Studi Mengenai Desain dan Implementasi *Two Stage Random Number Generator*

Nugroho Herucahyono
13504038

Program Studi Teknik Informatika
Institut Teknologi Bandung
Jl Ganesha 10, Bandung

E-mail : if14038@students.if.itb.ac.id, xinuc@students.itb.ac.id

Abstrak

Makalah ini akan membahas tentang desain dan implementasi dari *Two Stage Random Number Generator*. *Random number generator* adalah faktor yang sangat penting dalam kriptografi. *Random number generator* dapat menentukan keamanan suatu skema kriptografi. Akan tetapi, hampir tidak mungkin untuk mendapatkan angka yang benar-benar acak (*random*). Hal ini merupakan salah satu kelemahan (*vulnerability*) dalam suatu skema kriptografi.

Salah satu metoda yang banyak digunakan untuk meng-*generate* bilangan acak adalah *Pseudo-Random Number Generator*. Metoda ini akan meng-*generate* bilangan acak dari suatu algoritma yang telah ditentukan, sehingga seolah-olah akan dihasilkan bilangan acak. Algoritma yang digunakan pada *Pseudo-Random Number Generator* ini sangat beragam, semua berupaya untuk mendapatkan rangkaian bilangan seacak mungkin.

Namun apapun algoritma yang digunakan, hasil dari *Pseudo-Random Number Generator* ini sangat ditentukan oleh kondisi awalnya. Misalnya, sebuah algoritma meng-*generate* bilangan acak semu dengan menggunakan nilai awal yang ditentukan waktu, maka setiap kali kita meng-*generate* bilangan *random* pada waktu yang berbeda akan menghasilkan hasil yang berbeda pula. Hal ini masih memiliki banyak kelemahan. Misalnya pada algoritma tersebut penyerang mengetahui waktu meng-*generate* bilangan tersebut, maka penyerang akan mampu menebak bilangan acak semu yang kita *generate.*

Untuk memperkecil peluang penyerang mengetahui kondisi awal dari algoritma *pseudo-random* kita, maka nilai awalnya dapat kita gunakan hasil *random* pula. Sehingga kita akan menggunakan output dari suatu algoritma *pseudo-random* sebagai kondisi awal dari algoritma *random number generator* kita. Hal ini dapat memperkecil peluang penyerang menebak bilangan acak kita. Dengan demikian, diharapkan bilangan yang kita hasilkan akan menjadi semakin acak dan susah untuk ditebak.

Kata Kunci : *Two Stage Random Generator, Pseudo Random Generator*, Kriptografi, Lehmer.

Bilangan *Random* dalam kriptografi

Bilangan acak (*random*) merupakan hal yang sangat penting dalam kriptografi. Tanpa bilangan *random*, sebagian besar algoritma kriptografi akan dengan mudah dipecahkan. Dengan menggunakan bilangan acak dalam suatu algoritma kriptografi, akan mempersulit penyerang untuk menebak kunci maupun hasil enkripsi dari suatu kriptografi.

Bilangan *random* digunakan dalam kriptografi untuk berbagai hal. Misalnya untuk meng *generate* kunci, menghasilkan variabel acak, dan sebagainya. Kekuatan dari suatu algoritma kriptografi sangat ditentukan oleh bilangan acak.

Algoritma kriptografi dianggap paling kuat, yaitu *One time pad*, merupakan algoritma kriptografi yang sangat tergantung dengan

bilangan acak. Kunci yang digunakan diharuskan merupakan bilangan acak murni dan sepanjang pesan yang dikirim juga. Hal ini akan membuat kriptanalisis tidak mungkin dilakukan. Akan tetapi pada praktiknya, *One time pad* tidak dapat diterapkan secara praktis karena tidak mungkin untuk meng *generate* bilangan acak secara murni.

Selain *One time pad*, banyak sekali algoritma lain yang menggunakan bilangan *random* sebagai unsur penting dalam algoritmanya. Dengan memasukkan bilangan *random*, dianggap dapat menghilangkan kemungkinan penyerang menebak hasil dengan mengetahui algoritmanya.

Bilangan *random* yang murni memang dapat membuat kemungkinan menebak hasil menjadi tidak mungkin. Namun hingga saat ini kita tidak dapat membuat suatu *generator* bilangan acak yang dapat menghasilkan bilangan acak secara murni.

Masalah Pada *Random Number Generator*

Telah banyak algoritma *generator* bilangan acak yang diusulkan dan digunakan hingga saat ini. Algoritma-algoritma tersebut menggunakan berbagai pendekatan berbeda untuk menghasilkan bilangan *random* seacak mungkin.

Secara teoritis, kita tidak mungkin menghasilkan bilangan acak murni. Bilangan acak murni hanya didapatkan secara natural dari alam. Namun, walaupun tidak dapat menghasilkan bilangan acak murni, kita tetap berupaya agar bilangan acak tersebut sukar untuk ditebak.

Untuk mendapatkan bilangan acak banyak ahli yang mengusulkan berbagai algoritma dari yang sederhana hingga yang paling rumit. Dari penggunaan variabel waktu sebagai faktor pengacak hingga penggunaan teori biologi cellular. Algoritma-algoritma ini masing – masing memiliki kelebihan maupun kekurangan.

Banyak sekali faktor yang dapat kita bandingkan antara algoritma ini. Dalam penggunaannya, kita juga harus dapat memilih algoritma bilangan acak mana yang harus digunakan. Misalnya, ada algoritma bilangan acak yang sangat kuat namun memiliki waktu *generate* yang sangat lama, tentu saja algoritma ini tidak dapat digunakan untuk keperluan real time. Dalam hal ini, algoritma yang cepat lebih diutamakan walaupun mungkin tidak terlalu kuat.

***Pseudo – Random Number Generator* dan permasalahannya**

Karena tidak ada algoritma bilangan acak yang benar-benar sempurna, hingga saat ini kita masih terus mengembangkan cara-cara untuk mendapatkan bilangan *random* yang sukar ditebak. Kita berupaya untuk mendapatkan *generator* bilangan yang menyerupai *generator* bilangan acak. *Generator* bilangan ini meng*generate* bilangan acak semu (*pseudo random number*) yang memiliki tingkat kekuatan berbeda-beda menurut algoritma yang digunakan.

Pseudo-Random Number Generator menggunakan suatu *state* awal kemudian dengan suatu algoritma khusus akan menghasilkan bilangan acak semu. *State* awal yang digunakan diambil dari berbagai sumber yang dianggap cukup acak. Dengan demikian, *Pseudo Random Number Generator* ini akan menghasilkan suatu rangkaian bilangan yang menyerupai bilangan acak. Hasil dari *Pseudo Random Number Generator* ini akan digunakan dalam berbagai keperluan dalam kriptografi.

Pseudo Random Number Generator banyak sekali digunakan pada saat ini. Kemudahan implementasi dan kecepatannya merupakan faktor utama mengapa *Pseudo Random Number Generator* digunakan. Namun, sebagai algoritma *generator* bilangan acak, *Pseudo Random Number Generator* masih banyak memiliki kelemahan. Kelemahan utama adalah pada pemilihan *state* awal yang digunakan untuk meng *generate* bilangan acak.

Pada sebagian algoritma, misalnya *state* awal ini diambil dari variabel waktu *generate*. Dengan demikian, jika kita meng *generate* bilangan pada waktu yang berbeda akan menghasilkan bilangan yang berbeda pula. Namun hal ini merupakan kelemahan yang sangat nyata jika kita memperhatikan faktor yang dapat dimanfaatkan oleh penyerang. Misalnya, pada *generator* tersebut, penyerang mengetahui waktu *generate* bilangan acak, maka penyerang akan dapat menebak hasil bilangan acak nya. Hal ini sangat tidak diinginkan oleh desainer kriptografi.

