Analisis Pengembangan Algoritma Yarrow Menjadi Algoritma Fortuna

M. Haekal Izmanda Pulungan - 13507020
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
If17020@students.if.itb.ac.id

***Abstract*—Yarrow dan Fortuna merupakan algoritma *pseudo-random number generation* (PRNG) atau pembangkit bilangan acak semu. Makalah ini memberi penjelasan mengenai konsep dari PRNG, algoritma Yarrow, algoritma Fortuna yang merupakan pengembangan dari algoritma Yarrow, dan analisis mengenai pengembangan tersebut.**

***Index Terms*—Fortuna, PRNG, Yarrow**

I. PENDAHULUAN

Pada praktiknya, bilangan acak sangat sering digunakan pada kriptografi saat ini. Sebagai contoh, pada *stream chiper*, bit *stream* acak yang seukuran dengan ukuran sebuah pesan digunakan untuk mengenkripsi atau mendekripsi pesan tersebut. Bilangan acak juga digunakan pada tanda-tangan digital (*digital signature*), kunci pada algoritma enkripsi AES, bilangan prima yang sangat besar pada algoritma RAS, dan juga digunakan pada beberapa sistem pengidentifikasi lainnya.

Secara spesifik berikut adalah penggunaan bilangan acak pada aplikasi-aplikasi kriptografi.

- *session* dan kunci pesan pada chiper simetrik, seperti triple-DES atau Blowfish
- *seed* pada *routine* yang membangkitkan sebuah nilai matematik, seperti sebuah bilangan prima yang sangat besar untuk RSA atau ElGamal
- *salt* untuk dikombinasikan dengan password
- *initialitation vectors* (IV) untuk *block chiper*
- nilai acak untuk instansiasi pada skema tanda tangan digital, seperti DSA
- bilangan acak untuk protokol autentikasi, seperti Kerberos
- pada protokol, untuk memastikan keunikan pada setiap protokol yang sama yang dijalankan

Beberapa bilangan acak yang dibangkitkan akan ditampilkan secara jelas, seperti IV. Namun, beberapa penggunaan lainnya akan menjaga kerahasiaan bilangan acak dan menggunakannya sebagai kunci. Beberapa aplikasi juga membutuhkan bilangan yang secara kuantitas sangat besar, seperti Kerberos akan membangkitkan ribuan kunci untuk setiap sesi tiap jamnya, sedangkan aplikasi lainnya mungkin bahkan hanya membutuhkan sebuah bilangan acak saja.

Pembangkit bilangan acak (*random number generator* atau RNG) dapat dibagi ke dalam 3 (tiga) kategori.

Pertama adalah True RNG (TRNG). TRNG menghasilkan data acak berdasarkan pada sumber-sumber fisik yang tidak terprediksi, seperti *noise* pada resistor, atau *noise* pada osilator, dst. Bagaimanapun juga, data acak yang dihasilkan umumnya pada *low rates*, seperti 20 kbps.

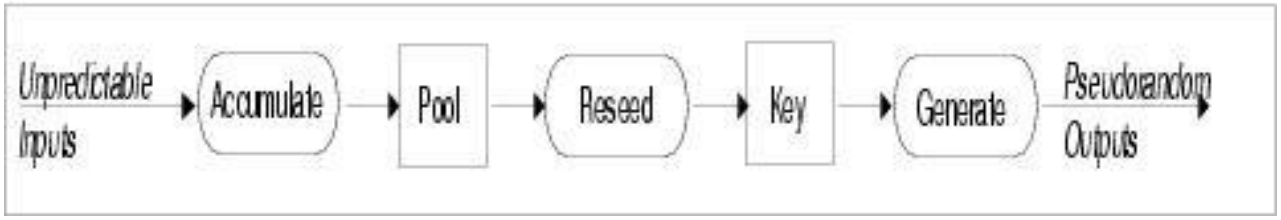
Kedua adalah *pseudo-random number generators* (PRNG). Angka yang benar-benar acak (seperti pada TRNG) cukup sulit (atau setidaknya, cukup repot) untuk dibangkitkan, terutama pada komputer yang didesain deterministik. Akhirnya, bilangan acak yang dihasilkan bersifat semu (*pseudo-random*).

