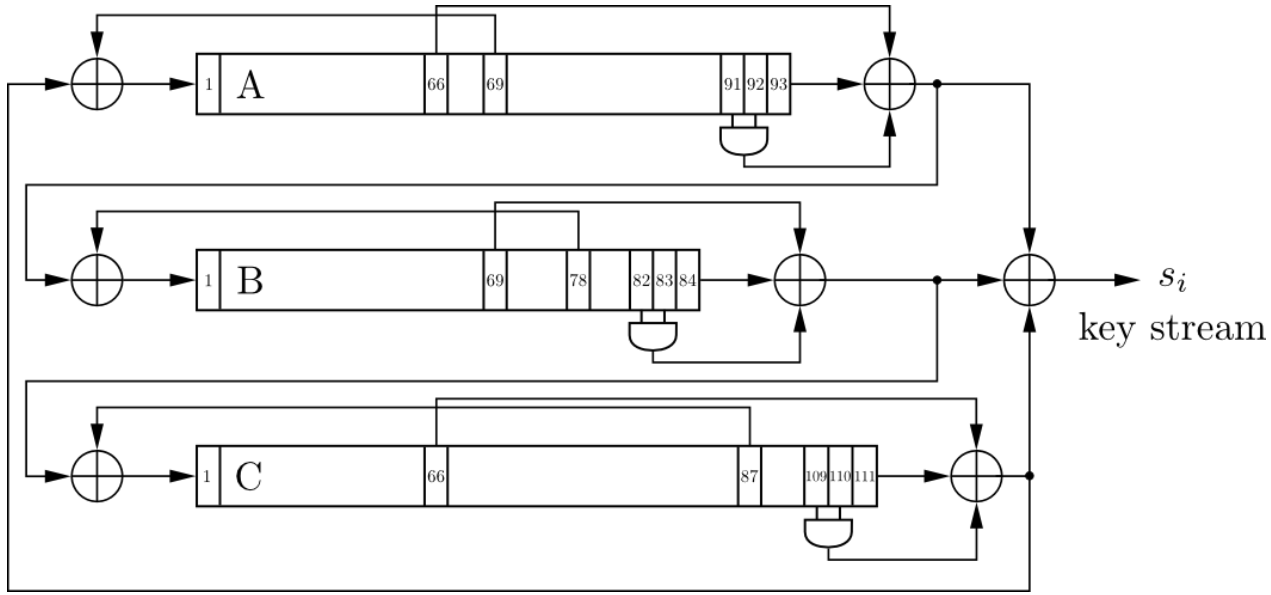
    - total sel register: 288

    - tiga buah gerbang AND

    - tujuh buah gerbang XOR



	Register length	Feedback bit	Feedforward bit	AND inputs
A	93	69	66	91, 92
B	84	78	69	82, 83
C	111	87	66	109, 110



Inisialisasi:

- Masukkan 80-bit *initialization vector* (IV) ke dalam A
- Masukkan 80-bit kunci ke dalam B
- Set c_{109} , c_{110} , c_{111} =1, sedangkan bit lainnya 0

Pemanasan (*warming-up*):

- Detak (*clock*) NLFSR 4 x 288 = 1152 kali tanpa menghasilkan luaran (luaran diabaikan)
- Tujuannya adalah melakukan pencampuran kunci/IV sehingga pengaruh kunci dan IV tersebar ke seluruh register

Encryption:

- XOR-kan ketiga buah luaran NLFSR untuk membangkitkan kunci-alir s_i
- XOR-kan s_i dengan bit plainteks p_i

Kelebihan Trivium

1. Trivium cocok untuk *lightweight cryptography*, karena *state* relatif kecil (hanya 288 bit)

2. *Hardware efficiency*. Sederhana jika diimplementasikan secara *hardware*, karena struktunta sederhana (terutama terdiri dari *flip-flop*/register, gerbang AND dan OR)