Desain *Pseudo Random Number Generator* yang umum didasarkan pada pendekatan

heuristic. Pendekatan ini menganggap *Pseudo Random Number Generator* sebagai sebuah program untuk meng *generate* rangkaian bit yang lolos dari sekumpulan *random*ness *test*. Karena *random*ness *test* tersebut ditentukan oleh pendesain algoritma, maka dimungkinkan untuk merancang serangkaian tes yang efisien untuk menggagalkannya.

Akibatnya, sebelum menggunakan *Pseudo Random Number Generator* pada suatu aplikasi, kita perlu mengadakan suatu tes yang ekstensif untuk membedakan antara hasil *Pseudo Random Number Generator* dengan data yang benar – benar *random* serta akibatnya pada aplikasi tersebut. Menggunakan *Pseudo Random Generator* pada kriptografi merupakan hal yang cukup beresiko, karena penyerang dapat mencari kelemahan dari *Pseudo Random Generator* tersebut.

Bilangan acak sangat sulit untuk di *generate*, terutama di dalam komputer, di mana sangat ditentukan oleh *source* input dengan entropi yang sangat rendah. Hal ini membuat desain *Pseudo Random Number Generator* diarahkan untuk menyaring *source* tersebut sehingga mendapatkan nilai entropi yang tinggi untuk digunakan sebagai parameter kunci *Pseudo Random Number Generator* seperti *seeds* dan *key*. Misalnya kita dapat menjumlahkan entropi sehingga cukup untuk mendapatkan *Pseudo Random Number Generator* dalam *state* yang tidak mungkin ditebak.

Dalam kondisi ini, diasumsikan bahwa kriptografi *Pseudo Random Number Generator* tidak dapat diserang selama menghasilkan sampel output dalam jumlah yang terbatas. Kemudian *Pseudo Random Number Generator* tersebut harus di *re seed* ulang ke *state* lain yang tidak dapat ditebak menggunakan entropi baru yang disaring dengan proses *reseeding* yang cukup mahal secara komputasi.

Hal ini ditujukan untuk mendapatkan nilai yang sesuai sehingga proses untuk menebak kunci sulit dilakukan. Output non-deterministic dapat diperoleh dengan me *restart* algoritma pada instant yang tidak dapat ditebak dengan kondisi awal yang juga tak dapat ditebak dan periode yang cukup besar.

Pseudo Random Number Generator adalah hardware, software atau campuran antara keduanya. *Generator* software didasarkan pada penggunaan rumus matematika yang menghubungkan antara output dari *generator* dengan output sebelumnya (misalnya persamaan, BBS, MWC) atau didasarkan pada penggunaan enkripsi atau teknik hash untuk mendapatkan output yang tak dapat ditebak (cryptographic *Pseudo Random Number Generator*).

Tipe pertama mudah dipecahkan, misalnya dalam hal persamaan kongruensial, atau waktu *generate* yang lambat dalam hal penggunaan BBS. Cryptographic *Pseudo Random Number Generator* bergantung pada kekuatan teknik kriptografi yang digunakan dan/atau fungsi hash satu arah sebagaimana desainnya khusus (yaitu, tidak didasarkan pada model matematika, tanpa pembuktian keamanan atau keacakan).

Konsep entropi biasa digunakan untuk meningkatkan kekuatan *Pseudo Random Number Generator* menghadapi serangan, terutama serangan yang ditujukan untuk menemukan kunci. Metode dan pemikiran yang baik untuk mendesain *Pseudo Random Number Generator* dapat ditemukan dalam RFC 170. Mencampur dan mengkombinasikan *Pseudo Random Number Generator* yang berbeda meningkatkan output statistik dan meningkatkan ketahanan terhadap serangan.

Untuk menyerang *generator* campuran, *determination* yang berhubungan dengan bagian output merupakan langkah pertama yang diserang, kemudian dapat dibantu dengan memecahkan *generator* tunggal dengan output yang telah diketahui. Analisis Fourier mungkin sangat berguna untuk memecahkan *generator* kombinasi ini.

Angka yang benar-benar *random* diperlukan untuk *state* awal. Seluruh urutan proses *generate* bilangan *random* mungkin sangat tergantung pada *state* awal tersebut. Akibatnya, untuk mendapatkan bilangan yang benar-benar acak, *Pseudo Random Number Generator* harus memiliki kondisi awal yang benar-benar acak pula.

Serangan terhadap *Pseudo Random Number Generator*, terutama yang ditujukan untuk memecahkan kunci dan *state* internal dengan re*source* kekuatan prosesing yang begitu cepat membuat desain *Pseudo Random Number Generator* menjadi sangat sulit, karena kemungkinan untuk dapat dipecahkan selalu ada.

Pseudo Random Number Generator biasanya didasarkan pada teori matematika masih mudah

dipecahkan atau jika sukar dipecahkan akan memiliki waktu *generate* yang sangat lama. Masalah kepemilikan juga merupakan masalah lain dalam *Pseudo Random Number Generator*. Pengalaman menunjukkan bahwa orang yang dapat merancang algoritma *Pseudo Random Number Generator* yang baik juga dapat memecahkannya, atau bahkan mengetahui jalan pintas untuk melakukannya (misalnya dengan trap door, S-boxes).

Untuk mengatasi masalah – masalah pada *Pseudo Random Number Generator* tersebut, kita dapat menggunakan *Two Stage Random Number Generator* sebagai suatu alternatif. Desain berorientasi serangan pada *Two Stage Random Number Generator* telah dibuat menggunakan teknologi baru yang tersedia, terutama Intel RNG, dan model matematika yang tepat untuk menghasilkan output yang tidak dapat ditebak dan periode output yang cukup panjang.

Model implementasinya berhasil bertahan terhadap serangan serta cocok untuk berbagai fungsi (misalnya Pluggable Secured Email Client / PSEC). Walapun tidak ada *Pseudo Random Number Generator* yang benar – benar ideal, namun tidak ada alasan untuk menggunakan *Pseudo Random Number Generator* dengan basis desain yang tidak jelas.

Two Stage Random Number Generator

Ide dasar dari *Two Stage Random Number Generator* adalah untuk dapat mengubah fungsi transisi atau fungsi output dari sebuah *Pseudo Random Number Generator* sehingga dapat bergantung pada sebuah variabel *random* baru (misalnya hasil dari *randomizer*). Jika dalam mengubah fungsi output *Pseudo Random Number Generator* kita hanya mengombinasikan output aslinya dengan output *randomizer* menggunakan operator komputer yang sesuai, maka disebut dengan *Combination Generator*.

Model yang diimplementasikan mengadaptasi persamaan *Linear Congruential Generator* (LCG) dengan mengganti suatu nilai konstan dengan output dari sebuah *randomizer*. *Dynamic Reseeding* dari *randomizer* tersebut merupakan suatu pertahanan dari serangan dan membuat *generator* yang diadaptasi bekerja sebagai penguat periode *Pseudo Random Number Generator*.

Model Matematika *Two Stage Random Number Generator*

Sistem kriptografi yang didasarkan pada sebuah *One time pad* (OTP) tergantung pada sebuah salinan data *random* yang di*generate* dari sebuah *Pseudo Random Number Generator* atau data yang benar – benar *random* yang telah dipersiapkan dan disimpan database sistem.