Ketiga adalah *cryptographically secure* PRNG (CSPRNG). CSPRNG adalah PRNG yang telah didesain secara khusus untuk bisa tahan pada serangan kriptografik. Sebagai gambaran, jika diberikan sebanyak k bit yang dibangkitkan oleh sebuah CSPRNG, secara komputasi tidak mungkin untuk memprediksi bit ke $(k + 1)$. Lebih lanjut lagi, walaupun seluruh atau sebagian dari *state* pada CSPRNG ditunjukkan, akan tidak mungkin untuk menebak angka yang telah dibangkitkan sebelumnya.

II. PSEUDO-RANDOM NUMBER GENERATOR

Berikut adalah pemaparan lebih lanjut mengenai hal-hal yang perlu diperhatikan dalam merancang sebuah PRNG.

- ***Entropy Accumulation***
  Ini adalah proses di mana PRNG menghasilkan sebuah *internal state* baru yang tidak dapat ditebak. Selama inisialisasi atau reseeding, keberhasilan mengakumulasi entropi dari sampel cukup penting. adalah hal yang penting untuk mengestimasi secara tepat jumlah entropi yang telah dikumpulkan setiap saat untuk menghindarkan serangan berisi tebakan secara iterasi tapi tetap melakukan *reseed*.
- ***Generating Pseudorandom Outputs***
  Mekanisme pembangkitan menghasilkan *output* dari PRNG.
  Mekanisme ini harus mengikuti hal-hal berikut.



Gambar 1. Diagram blok dari Algoritma Yarrow

1. Aman terhadap serangan kriptanalisis
2. Efisien
3. Aman terhadap backtracking melalui kunci
4. Mampu untuk membangkitkan *output* yang cukup panjang secara aman tanpa melalui proses *reseeding*

- ***Seed File Managing***
  Selama inisialisasi, sebuah *seed file* akan menyediakan *seed* (benih) kepada *generator*. Seed ini memungkinkan generator untuk menghasilkan data acak, bahkan sebelum entropi yang dikumpulkan cukup untuk *generator*. *Seed file* ini dibaca pada saat *start-up*.

III. ALGORITMA YARROW

Berikut adalah penjelasn mengenai algoritma Yarrow secara umum.

Dalam algoritma ini digunakan block chiper dan fungsi hash. Jika kedua algoritma ini *secure* (aman), dan PRNG mendapatkan entropi untuk memulai yang cukup, maka kita akan memperoleh PRNG yang cukup kuat.

Dibutuhkan 2 (dua) algoritma, yaitu:

- sebuah fungsi hash satu arah, $h(x)$, yang memiliki *output* berukuran m bit
- block chiper, $E()$, berukuran n bit dengan kunci berukuran k bit

Diasumsikan bahwa fungsi hash tersebut:

- *collision intractable*
- satu arah
- diberikan himpunan kemungkinan nilai *input* (M)

Diasumsikan bahwa *block chiper* tersebut:

- aman dari serangan *known-plaintext* dan *chosen-plaintext*
- *output*-nya baik secara statistik

Maksimum jumlah output dari hasil PRNG ini mencapai $\min(m, k)$.

A.Mekanisme Pembangkitan

Gambar 2 menunjukkan mekanisme pembangkitan berdasarkan penggunaan *block chiper* dengan menggunakan *counter*.

Misalkan, C adalah *counter* berukuran n bit. Untuk membangkitkan blok *output* selanjutnya (dengan ukuran n bit), digunakan kunci K. Berikut adalah proses pembangkitannya.

$$C \leftarrow (C + 1)\ mod\ 2^n$$
$$R \leftarrow E_K(C)$$

R adalah blok *output* selanjutnya dan K adalah kunci PRNG pada saat itu.

