Studi Lengkap Mengenai *Rabbit Cipher*

Andree Datta Adwitya – NIM : 13503062

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : datta@students.if.itb.ac.id

Abstrak

Makalah ini membahas tentang studi dari *Rabbit Cipher*, jenis *stream cipher* baru yang dikembangkan oleh Cryptico A/S. Cara kerja *cipher* ini mirip seperti *stream cipher* pada umumnya, yaitu dengan meng-XOR kan setiap bit dari plainteks dengan kunci untuk menjadi cipherteks. Namun, perbedaannya dari *stream cipher* yang lain terletak pada cara membangun kuncinya. *Rabbit cipher* membangun kunci berdasarkan kumpulan dari fungsi-fungsi non-linear. Secara singkat, Algoritma *Rabbit* .menggunakan kunci rahasia sepanjang 128-bit sebagai input dan membangun blok output *pseudo-random* sepanjang 128-bit dari kombinasi bit-bit *internal state* untuk setiap iterasi. Bit-bit *pseudo-random* inilah nanti yang akan menjadi kunci untuk enkripsi plainteks dan dekripsi cipherteks.

Pembahasan pada makalah ini dilakukan secara rinci dari sudut pandang desain algoritmanya, performansi dari algoritma, dan kekuatan algoritma *Rabbit* terhadap bermacam-macam serangan. Setelah pengembangan awalnya, *Rabbit Cipher* telah mengalami beberapa perkembangan dalam desain untuk meningkatkan kekuatannya. Versi yang dibahas di makalah ini adalah versi pengembangan terakhir yang memiliki performansi yang terbaik dan memiliki kekuatan yang hebat terhadap serangan kriptanalis. Dalam bagian performansi akan dibahas dan dibandingkan performansi dari cipher ini pada arsitektur processor Pentium III, Pentium 4, PowerPC, dan MIPS 4Kc. Jenis serangan yang sudah pernah dilakukan untuk membuktikan kekuatan cipher ini adalah *partial key guessing*, *guess-and-determine attack*, *distinguishing attack*, *divide-and-conquer attacks*, dan 'mod n' kriptanalisis.

Karakter dari Algoritma *Rabbit* yang mengesankan adalah performansinya yang tinggi yang ketika dihitung, kecepatan *software* untuk enkripsi/dekripsi yaitu sebesar 3,7 *clock cycles per byte* dengan processor Pentium III. Sampai saat ini, belum ditemukan cara kriptanalisis yang lebih baik dari *exhaustive key search* untuk menyerang *cipher* ini.

***Kata kunci**: Stream cipher, fast, non-linear, chaos, coupled, counter*, enkripsi, dekripsi.

1. Pendahuluan

Di dunia komputer saat ini, kriptografi sangat diperlukan baik untuk menyimpan atau mengirimkan data. Algoritma kriptografi untuk mengenkripsi data dapat menjamin keamanan dan keutuhan dalam proses pengiriman dan penyimpanan data sehingga data tetap rahasia dan hanya dapat diakses oleh orang tertentu. Proses dalam kriptografi dilakukan dengan cara mengenkripsi data mengggunakan kunci tertentu menjadi chiperteks sehingga isinya tidak dapat diakses. Untuk mengakses data yang asli, harus dilakukan proses dekripsi terhadap cipherteks dengan kunci yang sama ketika proses enkripsi. Setelah proses dekripsi selesai, maka data yang asli akan dapat diakses dengan utuh.

Jenis *cipher* yang umum dipakai dalam proses enkripsi/dekripsi data di komputer biasanya adalah jenis *cipher* blok(*block cipher*) dan *cipher* aliran(*stream cipher*). *Cipher* blok adalah jenis *cipher* yang melakukan proses enkripsi/dekripsi dengan membagi bit-bit data menjadi beberapa blok bit yang panjangnya sudah ditentukan, sedangkan *cipher* aliran melakukan proses enkripsi/dekripsi terhadap masing-masing bit tunggal dari bit-bit data. Sebagian besar *cipher* aliran yang ada sekarang mempunyai kelemahan baik di bagian dimana kuncinya mudah dipecahkan maupun di performansinya yang buruk terhadap komputer. Walaupun begitu, seiring dengan aktivitas komputer yang semakin

kompleks dan semakin cepat, maka *cipher* aliran tetap diperlukan untuk menjaga keamanan dan kecepatan aktivitas.

1.1 Sejarah *Rabbit Cipher*

Pada tahun 2002, beberapa orang di perusahaan Cryptico A/S bergabung membentuk sebuah tim untuk memecahkan masalah kunci dan masalah performansi dari *cipher* aliran. Tim tersebut membentuk suatu *cipher* aliran baru yang susah untuk dipecahkan kuncinya dan tetap mempertahankan performansi komputer yang tinggi. *Cipher* ini diberi nama kode *Rabbit Cipher*.

Desain dari *Rabbit* terinspirasi oleh tingkah laku kompleks dari *real-valued chaotic maps*. *Chaotic maps* mempunyai karakter utama yaitu sensitivitasnya yang bersifat eksponensial terhadap perubahan kecil sehingga perulangan dari *maps* seperti ini menghasilkan sesuatu yang terlihat acak dan tidak dapat dapat diprediksi untuk jangka waktu yang lama. Akan tetapi, *chaotic maps* ternyata tidak terlalu aman dalam bentuk diskrit. Banyak *cipher* yang dibuat dengan dasar *chaotic maps* mengalami masalah dalam menciptakan aliran kunci karena penanganan yang berbeda-beda terhadap angka *real*(pecahan) di berbagai jenis prosesor komputer.

Tujuan dari desain Rabbit adalah untuk mengambil keuntungan dari sifat acak *real-valued chaotic maps* dan tetap menciptakan kriptografi yang aman dan optimal dalam bentuk diskrit. Secara khusus, awal dari tahap desain dimulai dengan membuat *chaotic system* dari *non-linear maps* yang berpasangan.

1.2 *Rabbit Cipher* Secara Umum

Secara umum, algoritma *Rabbit* dapat dijelaskan dengan mudah. *Rabbit* menerima input berupa kunci rahasia dengan panjang 128-bit dan sebuah kode IV 64-bit jika diperlukan. Pada setiap iterasinya, kunci dan kode IV tersebut di kombinasikan dengan bit-bit *internal state* sehingga pada akhir iterasi dihasilkan bit-bit *pseudo-random* sepanjang 128-bit. Proses enkripsi/dekripsi dilakukan dengan meng-XOR-kan data *pseudo-random* sepanjang 128-bit tersebut dengan plainteks/cipherteks.

Ukuran dari *internal state* adalah 513 bit yang dibagi menjadi delapan variabel *state* berukuran 32-bit, delapan *counter* berukuran 32-bit dan 1-bit *counter carry*. Delapan variabel *state* tersebut di-*update* oleh delapan pasang fungsi *non-linear* dengan nilai *integer*.

1.3 Notasi

Notasi yang digunakan dalam makalah ini adalah: $\oplus$ untuk menyatakan fungsi XOR, & untuk menyatakan fungsi AND, << dan >> untuk menyatakan operasi pergeseran bit-*wise* ke kiri dan ke kanan, <<< dan >>> menyatakan operasi rotasi bit-*wise* ke kiri dan ke kanan, serta <> menyatakan konkatenasi dari dua kumpulan bit.