Salinan kunci ini dipertukarkan secara aman dan rahasia antara kedua belah pihak. Kemungkinan untuk meng copy data *random* atau kemudahan untuk menebak hasil dari *Pseudo Random Number Generator* membuat desain *Pseudo Random Generator* diarahkan pada segi output yang tidak dapat ditebak, dan waktu *generate* yang dapat diterima.

Pseudo Random Number Generator yang didesain tersebut harus dapat dibuat kembali dengan pola yang identik. Kunci *One time pad* yang dihasilkan mungkin dikirim bersama pesannya, yang berarti mengirim pesan dengan panjang dua kali pesan asal atau mengirim sebuah digest dari kunci *One time pad* yang asli sehingga kunci dapat di*generate* kembali.

Misalkan, sebuah *Pseudo Random Number Generator* dengan output yang dihasilkan oleh persamaan

$$x_{n+1} = f(x_n, x_{n-1}, ...)$$

dan *Pseudo Random Number Generator* yang lain dengan output *I(n)*, dimana *n* adalah bilangan yang menunjukkan *state*. x_{n+1} dapat dimodifikasi menjadi

$$x_{n+1} = f(x_n, x_{n-1}, ..., I(n+1))$$

Output dari persamaan ini tidak menjamin benar-benar acak, namun akan lebih buruk jika kita menggunakan konsep dari persamaan polinom kongruen yang terkenal dan telah banyak diuji, misalnya *Linear Congruential Generator* (LCG). Lehmer mengusulkan sebuah metoda kongruensial untuk meng*generate* serangkaian *pseudorandom number*s $x_1, x_2, x_3, \ldots$.

Pada metoda ini, setiap bilangan dalam rangkaian meng*generate* bilangan setelahnya seperti dalam algoritma ini. *Integer* x_n dikalikan dengan sebuah konstanta *a* kemudian

dijumlahkan dengan sebuah konstanta *c*. Hasilnya kemudian dibagi dengan sebuah konstanta *M*, dan sisa hasil pembagiannya diambil sebagai x_{n+1}. Sehingga :

$$x_{n+1} = ax_n + c(\mathrm{mod}\, M) \qquad (1)$$

Algoritma LCG dengan perkalian berparameter ini mampu lolos pada pengujian waktu. Algoritma ini dapat diimplementasikan secara efisien. Walaupun algoritma Lehmer ini memiliki beberapa kelemahan statistik seperti kemudahan untuk menebak parameter, namun jika parameter algoritma dipilih dengan benar dan implementasi algoritma ini pada software benar, *generator* yang tersebut dapat menghasilkan sebuah rangkaian bilangan tidak berhingga secara semu yang memenuhi sebagian besar tes ke *random* an.

Teori yang tersedia masih belum lengkap dan implisit. Belum diketahui bagaimana kita dapat menghitung periode dari rangkaian dalam kasus penting dimana ketika modulus *M* adalah sebuah bilangan prima besar atau memiliki sebuah faktor prima yang besar (*composite modulus*).

Tes empiris yang baru menunjukkan bahwa beberapa *Pseudo Random Number Generator* seperti *Combo*, *NCOMBO*, dan *Lagged-Fibonacci Using* memiliki statistik yang lebih baik dari LCG.

Teorema Dasar

Dalam langkah selanjutnya, semua bilangan adalah *integer*, dan modulus *M* adalah sebuah bilangan prima. Misalkan sebuah rangkaian bilangan *x* memenuhi:

$$x_{n+1} = ax_n(\mathrm{mod}\, M)$$
$$(n = 0,1,2,3...) \qquad (2)$$

Teori dasar dari persamaan Lehmer menyatakan bahwa rangkaian tersebut mulai berulang pada sebuah nilai *n* dimana memenuhi:

$$a^n = 1(\mathrm{mod}\, M) \qquad (3)$$

Misalkan bilangan *integer* positif *n* yang terkecil yang memenuhi adalah *d*. Kemudian *d* disebut sebagai periode dari rangkaian tersebut, dan bilangan tersebut harus membagi (*M* - 1) dan sama dengannya pada kasus akar primitif.

Persamaan Lehmer yang dimodifikasi

Dengan melakukan modifikasi pada persamaan Lehmer, kita bisa mendapatkan sebuah algoritma yang dapat bertahan melawan serangan yang ditujukan untuk melakukan prediksi pada outputnya dengan menggunakan contoh hasil *generate* sebelumnya tanpa menghilangkan kemampuan statistiknya. Dari persamaan pertama (1) bisa kita dapatkan:

$$x_{n+1} = a^n x_0 + a^{n-1}c + ... + c$$
$$(\mathrm{mod}\, M) \qquad (4)$$

Persamaan (4) memiliki empat variabel yang tidak diketahui, yaitu (*a*, *c*, x_0,, *M*), sehingga empat *sampel* yang berurutan cukup untuk memecahkannya. Dalam model yang diusulkan, *randomizer* adalah sebuah aliran kunci *Pseudo Random Number Generator* yang terdiri dari algoritma cipher IDEA dalam mode OFB.

Sebuah *seed* yang benar-benar *random* yang cukup besar yang disebut *Basic Random Data* (BRD) harus dipersiapkan. BRD ini menjadi *feed* dalam rangkaian untuk teknik enkripsi dan outputnya menjadi *feed* kembali untuk input dari teknik enkripsi. Output dari *randomizer* ini dinotasikan dengan *I*. Operator dan proses yang digunakan didefinisikan sebagai berikut:

- BRD (*Basic Random Data*) dengan panjang 2048 byte atau 512 *word* empat byte.
- k_r kunci enkripsi yang digunakan untuk menghitung kunci enkripsi IDEA yang sebenarnya.
- *H* Fungsi k_r yang mahal secara komputasi. Dinotasikan sebagai $H(k_r)$.
- IDEA Operator algoritma enkripsi dengan kunci $H(K_r)$. Algoritma ini mengoperasikan I_q pada langkah ke *q* untuk meng*generate* I_{q+1}. Panjang dari array *I* adalah 512 *words* empat byte.

Misalkan I_0 = *BRD*, || adalah operator gabungan,

$I = IDEA(BRD, H(k_r)) = IDEA(I_0, H(k_r))$, $I_2 = IDEA(I_1, H(k_r))$

secara umum:

$$I_{q+1} = IDEA(I_q, H(k_r)) \qquad (5)$$

Untuk setiap $q \geq 0$, I_q adalah 2048 byte dan dikelompokkan dalam 512 word empat byte.
Persamaan umum untuk *I* adalah :

$$I_{q+1}(2i+1) \,||\, I_{q+1}(2i+2)$$
$$= IDEA(I_q(2i+2), H(k_r))$$
$$(i = 0,1,...,255) \quad (6)$$

Output dari *Two Stage Random Number Generator* di *generate* dengan menggantikan konstanta *c* di dalam persamaan (1) dengan output dari *randomizer I:*

$$(\mathrm{mod}\, M) \quad (7)$$

Dimana $n = 1,2..., q = 1 + \lfloor (n-1)/512 \rfloor$

$j = n \bmod 512$ dan $I_q(0) = I_q(512)$.

Terlihat bahwa persamaan (7) merupakan generalisasi dari persamaan (4) dimana konstanta *c* digantikan oleh nilai acak (I_s). Statistik dari *sampel* yang di *generate* oleh persamaan (4) diterima karena *c* adalah konstanta.

Karena $I_q(j)$ adalah variabel independen dan didasarkan pada data *random* kemudian menggantikan *c* dengan *I* secara logika meningkatkan keistimewaan statistik dari *Pseudo Random Number Generator*. Pada kenyataannya, x_n di *generate* dengan mengkombinasikan (menjumlahkan) output dari Lehmer dengan kondisi awal *random*.