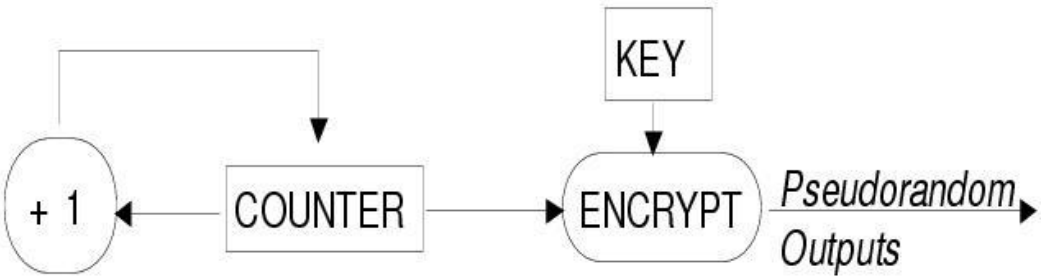
Jika sebuah kunci diperoleh pada suatu waktu, maka PRNG tidak akan mengeluarkan *output* lama sebelum kunci tersebut diperoleh. Oleh karena itu, jumlah blok yang dihasilkan tetap dihitung. Ketika dicapai suatu limit P_g (parameter keamanan sistem, $1 \le P_g \le 2^{n/3}$), dilakukan pembangkitan sebuah *output* PRNG berukuran k bit yang akan digunakan sebagai kunci baru.

$$K \leftarrow \text{Next } k \text{ bits of PRNG output}$$

Operasi seperti ini disebut *generator gate*. Perhatikan bahwa ini bukan operasi *reseeding* karena tidak adanya entropi baru.

Sebagaimana yang telah disebutkan sebelumnya, bahwa jumlah maksimum output dari generator di antara proses reseeding dibatasi hingga $\min(2^n, 2^{k/3}P_g)$ buah blok *output* dengan ukuran masing-masing n bit.

Pada praktiknya, nilai P_g harus diatur menjadi lebih kecil dari P_g sendiri, seperti $P_g - 10$, untuk meminimalkan banyaknya *output* yang bisa dipelajari untuk *backtracking*.



Gambar 2. Mekanisme Pembangkitan pada Algoritma Yarrow

B.Entropy Accumulator

Untuk mendapatkan entropi dari rangkaian masukan, maka kita mengkonkatenasikan seluruh *input*. Sebagai alternatif, dapat diambil sampel dari setiap sumber untuk setiap *pool*.

C. Reseed Mechanism

Mekanisme *reseeding* membangkitkan sebuah kunci baru *K* untuk pembangkit dari *entropy accumulator's pool* dan kunci saat itu. Waktu ekesekusi dari mekanisme reseeding ini bergantung pada parameter $P_t \geq 0$. Parameter ini bisa difiksasi pada implementasinya atau diatur menjadi dinamis.

Proses *reseed* terdiri dari langkah-langkah berikut.

1. *Entropy accumulator* menghitung hash dari hasil konkatenasi dari semua input ke dalam *fast pool*. Selanjutnya, hasilnya kita sebut v_0.
2. $v_i \leftarrow h(v_{i-1}|v_0|i)\ for\ i = 1,..,t$
3. $K \leftarrow h'(h(v_{P_t}|K), k)$
4. $C \leftarrow E_K(0)$
5. Reset semua entropi menjadi nol.
6. Hapus memori dari semua nilai antara.
7. Jika saat ini

Langkah 1 mengumpulkan *output* dari *entropy accumulator*. Langkah 2 menggunakan rumus iteratif panjang P_t *reseeding* yang membuat komputasi mahal jika memang diperlukan. Langkah 3 hash menggunakan fungsi hash *h* dan fungsi *h'*, yang akan segera didefinisikan untuk membuat kunci baru *K* dari kunci yang sudah ada dan nilai entropi baru v_{P_t}.
Langkah 4 mendefinisikan nilai baru dari *counter C*.