$A^{[g..h]}$ berarti bit urutan ke g sampai h dari variabel A. Dalam memberi nomor urut bit dari variabel, bit yang *least significant* diberi nomor urut 0. Nomor heksadesimal diberi awalan "0x". Untuk seluruh varabel dan konstanta digunakan notasi berbentuk *integer*.

2. Desain Dari *Rabbit Cipher*

Internal state dari *Rabbit Cipher* terdiri dari 513 bit. 512 bit pertama dibagi menjadi delapan variabel *state* $x_{j,i}$ berukuran 32-bit dan delapan *counter* $c_{j,i}$ berukuran 32-bit dimana $x_{j,i}$ adalah variabel *state* dari subsistem j pada iterasi i. dan $c_{j,i}$ adalah variabel *counter* yang berkorespondensi langsung dengan $x_{j,i}$. Lalu ada 1-bit *counter carry*, $\phi_{7,i}$ yang harus disimpan diantara iterasi. *Counter carry* ini diinisialisasi dengan nilai 0. delapan variabel *state* dan delapan *counter* ini diturunkan dari kunci yang dimasukkan ketika inisialisasi.

2.1 Skema Pembentukan Kunci

Algoritma *Rabbit* diinisialisasi dengan menurunkan kunci berukuran 128-bit menjadi delapan variabel *state* dan delapan *counter* sehingga ada korespondensi satu-satu antara kunci dengan variabel *state* awal, $x_{j,0}$, dan *counter* awal, $c_{j,0}$. Kunci ini, $K^{[127..0]}$, dibagi menjadi delapan sub-kunci: $k_0 = K^{[15..0]}$, $k_1 = K^{[31.16]}$, $k_2 = K^{[47..32]}$, $k_3 = K^{[63..48]}$, $k_4 = K^{[79..64]}$, $k_5 = K^{[95..80]}$, $k_6 = K^{[111..96]}$, $k_7 = K^{[127..112]}$. Variabel *state* dan *counter* diturunkan dari sub-kunci dengan cara sebagai berikut:

$$x_{j,0} = \begin{cases} k_{(j+1 \bmod 8)} <> k_j & \text{untuk } j \text{ genap} \\ k_{(j+5 \bmod 8)} <> k_{(j+4 \bmod 8)} & \text{untuk } j \text{ ganjil} \end{cases}$$

dan

$$c_{j,0} = \begin{cases} k_{(j+4 \bmod 8)} <> k_{(j+5 \bmod 8)} & \text{untuk } j \text{ genap} \\ k_j <> k_{(j+1 \bmod 8)} & \text{untuk } j \text{ ganjil} \end{cases}$$

Algoritma diiterasi empat kali untuk menghilangkan hubungan bit-bit pada kunci dengan bit-bit pada variabel *internal state*. Terakhir, variabel counter dimodifikasi dengan persamaan:

$$c_{j,4} = c_{j,4} \oplus x_{(j+4 \bmod 8),4}$$

Persamaan ini dibuat dengan tujuan untuk mencegah kriptanalisis terhadap kunci dengan cara membalikkan sistem *counter*.

2.1 Skema Pembentukan IV

IV diperlukan ketika kunci yang sama digunakan untuk mengenkripsi sejumlah data yang berbeda. Setiap penggunaan IV akan menghasilkan aliran kunci enkripsi/dekripsi yang unik, walaupun kunci inputnya sama.

Anggap *internal state* setelah skema pembentukan kunci adalah *master state*, dan ada kopi dari *master state* ini yang akan dimodifikasi dengan skema pembentukan IV. Skema pembentukan IV bekerja dengan cara memodifikasi *counter state* sebagai fungsi dari IV. Cara kerjanya adalah dengan meng-XOR kan IV 64-bit dengan seluruh *counter state* yang berukuran 256-bit. IV 64-bit dinotasikan dengan $IV^{[63..0]}$. Persamaan untuk modifikasinya adalah:

$$\begin{aligned} c_{0,4} &= c_{0,4} \oplus IV^{[31..0]} \\ c_{1,4} &= c_{1,4} \oplus (IV^{[63..48]} <> IV^{[31..16]}) \\ c_{2,4} &= c_{2,4} \oplus IV^{[63..32]} \\ c_{3,4} &= c_{3,4} \oplus (IV^{[47..32]} <> IV^{[15..0]}) \\ c_{4,4} &= c_{4,4} \oplus IV^{[31..0]} \\ c_{5,4} &= c_{5,4} \oplus (IV^{[63..48]} <> IV^{[31..16]}) \\ c_{6,4} &= c_{6,4} \oplus IV^{[63..32]} \\ c_{7,4} &= c_{7,4} \oplus (IV^{[47..32]} <> IV^{[15..0]}) \end{aligned}$$

sistem akan di-iterasi empat kali untuk memastikan seluruh bit *state* tergantung secara *non-linear* dengan bit-bit IV. Modifikasi counter dengan IV menjamin bahwa 2^{64} IV akan menghasilkan aliran kunci yang unik.

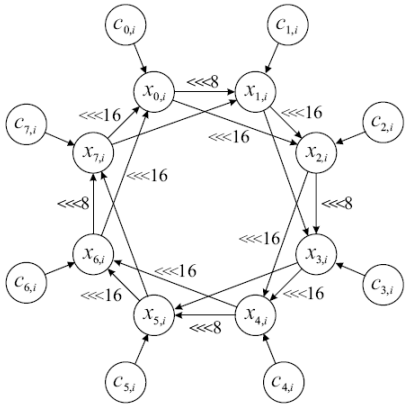
2.2 Fungsi *Next-state*

Inti dari algoritma Rabbit adalah pada iterasi dari sistem yang didefinisikan pada persamaan berikut:

$$x_{j,i+1} = \begin{cases} g_{j,i} + (g_{j\text{-}1 \bmod 8,i} <<<16) + (g_{j\text{-}2 \bmod 8,i} <<<16) \\ \text{(untuk } j \text{ genap)} \\ g_{j,i} + (g_{j\text{-}1 \bmod 8,i} <<< 6) + g_{j\text{-}2 \bmod 8,i} \\ \text{(untuk } j \text{ ganjil)} \end{cases}$$

$$g_{j,i} = ((x_{j,i+} c_{j,i})^2 \oplus ((x_{j,i+} c_{j,i})^2 >> 32)) \bmod 2^{32}$$

Skema dari sistem ini diilustrasikan dalam Gambar 1.