Output dari dua atau lebih *generator* sederhana dengan operator seperti +, X atau XOR, menghasilkan sebuah perpaduan yang lebih baik dari kedua komponennya. Hal ini menjelaskan mengapa output dari *Two Stage Random Number Generator* pada persamaan (7) memiliki ke*random*an yang lebih baik dari LCG.

Untuk mendapatkan jumlah *sampel* minimum untuk memecahkan, jumlah persamaan harus sama dengan jumlah bilangan yang tidak diketahui (*I, a, x_0, k_r, M*). Jika jumlah *sampel* output adalah *v*, maka:

Jumlah persamaan
= *v*+*v*/2
= **jumlah variabel tidak diketahui**
= 4+*v*+512 (8)

Solusi untuk *v* adalah 1032 dan jumlah dari persamaan = jumlah dari variabel = 1548 dimana 516 adalah persamaan non linear dari bentuk (6). Himpunan persamaan sebelumnya tidak dapat diselesaikan dengan mudah tergantung pada kekuatan dari algoritma enkripsi dan hash yang digunakan.

Jika penyerang memiliki solusi dari persamaan sebelumnya, maka sejumlah terhingga byte dari setiap message dalam sesion yang sama harus dikirim. Setiap x_n adalah sebuah word empat byte. Penyerang mungkin menebak, mengkontrol atau mengetahui bagian dari BRD dan k_r, menggunakannya dalam satu serangan input, sehingga lebih baik mendefinisikan sebuah faktor keamanan *f > 1* yaitu:

$$f = \frac{\text{Jumlah teoritis sample yang dibutuhkan}}{\text{Jumlah sesungguhnya sample yang dibutuhkan}}$$

Karakteristik *Two Stage Random Number Generator*

Periode *Two Stage Random Number Generator*
Output dari *Two Stage Random Number Generator* di *generate* dari melalui persamaan (1) dengan menggantikan konstanta *c* dengan output dari *randomizer* c_i. Hal ini menyebabkan

$$x_{n+1} = a^r x_0 + a^{r-1}c_1 + a^{r-2}c_2 + ... + c_r$$

Misalkan output dari *randomizer* c_r diulang setelah periode *s*, sehingga
$c_r + s = c_r$.
Sehingga

$$(a^{ms-1})(x_0 + (\sum_{i=1}^{s} a^{s-1}ci)(a^s - 1)^{-1}) = 0$$
$$(\mathrm{mod}\, M) \quad (10)$$

Maka akan dengan mudah menentukan bahwa periode *p* dari *Two Stage Random Number Generator* diberikan oleh persamaan:

$$p = \text{kelipatan terkecil}$$
$$(\text{periode randomizer,} \quad (11)$$
$$\text{periode generator Lehmer asli})$$

Jumlah Modular *Two Stage Random Generator* dalam satu periode lengkap

Untuk mendapatkan jumlah modular dari sebuah rangkaian, jumlah dari semua output sampel harus dijumlahkan dalam sebuah

periode yang lengkap. Jumlah modular tersebut sama dengan:

$$-m(\sum_{I=1}^{s} c_i)(a-1)^{-1} \qquad (12)$$

Rata-rata *Two Stage Random Generator* dalam satu periode lengkap

Total rata-rata didefinisikan sebagai jumlah dari semua output yang mungkin dalam satu periode untuk semua kondisi awal dibagi dengan jumlah bilangan yang bilangan yang dijumlahkan. Jumlah dari semua kondisi awal adalah jumlah dari semua nilai dari nol hingga *M-1*, yang sama dengan *M(M-1)/2*.

x_0	*Sampel* pertama	*Sampel* kedua	...
0	c_1	$ac_1 + c_2$	···
1	$a + c_1$	$a^2 + ac_1 + c_2$	···
...	...	...	...
$a^2 x_0$	$a^3 x_0 + c_1$	$a^4 x_0 + ac_1 + c_2$	···
...	...	...	...
$M-1$	$a(M-1) + c_1$	$a^2(M-1) + ac_1 + c_2$	···

Table 1: Proses *Generate* pada TSRG

Kolom pertama merupakan bilangan antara 0 hingga *M*-1, sehingga rata-ratanya adalah (*M* - 1)/2. Kolom kedua terdiri dari *M* elemen dari sebuah *generator* LCG dengan konstanta c_1 dan pengali *a* dan *sampel*

$$(1-a)^{-1} c_1 (\mathrm{mod}\, M)$$

Rata – ratanya juga merupakan (*M*-1)/2 dan untuk *M* yang sangat besar, mendekati *M*/2. Hasil yang sama juga berlaku untuk kolom yang lain. Sebagai contoh, sebuah *Pseudo Random Number Generator* yang merupakan *Two Stage Random Generator* dengan *randomizer* LCG dengan modulus 5 dan pengali 3 dengan kondisi awal 1 dinotasikan sebagai *L*(5, 3, 1) dan mengadaptasi Lehmer *Generator L*(7, 5).

Tabel berikut ini merupakan hasil dari persamaan tersebut, yang menyebabkan rata-rata total 3 dan kondisi awal yang buruk 2 serta periode *p Two Stage Random Generator* adalah $p = (6 \times 4)/2 = 12$

Arth. Avr	x	Mod. Sum	O. Avr	O. Std	0	1	2	3	4	5	6
2.5833	0	3	1.714286	1.380131	2	4	1	0	1	3	1
2.5833	1	3	1.714286	1.380131	2	4	1	0	1	3	1
2	2	3	1.714286	1.380131	3	0	6	0	3	0	0
3.75	3	3	1.714286	1.380131	1	0	1	4	2	1	3
3.75	4	3	1.714286	1.380131	1	0	1	4	2	1	3
2.5833	5	3	1.714286	1.380131	2	4	1	0	1	3	1
3.75	6	3	1.714286	1.380131	1	0	1	4	2	1	3
3					12	12	12	12	12	12	12

Table 2 : Statistik *Two-Stage* LCG

Dimana :
Arth. Avr = *Aritmetic Average*
O. Std = *Occureness Standart Deviation*
Mod. Sum = *Modular Sum*
O. Avr = *Occureness Average*

Untuk kondisi awal yang buruk $x_0 = 2$, periode dikurangi menjadi 4, yaitu kurang dari modulus 7, sehingga tidak semua himpunan output 0,…,6 di *generate*. Urutan output memiliki *X* = 2 dengan sebuah *rate* dua kali lebih besar dari yang lainnya (yaitu memiliki *redundancy* yang lebih tinggi).

Entropi yang jelas dari output sangat dipengaruhi, yang dapat membantu untuk mendeteksi kondisi awal yang buruk dan *reseeding* dinamik dengan periode yang lebih pendek. Total rata – rata yang dihitung adalah $(M-1)/2 = 3$.

Hasil ini memiliki sebaran yang merata. Sehingga untuk mendapatkan output *Two Stage Random Generator* yang merata, dilakukan *reseed* terhadap *randomizer* dengan menggunakan nilai x_n saat itu sebagai kondisi awal yang baru.

Desain dan komponen *Two Stage Random Generator*

Pada bagian ini, kita akan membahas tentang desain *Two Stage Random Generator*, komponen – komponennya, serta prosedur kerja dari *Two Stage Random Generator* tersebut. Ide dasar dari *Two Stage Random Generator* adalah menggunakan beberapa data awal *random* (BRD, *Basic Random Data*) dan melakukan *feed* BRD tersebut terhadap cipher blok misalny IDEA untuk mendapatkan bilangan – bilangan yang terlihat *random* c_i.

