STUDI MENGENAI *CRYPTOGRAPHICALLY SECURE PSEUDORANDOM NUMBER GENERATOR* TERMASUK TIGA ALGORITMA DI DALAMNYA: BLUM BLUM SHUB, FORTUNA, DAN YARROW

Victor – NIM : 13503001

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if13001@students.if.itb.ac.id

Abstrak

Makalah ini membahas mengenai *cryptographically secure pseudo-random number generator* (CSPRNG), yaitu sebuah *pseudo-random number generator* (PRNG) dengan kelengkapan yang membuatnya cocok untuk digunakan dalam bidang kriptografi. Banyak aspek dari kriptografi yang memerlukan angka-angka *random* seperti untuk kepentingan penciptaan kunci, *nonces*, *salt* untuk skema *signature*, atau *one-time pads*. "Kualitas" dari suatu sifat *random* yang diperlukan untuk aplikasi-aplikasi di atas amat beragam. Sebagai contoh untuk menciptakan sebuah *nonce* pada beberapa protokol hanya dibutuhkan suatu keunikan. Lain halnya dengan penciptaan kunci *master* yang memerlukan kualitas yang lebih tinggi, seperti misalnya *entropy* yang lebih tinggi. Dan untuk kasus *one-time pads*, kerahasiaan sempurna atas informasi baru diperoleh jika kunci dihasilkan oleh sumber dengan fungsi *random* yang memiliki *entropy* yang tinggi.

Algoritma yang ada dalam CSPRNG antara lain adalah Blum Blum Shub, Fortuna, dan Yarrow. Dalam makalah ini akan dibahas masing-masing algoritma di atas dengan sedikit perbandingan yang ada.

Kata kunci: cryptographically secure pseudo-random number generator, pseudo-random number generator, entropy, *blum blum shub*, *fortuna*, *yarrow*.

1. Pendahuluan

Kebutuhan dari sebuah PRNG juga dipenuhi oleh sebuah CSPRNG, tapi tidak sebaliknya. Kebutuhan CSPRNG dapat dibagi menjadi dua kelompok: pertama, adalah bahwa properti statistik yang dimilikinya bagus (berhasil melewati tes *statistical randomness*); dan kedua, bahwa mereka bertahan dengan baik terhadap serangan yang serius, bahkan ketika sebagian dari status awal atau status *running* mereka diperoleh oleh penyerang. Setiap CSPRNG harus memenuhi "*next-bit test*". Tes tersebut adalah sebagai berikut: Diberikan sebuah bit ke-*k* dari sebuah *random sequence*, tidak boleh ada algoritma *polynomial-time* yang dapat memprediksi bit ke (*k*+1) dengan probabilitas keberhasilan lebih besar dari 50%. Pada tahun 1982 Andrew Yao membuktikan bahwa jika sebuah *generator* melewati tes ini maka ia juga melewati seluruh tes statistik *polynomial-time* untuk *randomness*. Setiap CSPRNG harus melalui '*state compromise extentions*'. Pada kondisi di mana sebagian atau seluruh status telah diketahui (atau ditebak dengan benar), haruslah tidak mungkin untuk membangun kembali urutan angka *random* yang sama berdasarkan apa yang telah diketahui tadi. Sebagai tambahan, jika ada masukkan *entropy* ketika proses sedang berjalan, haruslah tidak mungkin untuk menggunakan pengetahuan atas status masukkan untuk memprediksi kondisi status CSPRNG yang akan datang.

Kebanyakan PRNG tidak sesuai untuk digunakan sebagai CSPRNG dan akan gagal di kedua tes. Pertama, semenjak banyak keluaran PRNG tampak acak untuk dapat melewati tes statistik, mereka tidak tahan terhadap *determined reverse engineering*. Tes statistik khusus akan dapat menemukan bahwa angka acak yang dihasilkan oleh sebuah PRNG ternyata tidak sepenuhnya acak. Kedua, untuk kebanyakan PRNG, ketika status mereka telah diketahui, semua angka acak yang akan datang dapat diprediksikan dengan mudah. CSPRNG dirancang secara eksplisit untuk tahan terhadap jenis *cryptanalysis* ini.