Gambar 1 Ilustrasi gambar dari fungsi *Next-state*

2.3 Sistem *Counter*

Sebelum setiap iterasi, dilakukan operasi increment pada *counter* dengan persamaan sebagai berikut:

$$c_{0,i+1} = \begin{cases} c_{0,i} + a_0 + \phi_{7,i} \bmod 2^{32} \\ (\text{untuk } j = 0) \\ c_{j,i} + a_j + \phi_{j\text{-}1,i+1} \bmod 2^{32} \\ (\text{untuk } j > 0) \end{cases}$$

Dan carry $\phi_{j,i+1}$ didapatkan dari:

$$\phi_{j,i+1} = \begin{cases} 1 \text{ jika } c_{0,i} + a_0 + \phi_{7,i} \geq 2^{32} \wedge j = 0 \\ 1 \text{ jika } c_{j,i} + a_j + \phi_{j\text{-}1,i+1} \geq 2^{32} \wedge j > 0 \\ 0 \text{ untuk lainnya} \end{cases}$$

Konstanta a_j didefinisikan sebagai berikut:

$$\begin{aligned} a_0 = a_3 = a_6 &= \text{0x4D34D34D} \\ a_1 = a_4 = a_7 &= \text{0xD34D34D3} \\ a_2 = a_5 &= \text{0x34D34D34} \end{aligned}$$

2.4 Skema Ekstraksi

Pada akhir setiap iterasi dihasilkan output kode *pseudo-random* berukuran 128-bit dengan cara:

$$
\begin{aligned}
s_i^{[15..0]} &= x_{0,i}^{[15..0]} \oplus x_{5,i}^{[31..16]} \\
s_i^{[31..16]} &= x_{0,i}^{[31..16]} \oplus x_{3,i}^{[15..0]} \\
s_i^{[47..32]} &= x_{2,i}^{[15..0]} \oplus x_{7,i}^{[31..16]} \\
s_i^{[63..48]} &= x_{2,i}^{[31..16]} \oplus x_{5,i}^{[15..0]} \\
s_i^{[79..64]} &= x_{4,i}^{[15..0]} \oplus x_{1,i}^{[31..16]} \\
s_i^{[95..80]} &= x_{4,i}^{[31..16]} \oplus x_{7,i}^{[15..0]} \\
s_i^{[111..96]} &= x_{6,i}^{[15..0]} \oplus x_{3,i}^{[31..16]} \\
s_i^{[127..112]} &= x_{6,i}^{[31..16]} \oplus x_{1,i}^{[15..0]}
\end{aligned}
$$

s_i adalah aliran kunci 128-bit pada iterasi ke-i.

2.5 Skema Enkripsi/Dekripsi

Untuk enkripsi/dekripsi, bit-bit yang sudah diekstraksi di-XOR kan dengan plainteks/cipherteks.

$$c_i = p_i \oplus s_i$$
$$c_i = p_i \oplus s_i$$

c_i dan p_i menyatakan chiperteks dan plainteks blok ke-i.

2.6 Algoritma *Rabbit* Dalam C

Berikut ini adalah algoritma *Rabbit Cipher* dalam bahasa C yang akan digunakan sebagai *Rabbit* API untuk pemrograman lain. Ada dua file utama, yaitu *rabbit*.h dan *rabbit*.c

***rabbit*.h**

```
/*********************************************************************/
/* File name: rabbit.h                                               */
/*-------------------------------------------------------------------*/
/* Header file for reference C version of the Rabbit stream cipher.  */
/*                                                                   */
/*                                                                   */
/* This source code is for little-endian processors (e.g. x86).      */
/*-------------------------------------------------------------------*/
/* Copyright (C) Cryptico A/S. All rights reserved.                  */
/*                                                                   */
/* YOU SHOULD CAREFULLY READ THIS LEGAL NOTICE BEFORE USING THIS SOFTWARE.  */
/*                                                                   */
/* This software is developed by Cryptico A/S and/or its suppliers.  */
/* All title and intellectual property rights in and to the software,  */
/* including but not limited to patent rights and copyrights, are owned by  */
/* Cryptico A/S and/or its suppliers.                                */
/*                                                                   */
/* The software may be used solely for non-commercial purposes       */
/* without the prior written consent of Cryptico A/S. For further    */
/* information on licensing terms and conditions please contact Cryptico A/S  */
/* at info@cryptico.com                                              */
/*                                                                   */
/* Cryptico, CryptiCore, the Cryptico logo and "Re-thinking encryption" are  */
/* either trademarks or registered trademarks of Cryptico A/S.       */
/*                                                                   */
/* Cryptico A/S shall not in any way be liable for any use of this software.  */
/* The software is provided "as is" without any express or implied warranty.  */
/*                                                                   */
/*********************************************************************/


#ifndef _RABBIT_H
#define _RABBIT_H

#include <stddef.h>

/* Type declarations of 32-bit and 8-bit unsigned integers. */
/* Note that some compilers may have differently sized integers. */
/* In this case the following type declarations have to be modified. */
typedef unsigned char cc_byte;
typedef unsigned int cc_uint32;
```

```
/* Structure to store the instance data (internal state) */
typedef struct
{
   cc_uint32 x[8];
   cc_uint32 c[8];
   cc_uint32 carry;
} rabbit_instance;


#ifdef __cplusplus
extern "C" {
#endif

/* All function calls return zero on success */
int rabbit_key_setup(rabbit_instance *p_instance, const cc_byte *p_key,
          size_t key_size);

int rabbit_iv_setup(const rabbit_instance *p_master_instance,
          rabbit_instance *p_instance, const cc_byte *p_iv, size_t iv_size);

int rabbit_cipher(rabbit_instance *p_instance, const cc_byte *p_src,
          cc_byte *p_dest, size_t data_size);

int rabbit_prng(rabbit_instance *p_instance, cc_byte *p_dest, size_t data_size);

#ifdef __cplusplus
}
#endif

#endif
```