Dalam hal ini, c_i pada dasarnya diperoleh dengan mengaplikasikan IDEA ke $c_{i\text{-}2048}$. Kunci dari IDEA didapatkan dengan melakukan hash terhadap kunci sebelumnya, *state* internal dan output dari *Two Stage Random Generator*. Hasilnya kemudian digunakan untuk melakukan *feed* terhadap sebuah *generator* Lehmer yang telah digeneralisasi untuk menghasilkan output.

Perkiraan entropi pada x_i dan c_i digunakan untuk mengkontrol periode dari kunci spesifik sebelum di *update*. Sebuah *randomizer* yang sistematik melakukan *reseed* yang dilakukan setiap sejumlah *M* sampel agar output dari *Two Stage Random Generator* menjadi merata dan untuk memperbesar periodenya.

Desain Perspektif

Sebuah *Pseudo Random Number Generator* bergantung pada algoritma kriptografi atau polinom dengan kondisi awal yang *random* atau sumber entropi yang terus menerus disaring.

Pada kasus terakhir, *Random Number Generator* dapat didesain dengan mengumpulkan dan menyaring entropi yang cukup dari bermacam sumber yang independen. Jika jumlah sumber independen tersebut sangat besar dan penyaringan entropi terhadapnya akurat, maka *Random Number Generator* yang mendekati sempurna akan didapatkan.

Jika sumber tersebut tidak memiliki bit entropi yang baru, maka kita dapat menggunakan mode kriptografi hingga entropi cukup atau selesai menghasilkan output. Mode kriptografik menghubungkan kemampuan diserang bergantung pada jumlah sampel yang dihasilkan secara kriptografi dan teknik *generate* yang digunakan.

Dalam sistem kriptografi *One time pad*, data *random* yang identik harus dimiliki oleh pengirim dan penerima. Jika *Pseudo Random Number Generator* digunakan untuk sistem kriptografi *One time pad*, maka objek identik yang terdiri dari data dan program harus dimiliki oleh kedua belah pihak.

Dalam kasus yang menggunakan sumber entropi, penerima harus mengetahui secara pasti dimana perubahan mode dilakukan, dan bagaimana *state* setelah melakukan perubahan mode kriptografi. Hal ini dapat dilakukan dengan menggunakan sebuah bahasa script khusus. Format dari bahasa script tersebut bisa terdiri dari bit entropi yang dipisahkan kemudian digabungkan dengan sejumlah sampel yang dihasilkan dengan menggunakan *state* yang baru.

Panjang dari program script tersebut kurang dari panjang data *random* yang di*generate* dan perbandingan diantara keduanya berbanding terbalik dengan jumlah perubahan mode. Nilai minimum dari rasio perbandingan sebelumnya didapatkan ketika *Pseudo Random Number Generator* bekerja dalam mode kriptografi secara permanen sehingga dapat diserang.

Dalam hal ini program script adalah satu *record* yang terdiri dari kondisi awal dan jumlah *sampel* output *Pseudo Random Number Generator* yang diinginkan. Jika pergantian mode terjadi sangat sering, maka panjang program script tersebut mungkin menjadi sangat panjang dan dapat dibandingkan dengan panjang data *random* itu sendiri.

Algoritma *Pseudo Random Number Generator* Yarrow yang sangat bagus, memiliki kelemahan ini ketika digunakan dalam keamanan *One time pad*. Hal ini berarti harus ada pendekatan lain yang dapat digunakan dengan hanya sedikit mengorbankan keistimewaannya. Pendekatan yang diusulkan adalah dengan mengumpulkan entropi yang cukup dan menyimpannya dalam suatu *pool* yang disebut *limited length pool* dan menggunakannya secukupnya untuk meng *generate* data *random* kriptografi dan *reseeding* ke *state* yang *unguessable* dilakukan sebelum meng *generate* sampel yang cukup untuk melakukan cracking.

Pool bit *random* (BRD dan kunci IDEA) didistribusikan secara aman kepada penerima. Jumlah sampel cukup untuk memecahkan didapatkan dari model matematik *generator* dan *reseeding* dinamik ke *state* yang *unguessable* dilakukan dengan menggunakan perhitungan yang mahal secara komputasi bergantung pada

entropi yang jelas dari *state* internal dan output dari *generator* itu sendiri. *Two Stage Random Generator* memiliki dua tipe mekanisme *reseeding*.

Yang pertama mengubah key *randomizer* ke *state* yang *unguessable*. Yang kedua terjadi setelah setiap *M* sampel untuk memaksa siklus *randomizer* menjadi *M*. Dalam hal ini, *pool* awal di *load* kembali ketika x_n digunakan sebagai kondisi awal untuk siklus berikutnya.

Permasalahan dalam Implementasi *Two Stage Random Generator*

Penerapan *Pseudo Random Number Generator* merupakan masalah tersendiri. *Programmer* harus memastikan bahwa hasil output merupakan hasil yang benar – benar diharapkan. Hal ini dapat dilakukan dengan baik dengan menggunakan modul komposisi dan modul vektor yang telah dites dengan baik.

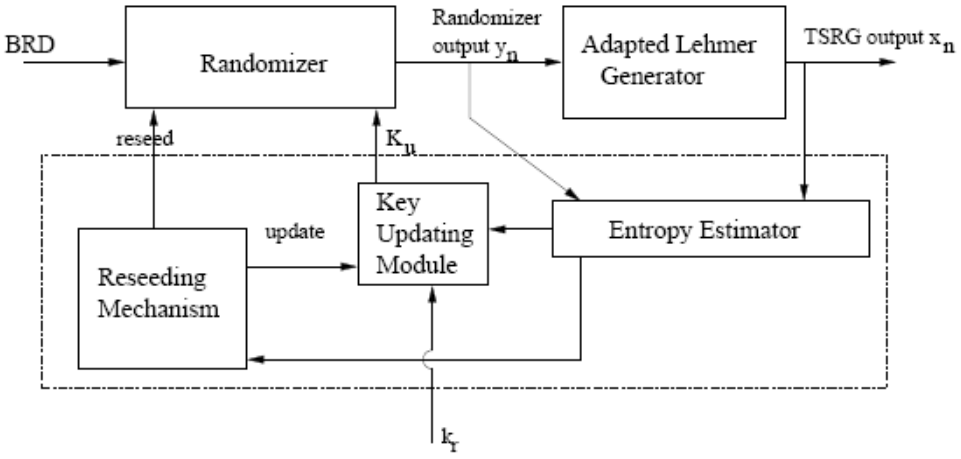
Modul komposisi dari *Two Stage Random Generator* adalah modul LCG, teknik cipher IDEA , *driver* Intel *Random Number Generator* dan modul *randomness testing* FIPS140-1. Semua modul tersebut dihasilkan oleh sumber yang dapat dipercaya dan dipublikasi dalam source code di Internet sehingga dapat diteliti dan ditest oleh banyak orang.

Dalam mendesain *Two Stage Random Generator*, hal – hal berikut perlu untuk diperhatikan:

- Desain berorientasi serangan. Yang artinya semua serangan yang diketahui benar-benar diperhitungkan dalam proses desain.
- Menggunakan primitif yang telah diuji dengan baik
- *Pseudo Random Number Generator* yang digunakan berdasarkan teori yang kuat, sehingga tidak mudah dipecahkan. (IDEA, yang digunakan di sini merupakan teknik enkripsi yang kuat).
- Komponen yang independen, sehingga dapat dengan mudah dikembangkan atau dimodifikasi.
- Efisien, sehingga dapat diimplementasikan dengan baik pada software sekalipun.
- Tahan terhadap *backtracking*, melakukan *update* key satu arah.
- Tahan terhadap serangan *Chosen-Input.*
- Dapat menghasilkan *state Pseudo Random Number Generator* yang baru secara dinamik.
- Memiliki karakteristik statistik yang baik. (periode besar, lolos *randomness test*)
- Didasarkan pada teori yang kuat.