Fungsi *h'* didefinisikan dalam hal *h*. Untuk menghitung $h'(m,k)$ kita membangun:

$$s_0 := m$$
$$s_i := h(s_0|\ldots|s_{i-1}) \qquad i = 1,\ldots$$
$$h'(m,k) := \text{first } k \text{ bits of } (s_0|s_1|\ldots)$$

Ini adalah sebuah fungsi e ectively `ukuran adaptor Bahwa converts masukan dari setiap panjang ke output dari specied panjang. Jika input lebih besar daripada yang dikehendaki keluar menaruh, fungsi mengambil bit input terkemuka. Jika input merupakan ukuran yang sama dengan output fungsi ini fungsi identitas. Jika input lebih kecil daripada output bit ekstra yang dihasilkan dengan menggunakan hash fungsi. Ini adalah jenis yang sangat mahal PRNG, tetapi untuk ukuran kecil kita menggunakan ini tidak menjadi masalah. Tidak ada alasan keamanan mengapa kita akan menetapkan baru nilai bagi *counter C* ini dilakukan untuk memungkinkan lebih implementasi dan masih mempertahankan compatibility antara berbeda implementasi. Menetapkan C loket memudahkan pelaksanaan untuk menghasilkan seluruh bu? eh output dari generator sekaligus. Jika reseed terjadi, keluaran baru harus berasal dari biji baru dan bukan dari keluaran lama. Menetapkan nilai C baru membuat ini sederhana: apapun data dalam output bu? eh hanya dibuang. Hanya menggunakan kembali nilai counter yang ada tidak kompatibel sebagai Di implementasi? erent telah di? erent ukuran? Keluar menempatkan bu ers,? dan dengan demikian counter telah maju ke Di? erent poin. Rewinding konter ke virtual `Berjalan posisi rawan kesalahan. Untuk reseed kolam lambat, kita makan hash yang lambat kolam ke dalam kolam cepat, dan kemudian melakukan sebuah reseed. Dalam general, ini reseed lambat seharusnya t P ditetapkan sebagai tinggi seperti ditoleransi.

D. Reseed Control

Modul kontrol reseed menentukan waktu reseed adalah untuk dilakukan. Sebuah reseed eksplisit terjadi ketika beberapa aplikasi secara eksplisit meminta operasi reseed. Hal ini dimaksudkan untuk digunakan hanya jarang, dan hanya dengan aplikasi yang menghasilkan bernilai sangat tinggi rahasia acak. Akses ke fungsi reseed eksplisit harus dibatasi dalam banyak kasus. reseed secara berkala terjadi secara otomatis. Kolam cepat digunakan untuk reseed setiap kali ada sumber perusahaan memiliki perkiraan entropi lebih dari beberapa nilai threshold. Kolam lambat digunakan untuk reseed setiap kali setidaknya dua dari sumber perusahaan memiliki perkiraan entropi di atas beberapa nilai ambang lain..

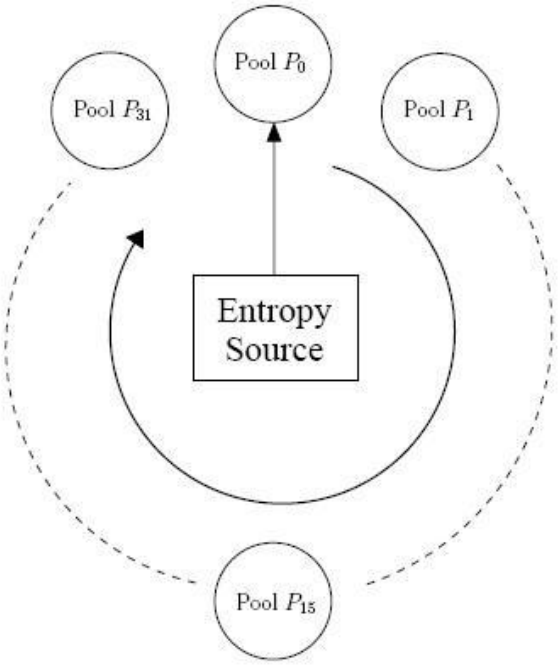
IV. ALGORITMA FORTUNA

Algoritma *Fortuna* merupakan salah satu algoritma pembangkit bilangan acak semu (*Pseudo-Random Number Generator/PRNG*). Algoritma ini pertama kali dipublikasikan oleh Niels Ferguson dan Bruce Schneier dalam bukunya *Practical Cryptography*.
Nama Fortuna diambil dari nama dewi romawi, yaitu Dewi keberuntungan dan kesempatan.

Pada dasarnya, Fortuna terbagi ke dalam tiga bagian besar, yaitu: *entropy accumulator*, *random number generator* (pembangkit bilangan acak), dan sistem untuk *seed file management*.