4. Salsa20

- Dikembangkan oleh Daniel J. Bernstein pada tahun 2005, merupakan salah satu *cipher* alir yang di-submit pada kompetisi *stream cipher* eSTREAM pada tahun 2004 - 2008.
- Menghasilkan panjang *keystream* 512 bit (atau 64 byte)
- Salsa20 menggunakan:
  - kunci eksternal dari user (*key*) = 256 bit
  - nonce (number used once) = 64 bit (ditentukan nilainya oleh aplikasi, harus unik)
  - counter = 64 bit (dimulai dari 0)
- *Nonce* dan *counter* tidak rahasia, *key* harus rahasia

Daniel J. Bernstein

21 languages

Article Talk Read Edit View history

From Wikipedia, the free encyclopedia

For the American businessman and activist, see Daniel J. Bernstein (businessman).

Daniel Julius Bernstein (born October 29, 1971) is an American mathematician, cryptologist, and computer scientist. He is a professor of computer science at the University of Illinois Chicago.[2] He was a visiting professor in the department of mathematics and computer science at the Eindhoven University of Technology,[3] and a visiting professor at CASA at Ruhr University Bochum through 2023.[4]

Early life [edit]

Bernstein attended Bellport High School, a public high school on Long Island, graduating in 1987 at the age of 15.[5] The same year, he ranked fifth in the Westinghouse Science Talent Search.[6] In 1987, he achieved a Top 10 ranking in the William Lowell Putnam Mathematical Competition,[7] and was a member of the second-place team from Princeton University the following year.[8] Bernstein earned a B.A. in mathematics from New York University (1991) and a Ph.D. in mathematics from the University of California, Berkeley (1995), where he studied under Hendrik Lenstra.[1]

Daniel J. Bernstein

Bernstein at 27C3 in 2010

Born	October 29, 1971 (age 54) East Patchogue, New York[1]

The Salsa20 family of stream ciphers

Daniel J. Bernstein *

Department of Mathematics, Statistics, and Computer Science (M/C 249)
The University of Illinois at Chicago
Chicago, IL 60607–7045
snuffle6@box.cr.yp.to

Abstract. Salsa20 is a family of 256-bit stream ciphers designed in 2005 and submitted to eSTREAM, the ECRYPT Stream Cipher Project. Salsa20 has progressed to the third round of eSTREAM without any changes. The 20-round stream cipher Salsa20/20 is consistently faster than AES and is recommended by the designer for typical cryptographic applications. The reduced-round ciphers Salsa20/12 and Salsa20/8 are among the fastest 256-bit stream ciphers available and are recommended for applications where speed is more important than confidence. The fastest known attacks use $\approx 2^{153}$ simple operations against Salsa20/7, $\approx 2^{249}$ simple operations against Salsa20/8, and $\approx 2^{255}$ simple operations against Salsa20/9, Salsa20/10, etc. In this paper, the Salsa20 designer presents Salsa20 and discusses the decisions made in the Salsa20 design.

1 Introduction

A sender and receiver share a short secret key. They use the secret key to encrypt a series of messages. A message could be short, just a few bytes, but it could be

Step 1: Tentukan parameter masukan (key, nonce, counter)

Step 2: Inisialisasi matriks *state* berukuran 4 x 4, setiap elemen matriks berukuran 1 word, 1 word = 32 bit. Total ukuran matriks 16 x 32 bit = 512 bit. *State* berisi konstanta (‘expand 32-byte k’), *key*, *counter*, dan *nonce* dengan pengaturan letak sebagai berikut:

Matriks *state*

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Initial state of Salsa20

"expa"	Key	Key	Key
Key	"nd 3"	Nonce	Nonce
Pos.	Pos.	"2-by"	Key
Key	Key	Key	"te k"

Catatan: Di dalam kdoe program, matriks *state* dinyatakan sebagai sebuah larik in[0..15]

Step 3: Simpan salinan state awal ke dalam larik x

```
for (i = 0; i < 16; ++i)
    x[i] = in[i];
```

Step 4: Lakukan 20 putaran (= 10 putaran ganda), satu putaran ganda terdiri dari 10 *column round* dan 10 *row round*.