Komponen *Two Stage Random Generator*

Komponen *Two Stage Random Generator* ditunjukkan dalam gambar berikut ini. *Two Stage Random Generator* pada dasarnya terdiri atas *Randomizer*, *Generator* Lehmer yang dimodifikasi, *Estimator* entropi, modul peng *update* key dan mekanisme *reseeding*.



Gambar 1 Komponen TSRG

Randomizer

Randomizer adalah sebuah *generator stream* yang didesain menggunakan teknik enkripsi dalam mode *feedback*. Vektor awalnya adalah blok enkripsi bukan satu (8 byte dalam kasus cipher IDEA) tetapi vektor ini merupakan 2048 byte yang di *feed* untuk dienkripsi dalam rangkaian kemudian di *feed* sebagai input kembali.

Output dari algoritma enkripsi dalam mode OFB setelah setiap iterasi dapat didekati dengan sebuah proses *random draw without replacement*. Dari hubungan sebelumnya, ukuran siklus rata – rata dihasilkan dari pendekatan dan untuk DES, biasanya dinyatakan secara umum untuk teknik enkripsi lainnya bahwa ukuran rata – rata siklus teknik enkripsi dengan blok input dari ukuran bit *m* dalam mode OFB adalah 2^{m-1}.

Array BRD dibagi menjadi blok input yang di *feed* ke dalam teknik enkripsi dalam rangkaian dan output yang berhubungan dengannya di *feed* lagi pada urutannya. Ukuran rata –rata siklus dalam kasus ini tidak sama dengan ukuran pada OFB dalam satu blok.

Penggunaan multi blok di sini adalah agar *generator* tahan terhadap serangan *key-compromise* dan bahkan untuk tahan terhadap serangan dimana penyerang memiliki *tool* untuk memecahkan algoritma enkripsi tersebut.

Berdasarkan model matematika pada bab sebelumnya, rangkaian dari sampel output yang menggunakan key spesifik sebelum *reseeding* tidak cukup untuk memecahkan himpunan persamaan yang independen terhadap teknik enkripsi yang digunakan atau untuk menentukan apakah algoritma ini dapat dipecahkan atau tidak.

IDEA adalah teknik enkripsi yang biasanya berada dalam mode OFB. IDEA diimplementasi dalam *Two Stage Random Generator* karena :

- Implementasi *Two Stage Random Generator* memungkinkan teknik enkripsi yang lain dapat diimplementasikan secara mudah.
- *Two Stage Random Generator* didesain efisien dalam implementasinya pada software, mengenkripsi 64 bit dengan menggunakan 128 bit key.
- *Two Stage Random Generator* telah dites dengan baik dan telah banyak dipublikasikan sejak pertama kali dikembangkan pada tahun 1990.
- Titik lemahnya telah dipelajari dan dipublikasikan. Modul verifikasi kunci belum diimplementasikan.

***Generator* Lehmer yang dimodifikasi**

Tugas dari modul ini adalah untuk memastikan ke *random* an dari output dan menambah kompleksitas sehingga mempersulit percobaan untuk memecahkannya. Keuntungan yang lain adalah bahwa *generator* ini bekerja sebagai penguat periode dengan faktor penguat *M*.

Nilai *processing cost* dapat diabaikan karena relatif kecil dibandingkan dengan waktu pemrosesan enkripsi. Karena *randomizer* di *reseed* setiap *M* sampel dan menggunakan persamaa (11) periode dari *Two Stage Random Generator* berada dalam urutan M^2. Setelah *reseeding*, kondisi awal Lehmer adalah kondisi yang terakhir kali di *generate* untuk memperkuat periode sebesar *M* kali.

Untuk semua kondisi awal dengan rentang waktu *update* yang tertentu dan *randomizer* di *reseed* setiap *M* sampel, maka periode yang diharapkan berada dalam urutan M^3 dan output tersebar secara merata.

Modul *Estimator* Entropi

Tugas dari modul ini adalah untuk menghitung entropi yang terlihat dari output *randomizer* dan *Two Stage Random Generator*. Modul peng *update* key dan mekanisme *reseeding* memanfaatkan outputnya menggunakan persamaan (15, 14).

Nilai entropi dari output *Two Stage Random Generator* dan *randomizer* dapat dihitung dengan menggunakan persamaan

$$H = -\sum (p \log p)$$

dimana p adalah probabilitas *generate* dari setiap karakter. Dan fungsi $\sum$ diterapkan untuk semua karakter dari ASCII 0 hingga 255. Perhitungan entropi selesai jika output di *generate* dalam kondisi berikut ini:

- Dua tabel masukan sebesar 256 (tabel entropi) dialokasikan untuk *randomizer* dan output dari *Two Stage Random Generator* dengan cukup baik, dan semua masukan diinisialisasi dengan nol.
- Setiap karakter di *generate* dalam *randomizer* atau dalam *Two Stage Random Generator*, nilai ASCII nya dikalkulasi dan masukan dalam tabel yang memiliki alamat yang bersesuaian dengan ASCII tersebut di *increment* dengan satu.
- Pada kasus yang spesifik, yaitu ketika nilai entropi perlu dihitung, probabilitas *p* dari setiap karakter dihitung.
- Menghitung entropi

$$H = \sum(-p \log p)$$

untuk 256 karakter.

Penghitungan entropi merupakan salah satu parameter sistem yang dapat diatur, dalam implementasi *Two Stage Random Generator* hal ini diatur untuk mengetes setengah dari periode *update* key yang diharapkan yang didefiniskan pada persamaan (14).

Pada permulaan *Two Stage Random Generator*, kedua tabel entropi diinisialisasi menjadi nol (*reseeded*) dan entropi BRD dihitung. Entropi *randomizer* dan entropi *Two Stage Random Generator* diberikan pada entropi BRD.

Periode *reseed* key yang pertama kali diperkirakan dihitung. Setelah setengahnya, percobaan lain yang didasarkan pada output yang di*generate* dilakukan. Ketika periode update key ditemui, key di *update* ke *state* yang *unguessable* dan tabel entropi diinisialisasi kembali.

Modul peng *update* key

Modul ini bertugas untuk mengubah key k_r kedalam *state* key yang *unguessable* k_u dengan menggunakan fungsi yang mahal secara komputasi. Meng *update* ke dalam *state* baru menggunakan output dari *Two Stage Random Generator* dikombinasikan dengan output dari *randomizer* (tidak diketahui oleh penyerang) dan nilai key saat ini untuk mendapatkan nilai key yang baru.

Walaupun key *updating* terjadi setiap periode *pk* yang ditentukan oleh persamaan (14), *updating* ini telah dipersiapkan sejak output di *generate*.

Pk = periode update key

$$= \left(\frac{\text{sampel minimum untuk cracking}}{\text{faktor keamanan}}\right) \times \text{Entropi output } randomizer \times \text{Entropi output TSRG}$$

Dalam persamaan (14), kita mengasumsikan bahwa *dependency* linear antara periode *update* key dan entropi (kurang dari 1). Kondisi awal yang buruk dan siklus *randomizer* yang pendek memaksa *key updating* dilakukan dalam periode yang kecil. Key yang baru dihasilkan dengan persamaan :

$$k_u(z+1) = hash(k_u(z) \,||\, (x_0 y_0) \,||\, (x_1 y_1) \,||\, \ldots (x_{pk} y_{pk})) \quad (10)$$

Dimana *hash* adalah fungsi hash SHA yang telah dimodifikasi yang mengembalikan nilai 128 bit, *z* adalah angka *update*, x_{pk} dan y_{pk} adalah sampel output terakhir sebelum dilakukan *update* key sebagaimana didefinisikan dalam persamaan (14).