Santha dan Vazirani membuktikan bahwa beberapa *bit streams* dengan tingkat acak yang lemah dapat dikombinasikan untuk menghasilkan *higher-quality quasi-random bit stream*. Bahkan sebelumnya, John von Neumann telah berhasil membuktikan bahwa

sebuah algoritma yang sederhana dapat menghilangkan sejumlah kecenderungan tertentu dalam *bit stream* apapun yang seharusnya diaplikasikan pada tiap *bit stream* sebelum menggunakan variasi apapun dari rancangan Santha-Vazirani. Field ini dinamakan *entropi extraction* dan adalah subjek dari sebuah penelitian aktif.

Rancangan CSPRNG dapat dibagi menjadi tiga kelas [4]:

1. Yang berdasar pada *block chipers / cryptographic primitives*.
2. Yang berdasar pada permasalahan matematis yang 'rumit'.
3. Yang berdasar pada rancangan khusus.

Pada kelas rancangan yang berdasar pada *cyrptographic primitives*, beberapa hal berikut patut diperhatikan:

- Sebuah *secure block chiper* dapat dikonversikan menjadi sebuah CSPRNG dengan cara menjalankannya dalam *counter mode*. Hal ini dilakukan dengan memilih sebuah kunci acak dan mengenkripsi sebuah 0, lalu mengenkripsi sebuah 1, lalu mengenkripsi sebuah 2, dan seterusnya. *Counter* juga dapat dimulai pada angka *arbitrary* selain 0. Dengan jelas dapat diketahui, periode akan berjumlah 2^n untuk sebuah *n*-bit *block chiper*; selain itu, nilai awal (misalnya, kunci dan "*plaintext*") tidak boleh diketahui oleh si penyerang atau semua tingkat keamanan akan hilang.
- Dalam beberapa kasus, sebuah nilai hash yang aman secara kriptografis akan nilai *counter* juga dapat bertindak sebagai sebuah CSPRNG yang baik. Pada kasus ini, adalah penting bahwa nilai awal dari counter ini berupa nilai acak dan rahasia. Jika *counter* adalah *bignum*, maka CSRPNG bisa saja memiliki periode yang tidak terbatas. Bagaimanapun juga, telah ada sedikit pembelajaran terhadap algoritma-algoritma untuk digunakan dalam hal ini, dan setidaknya beberapa penulis memperingatkan agar tidak mempraktekkan penggunaan ini (Young and Yung, Malicious Cryptography, Wiley, 2004, sec 3.2).
- Kebanyakan *stream chipers* bekerja dengan cara menciptakan sebuah pseudorandom stream of bits yang dikombinasikan (hampir selalu menggunakan XOR) dengan *plaintext*; terkadang *stream* ini dapat digunakan sebagai sebuah CSPRNG yang baik.

Satu rancangan dari kelas ini diikutsertakan dalam standar ANSI X9.17 (*Financial Institution Key Management (wholesale)*), dan telah diadopsi sebagai sebuah standar FIPS. Rancangan tersebut bekerja sebagai berikut:

- Input: formasi tanggal/waktu *D* (dalam resolusi maksimal yang dimungkinkan), sebuah 64 bit *random seed s*, dan sebuah kunci DES EDE *k*.
- Komputasi: I = DES_k (D)
- Output: tiap kali sebuah angka acak diperlukan, berikan keluaran x=DES_k(I xor *s*), dan *update* nilai *seed s* menjadi DES_k(x xor I).

Telah diusulkan bahwa algoritma ini dapat ditingkatkan performansinya dengan menggunakan AES menggantikan DES (Young and Yung, op cit, sect 3.5.1).

Pada kelas rancangan theorema angka, Blum Blum Shub memiliki bukti keamanan yang kuat, walau berdasar pada kondisi tertentu, berbasiskan pada kerumitan *integer factorization*. Walau demikian, implementasi terbilang lambat jika dibandingkan dengan beberapa rancangan lainnya.

Pada kelas rancangan khusus, terdapat beberapa praktek PRNG yang telah dirancangan agar aman secara kriptografis, termasuk:

- Algoritma Yarrow yang berusaha mengevaluasi kualitas entropy dari masukkan, dan versi terbarunya, Fortuna, yang tidak melalukan evaluasi tersebut.
- Fungsi dari Microsoft's Cryptographic Application Programming Interface: CryptGenRandom.
- ISAAC berbasiskan varian dari RC4 chiper.