A.Entropy Accumulator

Akumulator Fortuna mengumpulkan data benar-benar secara acak dari berbagai sumber entropi eksternal, dan menggunakannya untuk *seed* (dan *reseed*) dari pembangkit. Algoritmanya memungkinkan penggunaan sampai dengan 256 sumber entropi. Fortuna dirancang sehingga yang sistem akan tetap aman jika seorang penyerang mendapat kontrol, tapi tidak semua kontrol, dari entropi sumber. Ketahanan ini dicapai melalui penggunaan *pool* entropi, seperti yang diilustrasikan pada Gambar 3.



Gambar 3. Pool pada Fortuna

Kondisi acak dari sumber eksternal secara seragam dan siklis didistribusikan ke 32 *pool*, yang masing-masing dilabeli P_0, P_1,. . . , P_{31}. Selanjutnya, setiap kolam dapat berisi data acak (yang secara teori) berukuran tak terbatas. Untuk mengatasi ini masalah penampungan, data dalam *pool* tertentu dikompresi secara bertahap, setiap kali sebuah *event* ditambahkan ke dalam *pool* itu. Fungsi kompresi yang digunakan adalah fungsi hash SHA-256, sehingga menjaga ukuran kolam tetap konstan pada 32 *byte*.

Ketika *pool* P_0 kolam telah 'cukup' terakumulasi dengan data-data acak, pembangkit dapat di-*reseed*. Sebuah *counter r* akan melacak berapa kali generator telah di-*reseed* dari *pool*. *Counter* ini menentukan *pool* mana yang akan digunakan pada *reseed* saat itu (sebut saja P_i) yang akan disertakan dalam *reseed* jika 2^i habis membagi *r*. Oleh karena itu, P_0 disertakan dalam setiap reseed, P_1 di setiap *reseed* kedua, dst. Akibatnya, sedangkan nomor yang lebih tinggi kolam berkontribusi lebih jarang untuk reseedings, mereka. Meskipun demikian, mengumpulkan sejumlah besar entropi antara reseedings. Reseeding dilakukan oleh hashing kolam entropi yang ditentukan bersama-sama, menggunakan dua iterasi dari SHA-256, selanjutnya ditandai SHA_d-256, di mana *d* menunjukkan *'double*'.

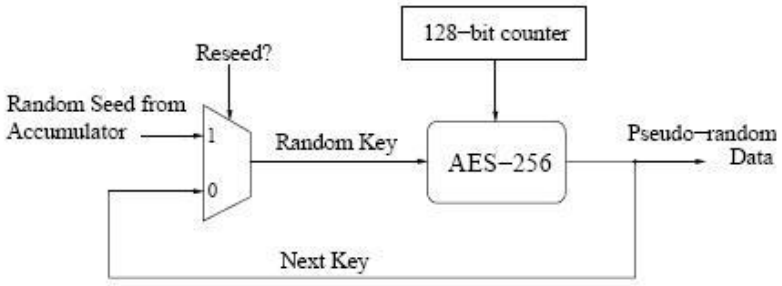
Sekali kolam telah digunakan dalam reseed, maka kemudian diatur ulang ke nol. Asalkan ada setidaknya satu sumber entropi dimana seorang penyerang tidak memiliki kendali, dia tidak akan dapat memprediksi isi minimal satu kolam, dan karena itu tidak akan mampu memecahkan RNG dengan cara ini.