Mekanisme *Reseeding*

Modul ini memiliki dua buah fungsi, yang pertama adalah untuk me *reseed randomizer* setiap *M* sampel dan yang keduan adalah untuk memicu *update* key secara dinamik. *Reseeding randomizer* menggunakan nilai dari output x_n sebagai kondisi awal yang baru dari operasi berikutnya untuk mendapatkan output dengan sebaran yang merata.

Prosedur kerja *Two Stage Random Generator*

Algoritma berikut ini menggambarkan mekanisme *generate* dari *Two Stage Random Generator*

```
Algorithm TSRG (Tsrg-Length,X0)
   Generator_Initial_Condition =
   X0
   Re_Seed ()
   Tsrg_Absolute_Counter = 0
   Ku_Temp = Kr
   DO WHILE
   Tsrg_Absolute_Counter <=
   Tsrg_Length
      Yn =
      Generate_Randomizer_Sampel
      ()
```

```
      Xn =
      Generate_Generator_Sampel
      ()
      Ku_Temp = hash (Ku_Temp ||
      Xn||Yn)
      Modular_Increment
      (Tsrg_Moduler_Counter)
      IF Toggle
         Update_Randomizer_
         Table()
         Update_Generator_
         Table()
         IF (Tsrg_Modulr_Counter
         >=
         0.5*Key_Update_Period)
            Toggle = False
            Recalculate_Key_
            Update_Period()
         ENDIF
      ENDIF
      IF MOD
      (Tsrg_Absolute_Counter=0)
         ReSeed()
      ENDIF
      IF (Tsrg_Modulr_Counter >=
      Key_Update_Period)
            Update_Key ()
      ENDIF
   ENDDO
END TSRG


Algorithm Re_Seed ()
   Toggle = True
      Randomizer_Input_Data =
   Basic_Random_Data
   Er = Apparent_Entropy
   (Randomizer_Input_Data)
   Eg = Er
   Key_Update_Period =
   Max_Sampels*Eg*Er/
   Factor_Of_Safety
   Tsrg_Module_Counter = 0
   Initialize
   (Generator_Entropy_Table)
   Initialize
   (Randomizer_Entropy_Table)
END Re_Seed


Algorithm
Recalculate_Key_Update_Period ()
   Er = Apparent_Entropy
   (Generator_Entropy_Table)
   Eg = Apparent_Entropy
   (Randomizer_Entropy_Table)
   Key_Update_Period=
   Min_Sampels_To_Crack*Eg*Er/
   Factor_Of_Safety
END
Recalculate_Key_Update_Period


Algorithm Update_Key ()
   Toggle = True
   Randomizer_Key =
   Size_Adaptor (Ku_Temp)
   Er = Apparent_Entropy
   (Randomizer_Input_Data)
   Eg = Er
   Key_Update_Period =
   Max_Sampels*Eg*Er/
   Factor_Of_Safety
   Tsrg_Module_Counter = 0
   Initialize
   (Generator_Entropy_Table)
   Initialize
   (Randomizer_Entropy_Table)
END Update_Key
```

Pertahanan *Two Stage Random Generator* terhadap Serangan

Permasalahan yang paling penting adalah penyerang, termasuk yang mengetahui desain *Random Number Generator*, harus tidak dapat membuat prediksi yang berguna tentang output dari *Random Number Generator* tersebut yang di *generate* sebelum maupun sesudah sampel yang telah diketahuinya.

Entropi dari output *Random Number Generator* harus sedekat mungkin dengan panjang bit nya sendiri. *State*ment dari output sebelumnya penting, namun tidak cukup karena banyak *Pseudo Random Number Generator* yang diterima secara statistik dapat dengan mudah dipecahkan.

Sebagai contoh, sekarang telah diketahui teknik untuk memecahkan semua *generator* polinomial kongruen. Karena tidak ada kriteria yang praktis dan dapat diterapkan secara universal untuk menyatakan keamanan dari *Pseudo Random Number Generator*, pertahanan terhadap semua serangan yang diketahui harus dilakukan hingga ditemukan serangan baru atau beberapa faktor keamanan ternyata terlalu diperhitungkan.

Berikut ini adalah beberapa serangan yang umum dilakukan dan pertahanan *Two Stage Random Generator* terhadapnya:

Direct Cryptanalysis Attact Defense

Sebagian besar kriptografi *Pseudo Random Number Generator* bergantung pada kekuatan enkripsi dan teknik hash untuk bertahan menghadapi serangan. Output dari *Two Stage Random Generator* tidak dapat digunakan untuk menyusun serangan ini karena tidak ada sampel yang cukup untuk menyelesaikan himpunan persamaan yang tidak *respective* pada penggunaan enkripsi dan fungsi hash.

Algoritma u*pdating* key yang mahal secara komputasi memaksa penyerang untuk menyelesaikan sebuah himpunan persamaan yang baru dengan jumlah bilangan yang tidak diketahui lebih banyak dari jumlah persamaan. Hal ini berarti bahwa *Two Stage Random Generator* tidak bergantung sepenuhnya pada algoritma enkripsi dan hash untuk bertahan terhadap serangan.

Input *Attack* Defense

Two Stage Random Generator mendapatkan input dari tiga sumber (Intel *Random Generator*, pergerakan mouse dan *timing* penekanan tombol). *Two Stage Random Generator* menerima input 2500 byte hanya dari Intel *Random Number Generator* saja. Jumlah bit *random* yang dimaksimalkan, dikumpulkan dari pergerakan mouse dan *timing* penekanan tombol kemudian disaring.

Data sumber entropi yang tinggi pada Intel *Random Number Generator* dikombinasikan dengan data berentropi rendah yang telah disaring. Walaupun output dari Intel *Random Number Generator* dijamin dapat diterima menggunakan tes ke *random* an FIPS 140-1, data yang telah dikombinasikan harus dites kembali menggunakan FIPS 140-1, di dalam modul ini untuk meyakinkan bahwa tidak ada demon program penyerang yang disusun untuk mengemulasikan *generate* dari data *random* yang diperlukan.

Sebuah eksklusif or diantara data *random* dari Intel *Generator* dan dari data dengan entropi rendah digunakan untuk mengkombinasikan data tersebut. Eksklusif or tidak aman secara sempurna, namun memiliki *response time* yang optimum. Data yang telah dikombinasikan dan di tes digunakan untuk meng *generate* BRD dan k_r. Mengetes ke *random* an dari BRD dan menyebar entropi dengan menyatukan modul dalam aplikasi mampu bertahan terhadap serangan *chosen-input*.

Jika penyerang mengetahui sebagian dari input tersebut dan mampu mengamati secara langsung output dari *Two Stage Random Generator*, maka ia memiliki kemungkinan untuk menyusun serangan *combined cryptananlysis known -input.*

Combined Cryptanalysis Known-Input *Attack*s

Misalkan penyerang mengetahui sebagian dari input yang mungkin merupakan bagian dari BRD dan pada saat bersamaan output dari *Two Stage Random Generator* dapat dibaca. Mengetahui desain dari *Two Stage Random Generator*, jika bagian dari BRD lebih besar dari 1/*f* seperti didefinisikan pada persamaan (14), serangan dapat berhasil.