rabbit.c

```
/*************************************************************************/
/* File name: rabbit.c                                                   */
/*-----------------------------------------------------------------------*/
/* Source file for reference C version of the Rabbit stream cipher.      */
/*                                                                       */
/*                                                                       */
/* This source code is for little-endian processors (e.g. x86).         */
/*-----------------------------------------------------------------------*/
/* Copyright (C) Cryptico A/S. All rights reserved.                      */
/*                                                                       */
/* YOU SHOULD CAREFULLY READ THIS LEGAL NOTICE BEFORE USING THIS SOFTWARE.    */
/*                                                                       */
/* This software is developed by Cryptico A/S and/or its suppliers.      */
/* All title and intellectual property rights in and to the software,    */
/* including but not limited to patent rights and copyrights, are owned by    */
/* Cryptico A/S and/or its suppliers.                                    */
/*                                                                       */
/* The software may be used solely for non-commercial purposes           */
/* without the prior written consent of Cryptico A/S. For further        */
/* information on licensing terms and conditions please contact Cryptico A/S  */
/* at info@cryptico.com                                                  */
/*                                                                       */
/* Cryptico, CryptiCore, the Cryptico logo and "Re-thinking encryption" are   */
/* either trademarks or registered trademarks of Cryptico A/S.           */
/*                                                                       */
/* Cryptico A/S shall not in any way be liable for any use of this software.  */
/* The software is provided "as is" without any express or implied warranty.  */
```

```
/*                                                                            */
/****************************************************************************/


#include "rabbit.h"


/* Left rotation of a 32-bit unsigned integer */
static cc_uint32 rabbit_rotl(cc_uint32 x, int rot)
{
   return (x<<rot) | (x>>(32-rot));
}


/* Square a 32-bit unsigned integer to obtain the 64-bit result and return */
/* the upper 32 bits XOR the lower 32 bits */
static cc_uint32 rabbit_g_func(cc_uint32 x)
{
   /* Temporary variables */
   cc_uint32 a, b, h, l;

   /* Construct high and low argument for squaring */
   a = x&0xFFFF;
   b = x>>16;

   /* Calculate high and low result of squaring */
   h = ((((a*a)>>17) + (a*b))>>15) + b*b;
   l = x*x;

   /* Return high XOR low */
   return h^l;
}


/* Calculate the next internal state */
static void rabbit_next_state(rabbit_instance *p_instance)
{
   /* Temporary variables */
   cc_uint32 g[8], c_old[8], i;

   /* Save old counter values */
   for (i=0; i<8; i++)
      c_old[i] = p_instance->c[i];

   /* Calculate new counter values */
   p_instance->c[0] += 0x4D34D34D + p_instance->carry;
   p_instance->c[1] += 0xD34D34D3 + (p_instance->c[0] < c_old[0]);
   p_instance->c[2] += 0x34D34D34 + (p_instance->c[1] < c_old[1]);
   p_instance->c[3] += 0x4D34D34D + (p_instance->c[2] < c_old[2]);
   p_instance->c[4] += 0xD34D34D3 + (p_instance->c[3] < c_old[3]);
   p_instance->c[5] += 0x34D34D34 + (p_instance->c[4] < c_old[4]);
   p_instance->c[6] += 0x4D34D34D + (p_instance->c[5] < c_old[5]);
   p_instance->c[7] += 0xD34D34D3 + (p_instance->c[6] < c_old[6]);
   p_instance->carry = (p_instance->c[7] < c_old[7]);

   /* Calculate the g-functions */
   for (i=0;i<8;i++)
      g[i] = rabbit_g_func(p_instance->x[i] + p_instance->c[i]);

   /* Calculate new state values */
   p_instance->x[0] = g[0] + rabbit_rotl(g[7],16) + rabbit_rotl(g[6], 16);
```

```
   p_instance->x[1] = g[1] + rabbit_rotl(g[0], 8) + g[7];
   p_instance->x[2] = g[2] + rabbit_rotl(g[1],16) + rabbit_rotl(g[0], 16);
   p_instance->x[3] = g[3] + rabbit_rotl(g[2], 8) + g[1];
   p_instance->x[4] = g[4] + rabbit_rotl(g[3],16) + rabbit_rotl(g[2], 16);
   p_instance->x[5] = g[5] + rabbit_rotl(g[4], 8) + g[3];
   p_instance->x[6] = g[6] + rabbit_rotl(g[5],16) + rabbit_rotl(g[4], 16);
   p_instance->x[7] = g[7] + rabbit_rotl(g[6], 8) + g[5];
}


/* Initialize the cipher instance (*p_instance) as a function of the */
/* key (*p_key) */
int rabbit_key_setup(rabbit_instance *p_instance, const cc_byte *p_key,
          size_t key_size)
{
   /* Temporary variables */
   cc_uint32 k0, k1, k2, k3, i;

   /* Return error if the key size is not 16 bytes */
   if (key_size != 16)
      return -1;


   /* Generate four subkeys */
   k0 = *(cc_uint32*)(p_key+ 0);
   k1 = *(cc_uint32*)(p_key+ 4);
   k2 = *(cc_uint32*)(p_key+ 8);
   k3 = *(cc_uint32*)(p_key+12);


   /* Generate initial state variables */
   p_instance->x[0] = k0;
   p_instance->x[2] = k1;
   p_instance->x[4] = k2;
   p_instance->x[6] = k3;
   p_instance->x[1] = (k3<<16) | (k2>>16);
   p_instance->x[3] = (k0<<16) | (k3>>16);
   p_instance->x[5] = (k1<<16) | (k0>>16);
   p_instance->x[7] = (k2<<16) | (k1>>16);

   /* Generate initial counter values */
   p_instance->c[0] = rabbit_rotl(k2, 16);
   p_instance->c[2] = rabbit_rotl(k3, 16);
   p_instance->c[4] = rabbit_rotl(k0, 16);
   p_instance->c[6] = rabbit_rotl(k1, 16);
   p_instance->c[1] = (k0&0xFFFF0000) | (k1&0xFFFF);
   p_instance->c[3] = (k1&0xFFFF0000) | (k2&0xFFFF);
   p_instance->c[5] = (k2&0xFFFF0000) | (k3&0xFFFF);
   p_instance->c[7] = (k3&0xFFFF0000) | (k0&0xFFFF);

   /* Clear carry bit */
   p_instance->carry = 0;

   /* Iterate the system four times */
   for (i=0; i<4; i++)
      rabbit_next_state(p_instance);

   /* Modify the counters */
   for (i=0; i<8; i++)
      p_instance->c[i] ^= p_instance->x[(i+4)&0x7];
```

```
   /* Return success */
   return 0;
}


/* Initialize the cipher instance (*p_instance) as a function of the */
/* IV (*p_iv) and the master instance (*p_master_instance) */
int rabbit_iv_setup(const rabbit_instance *p_master_instance,
          rabbit_instance *p_instance, const cc_byte *p_iv, size_t iv_size)
{
   /* Temporary variables */
   cc_uint32 i0, i1, i2, i3, i;

   /* Return error if the IV size is not 8 bytes */
   if (iv_size != 8)
      return -1;

   /* Generate four subvectors */
   i0 = *(cc_uint32*)(p_iv+0);
   i2 = *(cc_uint32*)(p_iv+4);
   i1 = (i0>>16) | (i2&0xFFFF0000);
   i3 = (i2<<16) | (i0&0x0000FFFF);

   /* Modify counter values */
   p_instance->c[0] = p_master_instance->c[0] ^ i0;
   p_instance->c[1] = p_master_instance->c[1] ^ i1;
   p_instance->c[2] = p_master_instance->c[2] ^ i2;
   p_instance->c[3] = p_master_instance->c[3] ^ i3;
   p_instance->c[4] = p_master_instance->c[4] ^ i0;
   p_instance->c[5] = p_master_instance->c[5] ^ i1;
   p_instance->c[6] = p_master_instance->c[6] ^ i2;
   p_instance->c[7] = p_master_instance->c[7] ^ i3;

   /* Copy internal state values */
   for (i=0; i<8; i++)
      p_instance->x[i] = p_master_instance->x[i];
   p_instance->carry = p_master_instance->carry;

   /* Iterate the system four times */
   for (i=0; i<4; i++)
      rabbit_next_state(p_instance);

   /* Return success */
   return 0;
}

/* Encrypt or decrypt data */
int rabbit_cipher(rabbit_instance *p_instance, const cc_byte *p_src,
          cc_byte *p_dest, size_t data_size)
{
   /* Temporary variables */
   cc_uint32 i;

   /* Return error if the size of the data to encrypt is */
   /* not a multiple of 16 */
   if (data_size%16)
      return -1;

   for (i=0; i<data_size; i+=16)
   {
      /* Iterate the system */
```

```
      rabbit_next_state(p_instance);

      /* Encrypt 16 bytes of data */
      *(cc_uint32*)(p_dest+ 0) = *(cc_uint32*)(p_src+ 0) ^
                p_instance->x[0]   ^   (p_instance->x[5]>>16)   ^   (p_instance-
>x[3]<<16);
      *(cc_uint32*)(p_dest+ 4) = *(cc_uint32*)(p_src+ 4) ^
                p_instance->x[2]   ^   (p_instance->x[7]>>16)   ^   (p_instance-
>x[5]<<16);
      *(cc_uint32*)(p_dest+ 8) = *(cc_uint32*)(p_src+ 8) ^
                p_instance->x[4]   ^   (p_instance->x[1]>>16)   ^   (p_instance-
>x[7]<<16);
      *(cc_uint32*)(p_dest+12) = *(cc_uint32*)(p_src+12) ^
                p_instance->x[6]   ^   (p_instance->x[3]>>16)   ^   (p_instance-
>x[1]<<16);

      /* Increment pointers to source and destination data */
      p_src += 16;
      p_dest += 16;
   }

   /* Return success */
   return 0;
}


/* Generate data with Pseudo-Random Number Generator */
int rabbit_prng(rabbit_instance *p_instance, cc_byte *p_dest,
          size_t data_size)
{
   /* Temporary variables */
   cc_uint32 i;

   /* Return error if the size of the data to generate is */
   /* not a multiple of 16 */
   if (data_size%16)
      return -1;

   for (i=0; i<data_size; i+=16)
   {
      /* Iterate the system */
      rabbit_next_state(p_instance);

      /* Generate 16 bytes of pseudo-random data */
      *(cc_uint32*)(p_dest+ 0) = p_instance->x[0] ^
                (p_instance->x[5]>>16) ^ (p_instance->x[3]<<16);
      *(cc_uint32*)(p_dest+ 4) = p_instance->x[2] ^
                (p_instance->x[7]>>16) ^ (p_instance->x[5]<<16);
      *(cc_uint32*)(p_dest+ 8) = p_instance->x[4] ^
                (p_instance->x[1]>>16) ^ (p_instance->x[7]<<16);
      *(cc_uint32*)(p_dest+12) = p_instance->x[6] ^
                (p_instance->x[3]>>16) ^ (p_instance->x[1]<<16);

      /* Increment pointer to destination data */
      p_dest += 16;
   }

   /* Return success */
   return 0;
}
```