Double-round 1
→ Column Round
→ Row Round

Double-round 2
→ Column Round
→ Row Round

...

Double-round 10
→ Column Round
→ Row Round

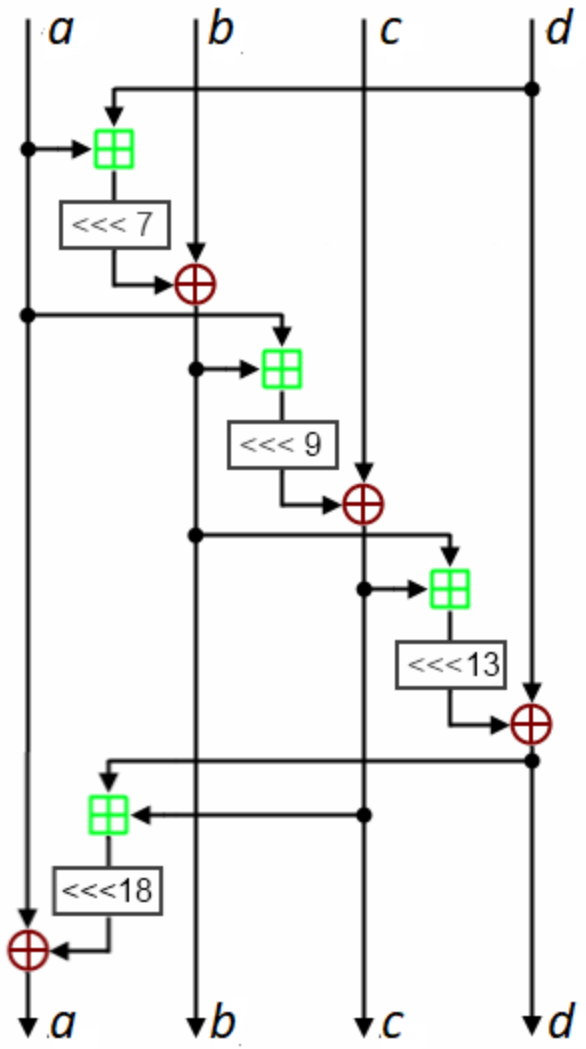
- *Column round* dan *row round* dilakukan di dalam sebuah struktur *quarter round* QR(a, b, c, d) yang menerima empat *word* sebagai masukan dan menghasilkan empat word luaran:

  $b \leftarrow b \oplus (a + d) <<< 7;$

  $c \leftarrow c \oplus (b + a) <<< 9;$

  $d \leftarrow d \oplus (c + b) <<< 13;$

  $a \leftarrow a \oplus (d + c) <<< 18;$



<<< *k* adalah pergeseran bit ke kiri

sejauh k bit dan bersifat rotasi (siklik)

+ adalah penjumlahan bit dalam mod 2^{32}

***Quarter Round* merupakan inti Salsa20**

- *Column round* menerapkan QR(a, b, c, d) kepada tiap kolom matriks *state* 4×4, sedangkan *row round* menerapkannya kepada tiap baris matriks.

// Column round

QR(0, 4, 8, 12) // kolom 1
QR(5, 9, 13, 1) // kolom 2
QR(10, 14, 2, 6) // kolom 3
QR(15, 3, 7, 11) // kolom 4

QR(a, b, c, d):
b ← b ⊕ (a + d) <<< 7;
c ← c ⊕ (b + a) <<< 9;
d ← d ⊕ (c + b) <<< 13;
a ← a ⊕ (d + c) <<< 18;