2. Tes Statistik

Berdasarkan NIST Special Publication 800-22, A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications [9], berikut adalah cara melakukan testing.

Beberapa tes statistik dapat diaplikasikan kepada sebuah *sequence* dengan tujuan untuk membandingkan dan mengevaluasi apakah *sequence* tersebut adalah sebuah *sequence* yang acak sebenarnya. Tingkat acak adalah sebuah properti dari probabilitas; maka dari

itu, properti dari sebuah *sequence* acak dapat dikarakteristikkan dan dijabarkan dalam konteks probabilitas. Hasil yang kemungkinan besar keluar dari tes statistik, ketika diaplikasikan pada sebuah *sequence* acak sebenarnya, diketahui sebagai sebuah priori dan dapat dijabarkan dalam konteks probabilitas. Terdapat tes statistik yang tak terbatas jumlahnya yang mungkin, masing-masing mengakses ada atau tidak adanya sebuah "pola" yang, jika dideteksi ada, akan mengindikasikan bahwa *sequence* tersebut tidak acak. Karena begitu banyak tes untuk menilai apakah sebuah *sequence* acak atau tidak, tidak ada sejumlah tes terbatas yang dikatakan "lengkap". Sebagai tambahan, hasil dari tes statistik harus diinterpretasikan dengan ketelitian dan kewaspadaan tinggi untuk menghindari kesimpulan yang salah tentang sebuah *generator* tertentu.

Sebuah tes statistik diformulasikan untuk mengetes sebuah *null hypotetis* (H_0) tertentu. Terkait dengan *null hypotetis* ini terdapat *alternative hypotesis* (H_a). Untuk tiap tes yang diaplikasikan, sebuah keputusan atau kesimpulan ditarik yang menerima atau menolak *null hypotetis*, misalnya: apakah *generator* memproduksi (atau tidak memproduksi) nilai-nilai acak, berdasarkan *sequence* yang diproduksinya.

Untuk masing-masing tes, sebuah statistik tingkat acak yang relevan harus dipilih dan digunakan untuk menentukan penerimaan atau penolakan *null hypotetis*. Di bawah asumsi dari tingkat acak, statistik seperti itu memiliki nilai-nilai yang mungkin terdistribusi. Sebuah teori referensi distribusi dari statistik di bawah *null hypotetis* ini ditentukan oleh metode matematis. Dari referensi distribusi ini, sebuah **nilai kritis** ditentukan (pada umumnya, nilai ini jauh terletak di ekor distribusi, katakanlah sejauh 99%). Selama tes dilakukan, sebuah nilai tes statistik dikomputasikan pada data (*sequence* yang sedang dites). Nilai tes statistik ini dibandingkan dengan nilai kritis. Jika nilai tes statistik melebihi nilai kritis, *null hypotetis* untuk tingkat acak ditolak. Sebaliknya, *null hypotetis* akan diterima jika nilai tes statistik kurang dari nilai kritis.

Dalam praktek, alasan bahwa tes hipotesis statistik berjalan adalah karena referensi distribusi dan nilai kritis bergantung pada dan diciptakan di bawah asumsi tentative atas tingkat acak. Jika asumsi tingkat acak adalah, secara fakta, benar untuk data yang ada, maka nilai tes statistik yang dihasilkan akan memiliki tingkat probabilistik yang amat rendah (misal 0,01%) untuk dapat melebihi nilai kritis.

Di lain pihak, jika nilai tes statistik yang dikalkulasi *memang* melebihi nilai kritis (misal jika kondisi di mana even dengan probabilitas kecil ini muncul), maka dilihat dari sudut pandang hipotesis statistik, even dengan nilai probabilistik kecil tersebut tidak seharusnya muncul secara natural. Maka dari itu, ketika nilai tes statistik melebihi nilai kritis, kesimpulan dibuat bahwa asumsi awal atas tingkat acak adalah salah. Dalam kasus ini, tes hipotesis statistik menyandang kesimpulan berikut: tolak H_0 (tingkat acak) dan terima H_a (tingkat tak acak).

Tes hipotesis statistik adalah sebuah prosedur penghasil-kesimpulan yang memiliki dua kemungkinan keluaran, apakah itu menerima H_0 (data adalah acak) atau menerima H_a (data tak acak). Tabel 2x2 di bawah ini (lihat **Tabel 1**) mengkaitkan status sebenarnya (*unknown*) dari data yang ada dengan kesimpulan yang datang ketika menggunakan prosedur tes.