3. Performansi

3.1 Kondisi pengujian

3.1.1 Asumsi

Selama pengujian, semua blok data disusun per 16-*byte*. Pada kenyataannya, tidak semua blok data di seluruh komputer akan tersusun per 16-*byte*. Walaupun disusun dengan ukuran lain, program akan tetap menghasilkan ciphertext yang valid, namun performansinya akan memburuk di beberapa platform prosesor.

3.1.2 Mengukur Performansi

Seluruh fungsi *Rabbit* diimplementasikan dengan standar bahasa C. Pengukuran performansi dilakukan dengan cara membaca *clock counter* dari prosesor sebelum dan sesudah pemanggilan prosedur yang akan diukur performansinya. Hasilnya dihitung dengan cara melihat selisih dari kedua nilai *clock* yang dihasilkan dikurangi dengan *overhead* prosesor untuk memanggil fungsi *timing*.

3.1.3 Mengukur Pemakaian Memori

Seluruh *internal state* berukuran 513 bit disimpan dalam sebuah struktur data instan yang menempati 68 *byte* dalam *platform* prosesor 32-bit. Kebutuhan memori yang nanti akan disampaikan menunjukan besar kapasitas memori yang dialokasikan di *stack* untuk pemanggilan fungsi atau prosedur. Kebutuhan memori untuk penyimpanan data sementara tidak diikutsertakan.

3.2 *Intel Pentium*

Pengukuran performansi dilakukan dengan menggunakan *Rabbit* versi baru yang deprogram menggunakan gabungan bahasa *assembly* dan bahasa C dan menggunakan instruksi MMX. Untuk mengkompilasi *Rabbit* digunakan Intel C++ 7.0 *compiler*.

3.2.1 Mengukur Performansi *Intel Pentium*

Performansi diukur dengan membaca hitungan *clock* yang sudah dilewati dengan memanggil primitif RDTSC. Hitungan *clock* dibaca persis sebelum dan sesudah pemanggilan fungsi enkripsi atau fungsi pembentukan kunci. Contoh *pseudo-code* nya adalah seperti ini:

```
start = read_clock_tick();
end = read_clock_tick();
overhead = end - start;
start = read_clock_tick();
key_setup(...);
end = read_clock_tick();
key_setup_time = end - start -
overhead;
start = read_clock_tick();
cipher(...);
end = read_clock_tick();
cipher_time = end - start -
overhead;
cipher_time_pr_byte =
cipher_time / data_size;
```

Overhead untuk pemanggilan fungsi masuk ke dalam perhitungan performansi, akan tetapi *overhead* dari *timing method* dikecualikan. Implementasi yang diuji menggunakan sebuah file DLL yang akan sedikit meningkatkan *overhead* dalam pemanggilan fungsi, namun *overhead* ini tidak akan mengganggu performansi enkripsi ketika program mengenkripsi data dalam jumlah yang sangat besar. Performansi diukur dengan memangil fungsi yang dikehendaki sebanyak 25 kali dan mengambil hasil yang terbaik. Kecepatan enkripsi terbaik didapatkan ketika program mengenkripsi data sebesar 16 kilo*byte* dalam memori.

3.2.2 Performansi *Pentium* III

Performansi diukur dengan *platform desktop* PC, 1.0 GHz Pentium III Processor(Intel 82815 chipset), dan berjalan diatas Windows 2000. Pengukuran juga dilakukan di sebuah laptop, 850 MHz Pentium III Processor(ALI M1621/M1533 chipset), dan berjalan diatas Windows Me. Hasil pengukuran keduanya mirip yang akan ditunjukkan oleh Tabel 1.

Tabel 1 Panjang kode, kebutuhan memori, dan performansi Pentium III

Fungsi	Panjang Kode	Memori	Performansi
Pembentukan Kunci	794 *byte*	32 *byte*	307 *cycle*
Pembentukan IV	727 *byte*	36 *byte*	293 *cycle*
Enkripsi/ Dekripsi	717 *byte*	36 *byte*	3,7 *cycle*/*byte*

3.2.3 Performansi *Pentium* 4

Performansi diukur dengan *platform desktop* PC, 1.7 GHz Pentium 4 Processor(Intel 82850 chipset), dan berjalan diatas Windows 2000. Hasil pengukurannya ditunjukkan oleh Tabel 2.

Tabel 2 Panjang kode, kebutuhan memori, dan performansi Pentium 4

Fungsi	Panjang Kode	Memori	Performansi
Pembentukan Kunci	698 *byte*	16 *byte*	468 *cycle*
Pembentukan IV	688 *byte*	20 *byte*	428 *cycle*
Enkripsi/ Dekripsi	762 *byte*	28 *byte*	5,1 *cycle*/*byte*

3.3 *Power*PC

Pengukuran performansi pada arsitektur PowerPC menggunakan program *Rabbit* versi terbaru yang dibangun dengan bahasa *assembly*. Pengujian dilakukan diatas *evaluation board* dengan *clock speed* 25 MHz dan prosesor PPC440GX dengan kecepatan 533 MHz. Hasil pengujian ditampilkan di Tabel 3.