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

// Row round

QR(0, 1, 2, 3) // baris 1
QR(5, 6, 7, 4) // baris 2
QR(10, 11, 8, 9) // baris 3
QR(15, 12, 13, 14) // baris 4

```
#define ROTL(a,b) (((a) << (b)) | ((a) >> (32 - (b))))
#define QR(a, b, c, d)(
        b ^= ROTL(a + d, 7), \
        c ^= ROTL(b + a, 9), \
        d ^= ROTL(c + b,13), \
        a ^= ROTL(d + c,18))
```

Step 5: State hasil 10 putaran-ganda dijumlahkan dengan state awal yang disimpan pada step 3:

$$y = (x' + x) \bmod 2^{32}$$

Jika plainteks berukuran n x 512 bit, maka Salsa20 dieksekusi sebanyak n kali, dengan *counter* bertambah 1, sedangkan *nonce* tetap.

Kode program C++:

```
#include <stdint.h>
#define ROTL(a,b) (((a) << (b)) | ((a) >> (32 - (b))))
#define QR(a, b, c, d)(  \
        b ^= ROTL(a + d, 7), \
        c ^= ROTL(b + a, 9), \
        d ^= ROTL(c + b,13), \
        a ^= ROTL(d + c,18))
#define ROUNDS 20
```

```
void salsa20_block(uint32_t out[16], uint32_t const in[16])
{
	int i;
	uint32_t x[16];

	for (i = 0; i < 16; ++i)
		x[i] = in[i];
	// 10 loops × 2 rounds/loop = 20 rounds
	for (i = 0; i < ROUNDS; i += 2) {
		// Column round
		QR(x[ 0], x[ 4], x[ 8], x[12]); // column 1
		QR(x[ 5], x[ 9], x[13], x[ 1]); // column 2
		QR(x[10], x[14], x[ 2], x[ 6]); // column 3
		QR(x[15], x[ 3], x[ 7], x[11]); // column 4
		// Row round
		QR(x[ 0], x[ 1], x[ 2], x[ 3]); // row 1
		QR(x[ 5], x[ 6], x[ 7], x[ 4]); // row 2
		QR(x[10], x[11], x[ 8], x[ 9]); // row 3
		QR(x[15], x[12], x[13], x[14]); // row 4
	}
	for (i = 0; i < 16; ++i)
		out[i] = x[i] + in[i];
}
```

Sumber: Wikipedia

Kelebihan Salsa20

1. Sangat cepat

Salsa20 hanya menggunakan operasi yang sangat sederhana (AXR):

- penjumlahan (*add*) modulo 2^{32},

- XOR,

- rotasi bit.

2. Memiliki keamanan yang baik

Salsa20 dirancang dengan 20 putaran untuk memberikan difusi yang kuat. Maksudnya, perubahan kecil pada *key*, *nonce*, *counter*, akan menghasilkan perubahan yang besar pada keystream.

3. Cocok untuk perangkat dengan sumber daya terbatas

Karena operasi internalnya sederhana, Salsa20 dapat diimplementasikan dengan baik pada berbagai perangkat, termasuk perangkat yang kemampuan komputasinya terbatas.

4. Cocok untuk implementasi perangkat lunak

Salsa20 dapat memperoleh performa yang baik hanya dengan operasi integer dasar. Karena itu Salsa20 sangat populer dalam komunitas kriptografi, dan desainnya kemudian menjadi dasar bagi **ChaCha**, yang akhirnya lebih banyak digunakan dalam protokol modern.

5. Menghasilkan blok *keystream* yang besar

Satu blok Salsa20 menghasilkan 16 × 32 bit = 512 bit = 64 byte keystream. Jika plaintext panjang, counter tinggal dinaikkan untuk menghasilkan blok berikutnya.

6. *Keystream* dapat dihasilkan secara paralel

Salah satu karakteristik penting Salsa20 adalah setiap blok *keystream* bergantung pada *key*, *nonce*, dan *counter*.