B.Pembangkit

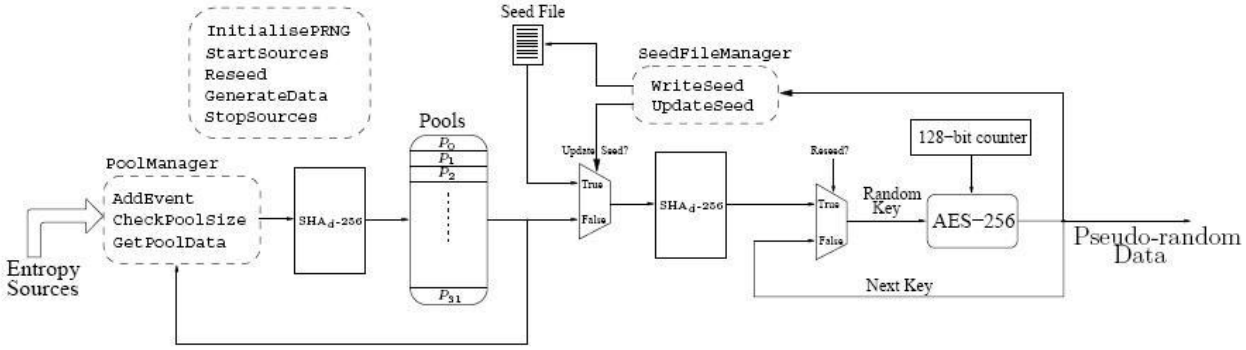
Generator mengambil ukuran yang tetap benih secara acak dari akumulator entropi dan menghasilkan sewenang-wenang panjang urutan data pseudo-random. Generator terdiri dari cipher blok di counter enkripsi modus, seperti yang ditunjukkan pada Gambar 4. Input plaintext ke cipher blok hanya sebuah counter, dan kunci 256-bit datang dari akumulator. Karena AES adalah 128-bit blok cipher, itu menghasilkan 16 byte data pada satu waktu. Sekali telah dihasilkan data ini, ia menghasilkan dua lebih lanjut blok, yang akan digunakan sebagai kunci baru berikutnya waktu yang diperlukan untuk menghasilkan data. Ini memenuhi persyaratan bahwa pengetahuan tentang arus keadaan generator tidak mengungkapkan data masa lalu dihasilkan. Jika generator adalah untuk menghasilkan sangat panjang urutan data acak dalam satu panggilan, data akan memiliki periode 2^{128}. Sejak periodik data tidak aman, data yang dihasilkan terbatas sampai 2^{20} byte per permintaan. Generator lalu akan berubah tombol, atau memeriksa *reseed*, sebelum melanjutkan.

C.Seed File Manager

Setelah inisialisasi Fortuna, sebuah "*seed file*" menyediakan *seed* untuk pembangkit. Hal ini memungkinkan *seed* awal generator untuk menghasilkan data acak, bahkan sebelum entropi dikumpulkan oleh *accumulator*. File seed dibaca pada saat *start-up*, dan *seed* baru segera dibangkitkan dan ditulis pada *file*. Sementara Fortuna mengakumulasi entropi, data ini digunakan untuk menciptakan *seed* dengan kualitas yang lebih baik. Direkomendasikan untuk menghasilkan *seed file* baru kira-kira setiap sepuluh menit, tapi ini benar-benar tergantung pada aplikasi, dan pada tingkat akumulasi entropi.



Gambar 4. Pembangkit pada Fortuna



Gambar 5. Contoh pengimplementasian Fortuna

V. ANALISIS PERBANDINGAN ANTARA YARROW DENGAN FORTUNA

Yarrow hanya terbatas pada paling besar 160 bit sesuai dengan ukuran entropy accumulation pool. Tiga-kunci triple-DES telah dikenal dapat melakukan serangan jauh lebih baik dari brute-force, namun perubahan mekanisme pencegahan backtracking cukup sering kunci yang cipher masih memiliki sekitar 160 bit keamanan dalam praktek. Pada beberapa titik di masa depan, kita mengharapkan untuk melihat standar blok cipher baru, AES. desain dasar Yarrow dengan mudah dapat mengakomodasi cipher blok baru. Namun, kita juga harus baik perubahan fungsi hash, atau datang dengan beberapa konstruksi fungsi hash khusus untuk menyediakan lebih dari 160 bit kolam entropi. Untuk AES dengan 128 bit, ini tidak akan menjadi masalah, karena dengan AES 192 bit atau 256 bit, itu harus ditangani. Kami mencatat bahwa kerangka Yarrow generik akan mengakomodasi blok cipher AES dan fungsi hash 256-bit (mungkin dibangun dari blok cipher AES) dengan tidak ada masalah.

Pada praktiknya, kelemahan-kelemahan Yarrow banyak muncul karena buruknya estimasi terhadap entropinya.