Tabel 3 Panjang kode, kebutuhan memori, dan performansi PowerPC

Fungsi	Panjang Kode	Memori	Performansi
Pembentukan Kunci	512 *byte*	72 *byte*	405 *cycle*
Pembentukan IV	444 *byte*	72 *byte*	298 *cycle*
Enkripsi/ Dekripsi	440 *byte*	72 *byte*	3,8 *cycle*/*byte*

3.4 MIPS 4Kc

Pengukuran performansi pada arsitektur MIPS 4Kc menggunakan program *Rabbit* versi terbaru yang dibangun dengan bahasa *assembly*. Pengembangan program dilakukan dengan menggunakan *The Embedded Linux Development Kit*(ELDK). Penulisan kode program pun dibuat untuk organisasi *little-endian* maupun *big-endian*.

3.4.1 Mengukur Performansi MIPS 4Kc

Performansi diukur dengan menggunakan fungsi `clock()` sebelum dan sesudah pemanggilan fungsi *Rabbit*. Pngukuran dilakukan dengan mengenkripsikan 4096 *byte* data. Sama seperti pada pengujian Pentium, *overhead* untuk pemanggilan fungsi masuk ke dalam perhitungan performansi, akan tetapi *overhead* dari *timing method* dikecualikan.

3.2.2 Performansi MIPS 4Kc

Performansi diukur dengan prosesor yang berjalan dengan kecepatan 150 MHz dengan sistem operasi Linux. Hasil pengukurannya ditunjukkan oleh Tabel 4 untuk *little-endian* dan Tabel 5 untuk *big-endian*.

Tabel 4 Panjang kode, kebutuhan memori, dan performansi MIPS 4Kc (*little-endian*)

Fungsi	Panjang Kode	Memori	Performansi
Pembentukan Kunci	856 *byte*	32 *byte*	749 *cycle*
Pembentukan IV	816 *byte*	32*byte*	749 *cycle*
Enkripsi/ Dekripsi	892 *byte*	40 *byte*	10,9 *cycle*/*byte*

Tabel 5 Panjang kode, kebutuhan memori, dan performansi MIPS 4Kc (*big-endian*)

Fungsi	Panjang Kode	Memori	Performansi
Pembentukan Kunci	960 *byte*	32 *byte*	749 *cycle*
Pembentukan IV	888 *byte*	32 *byte*	749 *cycle*
Enkripsi/ Dekripsi	1052 *byte*	40 *byte*	13,5 *cycle*/*byte*

4. Analisis Keamanan

4.1 Properti Pembentukan Kunci

Pembentukan kunci dapat dibagai menjadi tiga tahap, yaitu ekpansi kunci, iterasi sistem, dan modifikasi *counter*.

4.1.1 Ekspansi Kunci

Ada dua properti yang harus dipastikan pada tahap ini. Yang pertama adalah adanya korespondensi satu-satu antara kunci, *state*, dan *counter*, yang akan mencegah adanya kunci yang redundan. Properti yang lain adalah setiap satu iterasi pada fungsi *next-state*, setiap bit kunci harus mempengaruhi delapan variabel *state*.

4.1.2 Iterasi Sistem

Skema ekspansi kunci memastikan bahwa setelah dua iterasi dari fungsi *next-state*, seluruh bit dari *state* sudah diubah oleh semua bit kunci dengan probabilitas terukur 0,5. Sebuah marjin aman dibuat dengan mengiterasikan sistem sebanyak empat kali.

4.1.3 Modifikasi *Counter*

Walaupun *counter* harus disebarluaskan ke publik, modifikasi *counter* akan membuat proses kriptanalisis untuk menemukan kunci dengan cara membalikkan sistem *counter* menjadi sulit karena proses kriptanalisis membutuhkan informasi tambahan tentang variabel *state*. Dengan modifikasi *counter*, setiap kunci yang berbeda tidak dijamin dapat memberikan *counter* yang unik. Akan tetapi hal itu tidak menjadi masalah dan sudah diselesaikan dengan kode IV.

4.1.4 Serangan pada Fungsi Pembentukan Kunci

Karena adanya empat iterasi setelah ekspansi kunci dan modifikasi *counter*, baik bit-bit *counter* maupun bit-bit *state* bergantung secara penuh dan kuat secara *non-linear* terhadap bit-bit kunci. Hal ini membuat serangan dengan menebak bagian dari kunci menjadi sangat sulit. Walaupun bit-bit *counter* diketahui setelah adanya modifikasi *counter*, usaha untuk mencari kunci tetap akan sulit, namun akan mempermudah serangan dengan tipe-tipe yang lain.

Karena *non-linear* map pada *Rabbit* bersifat banyak ke satu(N..1), kunci input yang berbeda dapat menghasilkan aliran kunci output yang sama. Masalah ini dapat dikurangi dengan pertanyaan apakah kunci input yang berbeda juga dapat menghasilkan nilai *counter* yang sama, karena nilai *counter* yang berbeda akan hampir memastikan aliran kunci yang berbeda juga. Hal ini disebabkan karena ketika bagian yang bersifat periodik pada graf fungsional sudah tercapai, fungsi *next-state*, termasuk sistem *counter*, mempunyai relasi satu-satu dengan titik-titik yang ada pada periode tersebut.

Skema ekspansi kunci didesain sedemikian rupa sehingga setiap kunci akan menghasilkan nilai *counter* yang unik. Akan tetapi, modifikasi *counter* dapat menghasilkan nilai counter yang sama untuk dua kunci yang berbeda. Dengan berasumsi bahwa output setelah empat iterasi awal benar-benar random dan tidak berhubungan dengan sistem *counter*, probabilitas dari terciptanya *counter* yang sama diberikan dengan ”paradoks ulang tahun”. Contoh untuk 2^{128} kunci, satu pasang nilai yang sama diperkirakan terjadi dalam *counter state* 256-bit. Namun diperkirakan nilai *counter* yang sama ini akan menciptakan masalah.

Kemungkinan lain dalam serangan terhadap kunci kriptografi adalah dengan mengeksploitasi kesimetrisan antara *next-state* dengan fungsi pembentukan kunci. Sebagai contoh, anggap dua buah kunci, K_1 dan K_2 dihubungan dengan persama $K_1^{[i]} = K_2^{[i+32]}$ untuk semua i. Ini akan langsung berdampak pada relasi $x_{j,0} = x_{j+2,0}$ dan $c_{j,0} = c_{j+2,0}$. Jika konstanta a_j juga memiliki relasi yang mirip, fungsi *next-state* akan memelihara properti ini. Dengan cara yang sama, kesimetrisan hanya akan mengarahkan kepada kunci yang buruk. Contoh, jika $K_1^{[i]} = K_2^{[i+32]}$ untuk semua i, maka $x_{j,0} = x_{j+2,0}$ dan $c_{j,0} = c_{j+2,0}$. Akan tetapi, fungsi *next-state* tidak memelihara properti ini karena pada sistem *counter* $a_j \neq a_{j+2}$.