Secara konseptual:

Key + Nonce + Counter 0 → Keystream 0

Key + Nonce + Counter 1 → Keystream 1

Key + Nonce + Counter 2 → Keystream 2

Key + Nonce + Counter 3 → Keystream 3

Blok-blok tersebut pada prinsipnya dapat dihitung secara independen. Ini berbeda dari beberapa konstruksi *stream cipher* yang memiliki ketergantungan berantai antar keystream.

ChaCha20

- Chacha20 merupakan modifikasi dari Salsa20 yang dipublikasikan oleh Bernstein pada tahun 2008. ChaCha20 merupakan keluarga ChaCha (ChaCha lainnya: Chacha6, ChaCha7)
- Alasannya terutama adalah untuk memperoleh difusi yang lebih baik pada struktur internal sekaligus mempertahankan kecepatan dan kesederhanaan desain ARX (add, rotate, xor).

- ChaCha20 Mempertahankan keamanan Salsa20 sambil memperbaiki karakteristik desain
- Salah satu tujuan penting desain ChaCha adalah memperoleh implementasi yang sangat efisien pada perangkat lunak.
- ChaCha20 menjadi sangat populer dalam protokol modern

Perbedaan ChaCha dengan Salsa:

1. Initial state

Initial state of ChaCha

"expa"	"nd 3"	"2-by"	"te k"
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Counter	Nonce	Nonce