Jika data, pada kenyataannya, acak, maka kesimpulan untuk menolak *null hypotesis* (yang mengindikasikan bahwa data tidak acak) akan muncul dengan persentasi yang kecil. Kesimpulan ini dinamakan *Type I error*. Jika data, pada kenyataannya, tidak acak, maka kesimpulan untuk menerima *null hypotesis* (yang mengindikasikan bahwa data adalah acak) dinamakan *Type II error*. Kesimpulan untuk menerima H_0 ketika data adalah acak sebenarnya, dan untuk menolak H_0 ketika data tidak acak, adalah benar.

Probabilitas dari *Type I error* sering dinamakan **tingkat signifikansi** dari tes. Probabilitas ini dapat diatur dan diperuntukkan untuk tes dan didenotasikan sebagai α. Untuk tes, α adalah probabilitas tes akan mengindikasikan bahwa sebuah *sequence* tidak acak jika sesungguhnya *sequence* itu acak. Yakni ketika sebuah *sequence* tampak tidak acak ketika dihasilkan oleh *generator* yang baik. Nilai umum dari sebuah α dalam kriptografi adalah sekitar 0,01.

Tabel 1 Tabel Status dan Kesimpulan

TRUE SITUATION	CONCLUSION	
	Accept H_0	Accept H_a (reject H_0)
Data is random (H_0 is true)	No error	Type I error
Data is not random (H_a is true)	Type II error	No error

Probabilitas dari *Type II error* didenotasikan dengan β. Untuk tes, β adalah probabilitas tes akan mengindikasikan bahwa sebuah *sequence* acak jika sesungguhnya *sequence* itu tidak acak; yakni ketika sebuah *generator* yang buruk menghasilkan sebuah sequence yang tampak acak. Tidak seperti α, β bukan merupakan nilai pasti. β dapat berupa nilai yang beragam karena terdapat sejumlah cara yang tidak terbatas untuk sebuah *data stream* dapat menjadi tidak acak, dan tiap cara yang berbeda memiliki nilai β yang berbeda. Kalkulasi dari *Type II error* β lebih sulit dilakukan daripada mengkalkulasi nilai α karena banyaknya kemungkinan atas tingkat tidak acak.

Satu dari tujuan utama tes adalah untuk meminimalkan probabilitas Type II error, misalnya untuk meminimalkan probabilitas diterimanya sebuah *sequence* yang dianggap diproduksi oleh *generator* yang baik padahal sebenarnya diproduksi *generator* buruk. Probabilitas α dan β terkait satu sama lain dan terkait juga dengan besar *n* dari *sequence* yang dites sedemikian hingga jika dua dari ketiga nilai tersebut diketahui, nilai yang ketiga dapat ditentukan secara otomatis. Para praktisi biasanya memilih sebuah sampel berukuran *n* dan sebuah nilai untuk α (probabilitas *Type I error* – tingkat signifikansi). Kemudian sebuah nilai kritis untuk statistik yang ada dipilih yang akan menghasilkan nilai β terkecil (probabilitas *Type II error*). Yakni, sebuah ukuran sampel yang sesuai dipilih berikut dengan probabilitas penerimaan untuk menentukan sebuah *generator* buruk telah memproduksi sebuah *sequence* yang sesungguhnya acak. Kemudian, nilai batas penerimaan dipilih sedemikian hingga probabilitas dari penerimaan yang salah atas sebuah *sequence* dianggap acak memiliki nilai sekecil mungkin.