4.2 Properti Pembentukan IV

Ekspansi IV ini dipilih dengan tujuan untuk membuat skema rotasi spesifik dari fungsi-g agar menjadi lebih baik. Setiap bit IV akan mempengaruhi empat fungsi-g yang berbeda pada iterasi pertama, yang merupakan jumlah pengaruh maksimal yang mungkin karena ukuran IV yang hanya 64-bit.Skema ini juga menjamin bahwa delapan variabel *state* akan terkena pengaruh IV setelah satu iterasi.

Sistem lalu diiterasi sebanyak empat kali untuk menjamin bit-bit *state* bergantung secara *non-linear* terhadap bit-bit IV. Modifikasi *counter* dengan menggunakan IV dipilih karena seluruh 2^{64} kemungkinan IV yang berbeda kan mengarah kepada aliran kunci yang unik.

4.2.1 Tujuan Penggunaan IV

Tujuan penciptaan skema IV dari sisi keamanan adalah panjang IV yang 64-bit yang dapat mengenkripsi sampai 2^{64} plainteks dengan kunci 128-bit yang sama. Dengan memanggil fungsi pembentukan IV sebanyak sampai 2^{64} kali. Maka cipherteks-cipherteks yang dihasilkan tidak akan bisa disamakan dan fungsi pembentukan kunci akan terlihat mendekati sangat acak.

Ada beberapa alasan secara kualitas kenapa skema pembentukan IV dapat memenuhi tujuan dari sisi keamanan seperti diatas. Fungsi *next-state* mempunyai properti difusi yang baik. Sebagai contoh, hanya setelah dua iterasi setiap bit *state* bergantung pada setiap bit kunci dengan probabilitas terukur 1,5. Lebih lanjut, setiap output *byte*(atau bit) bergantung secara virtual terhadap seluruh input *byte*(atau bit). Juga diketahui bahwa fungsi *next-state* dari *Rabbit Cipher* sangat bersifat *non-linear*. Karena sifat difusi yang baik dan properti *non-linear*, dan karena semua bit IV tergabung dengan seluruh bit *state* setelah empat kali iterasi pembentukan IV, serangan differensial tidak mungkin dilakukan terhadap keamanan *cipher* ini.

4.3 Properti *Counter*

Kedinamisan dari *counter* terdiri dari panjang periode dan probabilitas *bit-flip* dari bit-bit individual.

4.3.1 Panjang Periode

Fitur yang paling penting dari *cipher* aliran yang memiliki counter adalah adanya batas bawah dari panjang periode. Sistem *counter* yang diadopsi *Rabbit Cipher* mempunyai panjang periode 2^{256}-1.

4.3.2 Probabilitas dari *Bit-flip* di *Counter*

Untuk *counter* berukuran 256-bit yang di-*increment* dengan 1, panjang periode untuk bit posisi i adalah 2^{i+1}. Hal ini mengimplikasikan bahwa bit yang *least significant* mempunyai probabilitas *bit-flip* sebesar 1 dan bit *most significant* mempunyai probabilitas *bit-flip* 2^{-255}. Konsekuensinya, bit *most significant* akan tetap konstan pada nilainya selama 255 iterasi, sehingga membuat bit tersebut sangat mudah ditebak.

Di lain pihak, semua bit di *counter* yang sudah didefinisikan pada bagian 2.3 akan mempunyai panjang periode yang sama karena setiap bit secara tidak langsung dipengaruhi oleh bit lainnya yang disebabkan oleh adanya masukan *carry*, $\phi_{7,i}$, ke counter $c_{0,i}$. Hal ini menyiratkan bahwa setiap bit memiliki panjang periode yang sama dengan sistem secara keseluruhan. Setelah dihitung, probabilitas *bit-flip* untuk setiap posisi bit bersifat unik. Karena semua *counter* memiliki periode yang penuh dan probabilitas *bit-flip* yang unik, maka usaha memprediksi pola bit dari variabel *counter* menjadi terlihat susah.

4.4 Analisis Aljabar

4.4.1 Serangan *Guess-and-Verify*

Serangan dengan tipe seperti ini hanya *feasible* dilakukan jika ada bagian dari state yang harus diketahui untuk memprediksi pecahan bit dari output yang signifikan. Seorang kriptanalis akan menebak bagian dari *state*, memprediksikan bit-bit outputnya dan membandingkan dengan bit-bit output yang asli. Strateginya adalah untuk memprediksi secara akurat satu *byte* output yang diekstrasi dengan cara menebak sedikit mungkin *byte* input ada.

Rata-rata, seorang kriptanalis harus menebak antara 2-12 *byte* input untuk fungsi-*g* yang berbeda, sehingga total sebanyak 192 bit yang harus ditebak. Akan tetapi menghitung *byte* dengan bit terekstrak yang lebih sedikit tetap menghasilkan usaha yang lebih berat daripada *exhaustive key search*. Kesimpulannya, serangan seperti ini sangat tidak mungkin dilakukan dengan menebak dalam jumlah bit yang lebih sedikit daripada ukuran kunci.

4.4.2 Serangan *Guess-and-Determine*

Strategi dalam serangan ini adalah dengan menebak beberapa variabel *cipher* yang tidak diketahui dan dari situ dilakukan deduksi terhadap variabel lainnya yang masih tidak diketahui. Untuk kemudahan, diasumsikan bahwa *counter* bersifat statik. Serangan sederhana dengan tipe seperti ini terdiri dari menebak sisa 128 bit dari *internal state* dari 128 bit yang diekstraksi untuk setiap iterasi dari dua iterasi yang berurutan. Angka ini untuk menebak sisa 128+128 bit dan menurunkan nilai *counter*. Masing-masing hasil dari sistem harus diiterasi beberapa kali untuk memverifikasi output.

Akan tetapi pada serangan diatas diasumsikan bahwa tidak ada keuntungan yang bisa diambil jika membagi *counter* dan variabel *state* menjadi blok-blok yang lebih kecil. Sebuah serangan yang mengeksploitasi kemungkinan ini dijabarkan sebagai berikut: Bagi variabel *state* dan *counter* berukuran 32-bit menjadi variabel 8-bit. Buat sistem persamaan yang terdiri dari $8 \cdot 4$ 8-bit subsistem untuk N iterasi bersamaan dengan $(N + 1) \cdot 8$ fungsi ekstraksi yang berkorespondensi yang dibagi menjadi fungsi $(N + 1) \cdot 16$ 8-bit. Untuk mendapatkan sistem persamaan tertutup, dibutuhkan output dari fungsi ekstraksi $4 \cdot 8$. Sebagai contoh untuk $N = 3$, sistem persamaan terdiri dari 160

pasang persamaan dengan $8 \cdot 4$ *byte counter* yang tidak diketahui dan $(3 + 1) \cdot 8 \cdot 4$ *byte state* yang tidak diketahui dengan total 160 tidak diketahui.