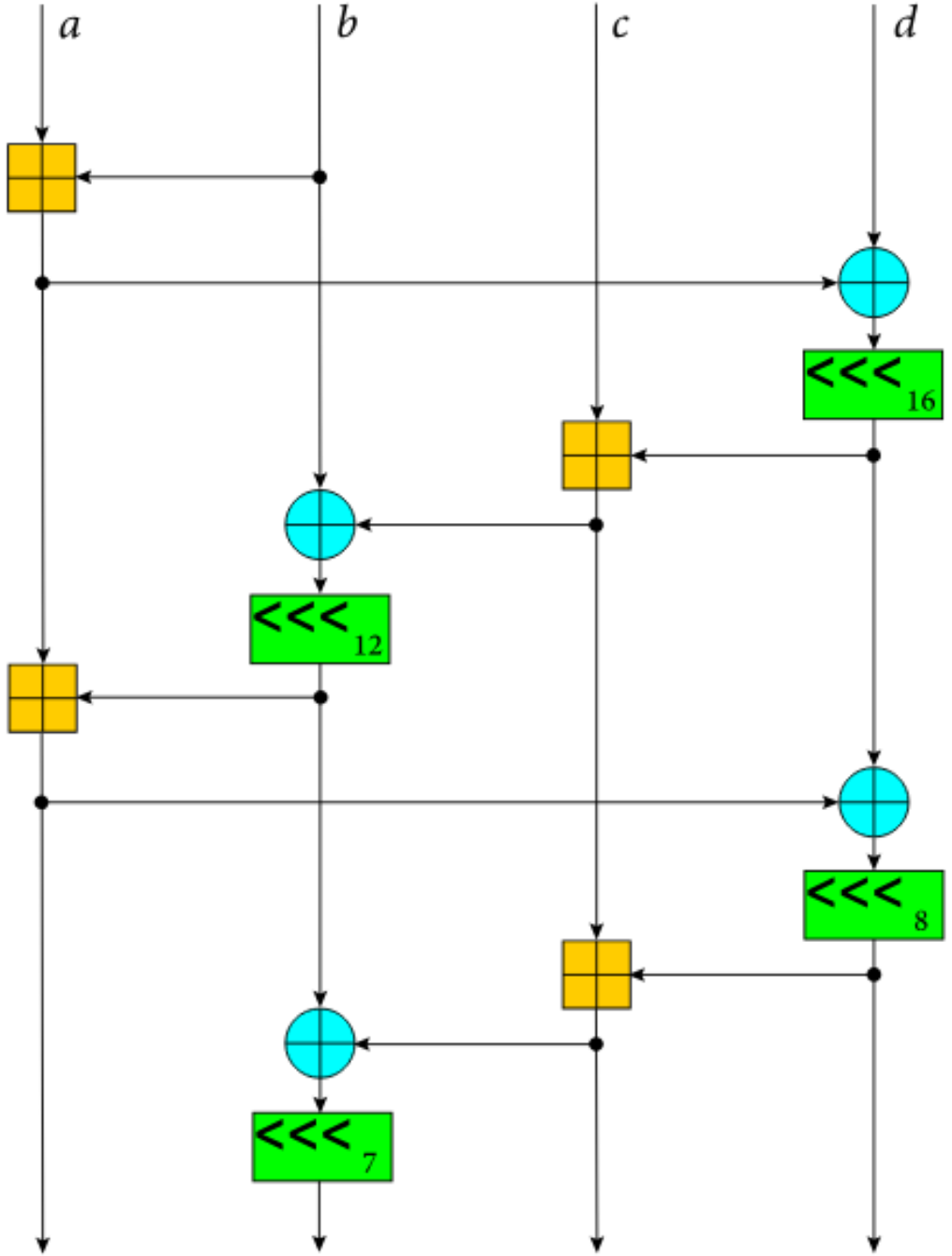
2. Quarter round QR

$a \leftarrow a + b;\quad d \leftarrow d \oplus a;\quad d \leftarrow d \lll 16;$

$c \leftarrow c + d;\quad b \leftarrow b \oplus c;\quad b \leftarrow b \lll 12;$

$a \leftarrow a + b;\quad d \leftarrow d \oplus a;\quad d \leftarrow d \lll 8;$

$c \leftarrow c + d;\quad b \leftarrow b \oplus c;\quad b \leftarrow b \lll 7;$



3. *Column round* dan *diagonal round*

```
// Column round
QR(0, 4,  8, 12)        // kolom 1
QR(1, 5,  9, 13)        // kolom 2
QR(2, 6, 10, 14)        // kolom 3
QR(3, 7, 11, 15)        // kolom 4

// Diagonal round
QR(0, 5, 10, 15)        // diagonal 1
QR(1, 6, 11, 12)        // diagonal 2
QR(2, 7,  8, 13)        // diagonal 3
QR(3, 4,  9, 14)        // diagonal 4
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

```
#define ROTL(a,b) (((a) << (b)) | ((a) >> (32 - (b))))
#define QR(a, b, c, d) (                                \
        a += b,  d ^= a,  d = ROTL(d,16),               \
        c += d,  b ^= c,  b = ROTL(b,12),               \
        a += b,  d ^= a,  d = ROTL(d, 8),               \
        c += d,  b ^= c,  b = ROTL(b, 7))
#define ROUNDS 20
```

Sumber: Wikipedia

```
void chacha_block(uint32_t out[16], uint32_t const in[16])
{
        int i;
        uint32_t x[16];

        for (i = 0; i < 16; ++i)
                x[i] = in[i];
        // 10 perulangan × 2 ronde/perulangan = 20 ronde
        for (i = 0; i < ROUNDS; i += 2) {
                // Ronde ganjil
                QR(x[0], x[4], x[ 8], x[12]); // kolom 0
                QR(x[1], x[5], x[ 9], x[13]); // kolom 1
                QR(x[2], x[6], x[10], x[14]); // kolom 2
                QR(x[3], x[7], x[11], x[15]); // kolom 3
                // Ronde genap
                QR(x[0], x[5], x[10], x[15]); // diagonal 1
                QR(x[1], x[6], x[11], x[12]); // diagonal 2
                QR(x[2], x[7], x[ 8], x[13]); // diagonal 3
                QR(x[3], x[4], x[ 9], x[14]); // diagonal 4
        }
        for (i = 0; i < 16; ++i)
                out[i] = x[i] + in[i];
}
```

Sekian