Hal ini menjelaskan mengapa kita menggunakan parameter faktor keamanan (*factor of safety*) dalam persamaan ini.

***State* Compromise *Attack* Defense**

Misalkan untuk suatu alasan, penyerang mempelajari *state* internal *I*, menyerang beberapa kelemahan, melakukan penetrasi temporer, mampu melakukan kriptanalisis dengan sukses dan sebagainya. Sekali *state* diketahui, maka serangan selanjutnya dapat disusun kembali.

Backracking *Attack*s

Sebuah *backtracking attack* menggunakan hubungan antara *state Pseudo Random Number Generator I* pada waktu *t* untuk mempelajari output *Pseudo Random Number Generator* sebelumnya. Algoritma *update* key adalah sebuah proses satu arah dengan menggunakan pemotongan output dari fungsi hash.

Sehingga tidak mungkin untuk mendapatkan key dari sampel sebelumnya, walaupun *state*-compromise *attack* telah berhasil. Hal ini merupakan alasan lain menggunakan teknik enkripsi IDEA yang memiliki panjang key 128 bit sedangkan output dari fungsi hash adalah 160 bit.

Permanent Compromise *Attack*s

Sebuah *Permanent Compromise Attacks* terjadi jika sekali penyerang mengetahui *I* pada suatu waktu *t*, semua nilai *I* sebelum dan sesudahnya menjadi mudah untuk diserang. Nilai sebelumnya tidak dapat diprediksi, namun nilai sesudahnya dapat diprediksi jika output dari *Two Stage Random Generator* dapat diamati atau dihitung (misalnya, parameter *M* dan *a* dalam persamaan Lehmer diketahui).

Two Stage Random Generator tidak memproses input sehingga tidak dapat bertahan dari *state compromise attack*. Jelas bahwa *Two Stage Random Generator* harus bertahan terhadap *compromise extention attack* seteliti mungkin.

Di dalam model implementasi yang diusulkan, variabel *state* internal dan variabel key disimpan dalam virtual memori Windows. *Memory Manager* akan memindahkan dari memori *page* ke harddisk dimana *state* dapat diketahui. *State Compromise Attack* masih mungkin dapat dilakukan dengan mengalokasikan variabel *state* internal (*key* dan *I*) didalam *page* memori yang secara diakses secara ekstensif.

Kode berikut ini menunjukkan kebutuhan memori untuk segmen data statik *Two Stage Random Generator* dan bagaimana serangan dapat dihindari.

```
BYTE Generator_Entropy_Table
[256];
BYTE Basic_Random_Data [2048];
BYTE Key_Random [16];
BYTE Key_Updated [16];
BYTE Internal_State[2048];
BYTE
Randomizer_Entropy_Table[256];
```

Untuk setiap karakter yang di *generate*, tabel entropi di*update* menghindari *swap* disk pada *page* tersebut.

Other *State* Compromise *Attack*s

Serangan untuk menebak secara iteratif dan *meet in the middle attacks* yang biasanya dihubungkan dengan *input attacks* tidak dapat diterapkan pada *Two Stage Random Generator*.

Randomness Test* pada *Two Stage Random Generator

Randomness test dilakukan dengan menggunakan empat *randomness testing program* (OTP, ENT, FIPS140-1, 2 dan Diehard). Dengan BRD kondisi awal, dan key awal yang sama, output dengan panjang yang berbeda di *generate* untuk melihat perubahan hasil *randomness test*. *Array* 2500 byte yang daripadanya semua kondisi awal berasal harus lolos tes dengan FIPS140-1.
Tes yang sama juga diberlakukan pada setiap *Two Stage Random Generator* dengan blok output 2500 byte. Menerapkan program ENT pada 40000 byte *sampel* output menghasilkan bahwa *Two Stage Random Generator* lolos tes tersebut. 1 mega byte output *random* lolos tes FIPS140-2 dengan menggunakan program OTP.

Untuk tes dengan Marsaglia Diehard, tabel berikut menampilkan hasilnya dibandingkan dengan LCG yang asli. Hasil dari semua tes di atas adalah :

Two Stage Random Generator lolos dalam semua *randomness test* dimana *generator* lain mungkin gagal.

	LATTIC	PARKLOT	MTUPLE	OPSO	BDAY	OPERM	RUNS	RANK
LCG	Fail	PASS	Fial	Fail	PASS	PASS	PASS	PASS
TSRG	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS

Table 3 Hasil Tes Diehard

Kesimpulan

Two Stage Random Generator merupakan salah satu pendekatan yang dapat digunakan untuk mendapatkan bilangan *random* seacak mungkin. *Two Stage Random Generator* menghubungkan antara dua buah *Pseudo Random Number Generator* dengan menjadikan output dari satu *Pseudo Random Number Generator* sebagai salah satu input masukan *Pseudo Random Number Generator* yang lain.

Generator yang pertama merupakan *Pseudo Random Number Generator* sembarang (*randomizer*) yang outputnya digunakan untuk mengubah fungsi output dari *Pseudo Random Number Generator* yang kedua, (*adapted generator*) sehingga nilainya tergantung pada output *randomizer*.

Karakteristik dari *Two Stage Random Generator* adalah *repeatable, jump ahead*, mudah diimplementasikan, mudah dalam *generate*, dan sulit untuk diprediksi. Semua karakteristik tersebut telah dipelajari dan dievaluasi. Salah satu algoritma yang diimplementasikan, yaitu *hybrid Pseudo Random Number Generator*, didapatkan dengan melakukan modifikasi pada sebuah *generator* Lehmer dengan output dari sebuah *randomizer* kriptografik.

Randomizer yang diimplementasika merupakan sebuah *generator* aliran kunci yang menggunakan algoritma IDEA dalam mode OFB dengan *dynamic key updating* dan *reseeding* yang dilakukan secara terus menerus untuk menghalangi usaha memprediksi output, mendapatkan output yang memiliki sebaran merata, dan memperpanjang periode keseluruhan *Two Stage Random Generator*.

Serangan kriptanalisis dapat dipertahankan dengan melakukan *reseeding* pada *randomizer* sebelum menghasilkan jumlah minimum sampel yang dapat digunakan untuk memecahkan, didasarkan pada model matematika deduksi. *Dynamic key Updating* mampu mempertahankan terhadap *permanent compromise*, *backtracking* dan *iterative guessing attack.*

Melakukan pengetesan terhadap *randomness* dari BRD yang digunakan sebagai input dari *randomizer* dari Intel *Random Number Generator* dan menyebar entropi pada implementasinya mampu mempertahankan terhadap serangan *chosen input attack.*

Karakteristik yang diperlukan untuk suatu implementasi bisa didapatkan dengan pemilihan *randomizer* yang tepat dan dengan memodifikasi fungsi *generator*, sehingga *Two Stage Random Generator* sesuai dengan kebutuhan kita.

Hingga saat ini memang tidak ada *generator* bilangan acak yang sempurna, namun *Two Stage Random Number Generator* merupakan salah satu upaya untuk mendapatkan bilangan *random* yang lebih baik.

Referensi:

[1]. Hussein Gamal, Dakroury Yasser, Hassen Bahaa, Badr Ahmed, "*Two-Stage Random Generator* (TSRG); *Attack*-Oriented Design and Implementation" 2002.

[2]. Abdulai Mustapha, "Inexpensive Parallel *Random Number Generator*", 2003.

[3]. CryptoBytes, Volume 4, *Number* 1 - Summer 1998.

[4]. P. Gutmann. "Software generation of practically strong *random number*s", USENIX Security Symposium, 1998.

[5]. http://wwws.irb.hr/~stipy/*random*/Diehard.html