Masing-masing tes berdasarkan pada nilai tes statistik yang sudah dikalkulasikan, yang merupakan fungsi dari data. Jika nilai tes statistik adalah *S* dan nilai kritis adalah *t*, maka probabilitas *Type I error* adalah $P(S > t \parallel H_0$ adalah true) = P(reject $H_0 \mid H_0$ adalah true), dan probabilitas Type II error adalah $P(S \leq t \parallel H_0$ adalah false) = P(terima $H_0 \mid H_0$ adalah false). Tes statistik digunakan untuk mengkalkulasi nilai *P* yang menyimpulkan kekuatan tingkat penolakan *null hypotesis*. Untuk tes-tes ini, masing-masing nilai *P* adalah probabilitas sebuah *generator* angka acak yang sempurna akan memproduksi sebuah *sequence* yang kurang acak dibandingkan *sequence* yang dites, jika diberikan semacam tingkat tidak acak oleh tes. Jika sebuah nilai *P* untuk sebuah tes ditentukan sama dengan 1, maka *sequence* akan tampak memiliki tingkat acak yang sempurna. Nilai *P* sama dengan satu mengindikasikan bahwa *sequence* tampak sama sekali tak acak. Sebuah tingkat signifikansi (α) dapat dipilih untuk tes-tes. Jika nilai $P \geq \alpha$, maka *null hypotesis* diterima; misalnya *sequence* tampak acak. Jika $P < \alpha$, maka *null hypotesis* ditolak, misalnya *sequence* tampak tidak acak. Parameter α menyatakan probabilitas *Type I error*. Pada dasarnya, α dipilih antara rentang [0,001 – 0,01].

Misalnya, sebuah α yang bernilai 0,001 mengindikasikan bahwa diharapkan ada 1 dari 1000 *sequence* yang ditolak oleh tes jika *sequence* adalah acak. Untuk sebuah nilai $P \geq$ 0,001, sebuah *sequence* dapat dianggap acak dengan nilai kepercayaan (*cofidence*) 99,9%. Untuk nilai $P <$ 0,001, sebuah *sequence* dapat dianggap tidak acak dengan nilai kepercayaan 99,9%.

3. Blum Blum Shub

Blum Blum Shub (BBS) adalah sebuah *pseudorandom number generator* yang dibuat pada tahun 1986 oleh Lenore Blum, Manuel Blum, dan Michael Shub (Blum et al, 1986) [1].

BBS mengambil bentuk sebagai berikut:

$$x_{n+1} = (x_n)^2 \bmod M$$

di mana $M = pq$ adalah hasil kali dari dua bilangan prima besar *p* dan *q*. Dalam tiap langkah dari algoritma ini, beberapa keluaran dihasilkan dari x_n; keluaran tersebut secara umum adalah pasangan bit dari x_n atau satu atau lebih bit yang signifikan dari x_n.

Kedua nilai prima, *p* dan *q*, harus kongruen dengan 3 (mod 4) (hal ini akan menjamin tiap *quadratic residue* memiliki satu akar kuadrat yang juga merupakan *quadratic residue*) dan gcd(φ(*p*-1), φ(*q*-1)) juga harus kecil (hal ini membuat panjang siklus menjadi besar).

Dalam matematika, *greatest common divisor* (gdc), terkadang dikenal juga dengan nama *greatest common factor* (gfc) atau *highest common factor* (hfc), dari dua buah bilangan integer tak nol, adalah bilangan integer positif terbesar yang dapat membagi habis kedua bilangan tadi. Nilai *greatest common divisor* dari *a* dan *b* ditulis dengan notasi gdc(*a*,*b*). Sebagai contoh, gdc(12,18) = 6, gdc(-4,14) = 2, dan gdc(5,0) = 5. Kedua bilangan disebut *coprime* atau *relatively prime* jika *greatest common divisor* yang dihasilkan oleh

keduanya bernilai satu. Sebagai contoh, 9 dan 28 adalah *relatively prime*.

Dalam teori bilangan, tolient φ(*n*) dari sebuah bilangan integer *n* didefinisikan sebagai jumlah integer positif yang kurang dari atau sama dengan *n* dan *coprime* dengan *n*. Misalnya, φ(8) = 4 semenjak empat bilangan 1, 3, 5, dan 7 adalah *coprime* dari 8. Totient biasa dipanggil Euler totient atau Euler's totient, setelah matematisian Leonhard Euler, yang mempelajarinya.

Karakteristik yang menarik dari BBS *generator* adalah adanya kemungkinan untuk menghitung nilai x_i secara langsung:

$$x_i = \left(x_0 {}^{2^i \bmod (p-1)(q-1)}\right) \bmod M$$

Generator ini tidak cocok digunakan untuk simulasi, hanya untuk kriptografi, karena tidak begitu cepat. Namun demikian, BBS memiliki bukti keamanan yang tidak lazim, yang mengkaitkan kualitas generator dengan kerumitan komputasi atas sebuah *integer factorization*. Ketika kedua bilangan prima dipilih dengan benar, dan O (log log *M*) *lower-order bits* dari tiap x_n adalah keluaran, maka dalam limit *M* yang bertambah besar, membedakan bit keluaran dari angka acak akan sesulit memfaktorisasikan *M*.