Sebuah strategi untuk memecahkan persamaan tersebut harus dicari dengan cara menebak *byte* input sesedikit mungkin dan menentukan sisa *byte* yang tidak diketahui. Efisiensi dari strategi tersebut bergantung pada jumlah variabel yang harus ditebak sebelum proses penentuan dimulai. Jumlah variabel tersebut diberikan oleh subsistem berukuran 8-bit dengan jumlah variabel input paling sedikit. Dengan tidak melihat *counter*, maka setiap *byte* dari fungsi next-state bergantung pada 12 *byte* input. Jika *counter* diikutsertakan, maka setiap *byte* output dari subsistem bergantung pada 24 *byte* input. Kesimpulannya, kriptanalis harus menebak lebih dari 128 bit sebelum proses penentuan dimulai, yang akhirnya membuat serangan seperti ini menjadi tidak *feasible*. Membagi sistem menjadi blok-blok yang lebih kecil daripada *byte* pun hanya akan menghasilkan kesimpulan yang sama.

4.5 Analisis Differensial

Dalam analisis ini digunakan dua skema berbeda yang didefinisikan sebagai berikut: Ambil dua input, $x`$ dan $x``$, dan output yang berkorespondensi $y`$ dan $y``$, lalu perbedaan modulus pengurangan input dan output didefinisikan dengan $\Delta x = x` \oplus x``$ dan $\Delta y = y` \oplus y``$. Karena skema modulus pengurangan menyebabkan *counter* menjadi linear, banyak kasus yang akan mereferensi ke masalah ini. Perbedaan XOR juag menjadi masalah dalam beberapa kasus. Akan tetapi belum ada skema lain yang lebih baik.

4.5.1 Differensial dari fungsi-*g*

Pada prinsipnya, seluruh kemungkin 2^{64} differensial harus dihitung, namun ketika XOR adalah operator pembedanya, investigasi dari fungsi-*g* dengan panjang kata yang lebih kecil telah membuktikan bahwa struktur dari differensial dengan probabilitas terbesar tetap sama untuk panjang kata yang berbeda. Pada penghitungan probabilitas untuk seluruh differensial pada fungsi-*g* berukuran 8-, 10-, 12-, 14-, 16-, dan 18-bit, strukturnya mempunyai karakter satu blok mempunyai ukuran sekitar ¾ panjang kata.

Fungsi-*g* bersifat non-injektif, yang berarti ada perbedaan input selain 0 yang mengarah menuju perbedaan output bernilai 0. Tidak ada struktur yang jelas yang dapat diamati, sehingga differensial dengan probabilitas terbesar tidak dapat ditentukan untuk fungsi-*g* 32-bit. Probabilitas-probabilitas ini berhubungan secara skalar dengan panjang kata. Dengan asumsi skalar bergerak terus menuju 32-bit, differensial dengan probabilitas terbesar diperkirakan sekitar 2^{-17}. Probabilitas ini lebih rendah dibandingkan dengan kasus XOR sebagai operator differensial. Untuk mencapai differensial dengan probabilitas 1, differensial tersebut harus sama dengan panjang kata, yang berarti urutan *non-linear* dari fungsi-*g* dalam kondisi maksimal.

5. Simpulan

Dalam makalah ini, sudah dibahas jenis *cipher* aliran yang masih masuk dalam kategori baru, yaitu *Rabbit Cipher*. Makalah ini sudah menjelaskan dengan cukup lengkap mengenai desain algoritma *Rabbit*, performansi dan implementasi dari *Rabbit*, serta evaluasi dari masalah-masalah keamanan pada *Rabbit Cipher*.

Desain dari *Rabbit Cipher* mengambil keuntungan dari sifat acak *real-valued chaotic maps* dan tetap menciptakan kriptografi yang aman dan optimal dalam bentuk diskrit. *Rabbit Cipher* menggunakan kunci input dengan panjang 128-bit yang dipecah menggunakan skema pembentukan kunci dan skema *counter* menjadi delapan variabel *state* berukuran 32-bit, delapan *counter* berukuran 32-bit dan 1-bit ounter carry. Variabel *state* inilah yang akan membangun kunci *pseudo-random* baru berukuran 128-bit yang akan digunakan untuk meng-XOR kan plainteks/chiperteks dalam proses enkripsi/dekripsi.

Yang paling penting dalam masalah keamanan adalah banyaknya macam serangan yang sudah dicoba dilakukan terhadap *Rabbit Cipher*, namun tidak ada serangan yang lebih baik daripada metode *exhaustive key search*.

Performansi yang sudah diukur untuk proses enkripsi dekripsi dari berbagai *platform* prosesor adalah 3,7 *clock cycle* per *byte* untuk Pentium III, 5,1 *clock cycle* per *byte* untuk Pentium 4, 3,8 *clock cycle* per *byte* untuk Power PC, dan rata-rata 12,2 *clock cycle* per *byte* untuk MIPS 4Kc.

DAFTAR PUSTAKA

[1] Boesgard, Martin, Mette Vesterager, Thomas Pedersen, Jesper Christiansen, and Ove Scavenius. *Rabbit: A New High-Performance Stream Cipher.* Cryptico A/S. 2003. http://www.cryptico.com/. Tanggal akses: 23 September 2006 pukul 15:00.

[2] Boesgaard, Martin, Thomas Pedersen, Mette Vesterager, and Erik Zenner. *The Rabbit Stream Cipher - Design and Security Analysis.* Cryptico A/S. 2004. http://www.cryptico.com/. Tanggal akses: 23 September 2006 pukul 15:00.

[3] Munir, Rinaldi. *Diktat Kuliah IF5054 Kriptografi*. Institut Teknologi Bandung. 2006

[4] Rijmen, Vincent. *Analysis of Rabbit.* Cryptico A/S. 2003. http://www.cryptico.com/. Tanggal akses: 23 September 2006 pukul 15:00.

[5] White Papers(anonym). *"mod n" Cryptanalysis of Rabbit.* Cryptico A/S. 2003. http://www.cryptico.com/. Tanggal akses: 23 September 2006 pukul 15:00.

[6] White Papers(anonym). *Algebraic Analysis of Rabbit.* Cryptico A/S. 2006. http://www.cryptico.com/. Tanggal akses: 23 September 2006 pukul 15:00.

[7] White Papers(anonym). *Analysis of the Key Setup Function in Rabbit*. Cryptico A/S. 2003. http://www.cryptico.com/. Tanggal akses: 23 September 2006 pukul 15:00.

[8] White Papers(anonym). *Periodic Properties of Rabbit.* Cryptico A/S. 2003. http://www.cryptico.com/. Tanggal akses: 23 September 2006 pukul 15:00.

[9] White Papers(anonym). *Rabbit Stream Cipher, Algorithm Specification.* Cryptico A/S. 2005. http://www.cryptico.com/. Tanggal akses: 23 September 2006 pukul 15:00.

[10] White Papers(anonym). *Rabbit Stream Cipher, Performance Evaluation.* Cryptico A/S. 2005. http://www.cryptico.com/. Tanggal akses: 23 September 2006 pukul 15:00.

[11] White Papers(anonym). *Security Analysis of the IV-Setup for Rabbit.* Cryptico A/S. 2003. http://www.cryptico.com/. Tanggal akses: 23 September 2006 pukul 15:00.