Jika dibandingkan dengan Fortuna, maka diperoleh bahwa perbedaan pada penanganan *entropy accumulator*-nya. Pada Yarrow, setiap sumber entropi harus digabungkan dengan mekanisme untuk mengestimasi entropi sebenarnya yang disuplai. Yarrow juga hanya memakai 2 (dua) pool. Selain itu, Yarrow lebih cocok menggunakan SHA-1 daripada SHA-256.

Algoritma Fortuna menggunakan data yang benar-benar acak yang dikumpulkan dari sumber-sumber entropi untuk membuatnya dalam kondisi yang tidak deterministik. Proses yang bersangkutan adalah yang disebut operasi "reseeding".

Bagaimanapun juga, algoritma Fortuna menjalankan operasi reseeding setelah diperoleh sebuah angka *fix* dari iterasi. Pengetesan menunjukkan bahwa kualitas dari pembangkitan bilangan acak meningkat secara signifikan dengan adanya peningkatan pada frekuensi dari operasi reseeding. Semakin tinggi frekuensi reseeding ini maka semakin banyak data acak dari sumber entropi. Ini menunjukkan adanya kebutuhan sebuah algoritma yang mengadopsi frekuensi dari operasi reseeding berdasarkan pada jumlah data yang dikumpulkan dari lingkungan sekitar.

Pernyataan sebelumnya menunjukkan bahwa sebenarnya algoritma Fortuna sendiri masih bisa dikembangkan.

Salah satu pengembangan yang pernah dilakukan adalah dengan pendekatan *fuzzy*, di mana peningkatan performa dalam menghasilkan bilangan acak dilakukan dengan merekayasa operasi reseeding yang dilakukan. Rekayasa ini bisa dilakukan dengan menambah frekuensinya. Dampaknya telah dinyatakan pada penjelasan sebelumnya.

V. KESIMPULAN

- Pembangkitan bilangan acak sangat banyak digunakan pada kriptografi dan memgang peranan cukup vital.
- Ada tiga macam pembangkitan bilangan acak, yaitu TRNG, PRNG, dan CSPRNG.
- Secara umum, ada tiga garis besar pada PRNG, yaitu *entropy accumulation*, *generating pseudo-random output*, dan *seed file managing*.
- Contoh dari PRNG adalah algoritma Yarrow. Selain itu ada juga algoritma Fortuna yang merupakan pengembangan dari algoritma Yarrow.
- Perbedaan antara Yarrow dan Fortuna terletak pada penangan *entropy accumulator*-nya. Yarrow menggunakan estimasi entropi yang sebenarnya yang disuplai. Selain itu, Yarrow menggunakan SHA-1 daripada SHA-256. Fortuna menggunakan SHA-256.
- Pada dasarnya Fortuna masih dapat dikembangkan. Salah satu pengembangan yang dilakukan adalah dengan menggunakan pendekatan *fuzzy*, yang merekayasa frekuensi operasi *reseeding*-nya.

REFERENSI

[1] J. Kelsey, B. Scheier, and N. Ferguson, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator"

[2] S. W. Putri, "Pembangkitan Nilai MAC dengan Menggunakan Algoritma Blowfish, Fortuna, dan SHA-256 (MAC-BF256)", 2007

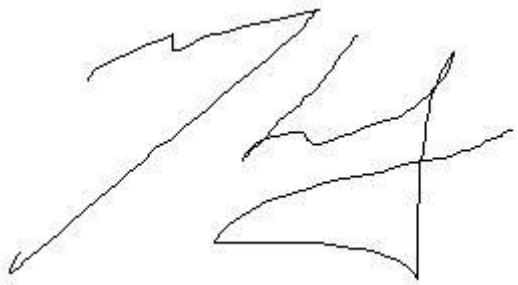
[3] R. McEvoy, J. Curan, P. Cotter, and C. Murphy, "Fortuna: Cryptographically Secure Pseudo-Random Number Generation In Software And Hardware"

[4] M. A. Akbar and M. Z. Khalid, "Fuzz-Fortuna: A fuzzified approach to generation of cryptographically secure Pseudo-random numbers"

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 17 Mei 2010



M. Haekal Izmanda Pulungan
135 07 020