Jika *integer factorization* sudah rumit (seperti diketahui umum) maka BBS dengan *M* yang besar akan memiliki sebuah keluaran yang bebas dari pola tak acak apapun yang dapat dicari dengan sejumlah perhitungan. Hal ini membuat BBS seaman teknologi kriptografi lainnya yang berkaitan dengan masalah faktorisasi, seperti misalnya enkripsi RSA.

Contoh: Didapat *p=11*, *q=19*, dan *s=3*. Diharapkan akan menghasilkan panjang siklus yang besar untuk angka-angka kecil tersebut, karena gcd(φ(*p*-1), φ(*q*-1)) = 2. *Generator* mulai mengevaluasi s_0 dengan mengunakan x_{-1} = *s* dan menciptakan sequence x_0, x_1, x_2, … x_5 = 9, 81, 82, 36, 42, 92. Jika pasangan bit digunakan untuk menentukan keluaran, maka bit keluaran adalah 0 1 1 0 1 0.

Berikut adalah *source code* Blum Blum Shub [3] dalam bahasa Java.

```
package com.modp.random;
import java.util.Random;
import java.security.SecureRandom;
import java.math.BigInteger;

/**
 * The Blum-Blum-Shub random number generator.
 *
 * <p>
 * The Blum-Blum-Shub is a "cryptographically secure" random number
 * generator.  It has been proven that predicting the ouput
 * is equivalent to factoring <i>n</i>, a large integer generated
 * from two prime numbers.
 * </p>
 *
 * <p>
 * The Algorithm:
 * </p>
 * <ol>
 * <li>
 *  (setup) generate two secret prime numbers <i>p</i>, <i>q</i> such that
 *   <i>p</i> &ne; <i>q</i>, <i>p</i> &equiv; 3 mod 4, <i>q</i> &equiv; 3 mod 4.
 * </li>
 * <li> (setup) compute <i>n</i> = <i>pq</i>.  <i>n</i> can be re-used, but
 *   <i>p</i>, and <i>q</i> are secret and should be disposed of.</li>
 * <li> Generate a (secure) random seed <i>s</i> in the range [1, <i>n</i> -1]
 *   such that gcd(<i>s</i>, <i>n</i>) = 1.
 * <li> Compute <i>x</i> = <i>s</i><sup>2</sup> mod <i>n</i></li>
 * <li> Compute a single random bit with:
 *   <ol>
 *   <li> <i>x</i> = <i>x</i><sup>2</sup> mod <i>n</i></li>
 *   <li> return Least-Significant-Bit(<i>x</i>) (i.e. <i>x</i> & 1)</li>
 *   </ol>
 * Repeat as necessary.
 * </li>
 * </ol>
 *
 * <p>
 * The code originally appeared in <a href="http://modp.com/cida/"><i>Cryptography for
 * Internet and Database Applications </i>, Chapter 4, pages 174-177</a>
 * </p>
 *
 * @author Nick Galbreath -- nickg [at] modp [dot] com
 * @version 3 -- 06-Jul-2005
 *
 */
```

```
public class BlumBlumShub implements RandomGenerator {

	// pre-compute a few values
	private static final BigInteger two = BigInteger.valueOf(2L);

	private static final BigInteger three = BigInteger.valueOf(3L);

	private static final BigInteger four = BigInteger.valueOf(4L);

	/**
	 * main parameter
	 */
	private BigInteger n;

	private BigInteger state;

	/**
	 * Generate appropriate prime number for use in Blum-Blum-Shub.
	 *
	 * This generates the appropriate primes (p = 3 mod 4) needed to compute the
	 * "n-value" for Blum-Blum-Shub.
	 *
	 * @param bits Number of bits in prime
	 * @param rand A source of randomness
	 */
	private static BigInteger getPrime(int bits, Random rand) {
		BigInteger p;
		while (true) {
			p = new BigInteger(bits, 100, rand);
			if (p.mod(four).equals(three))
				break;
		}
		return p;
	}

	/**
	 * This generates the "n value" -- the multiplication of two equally sized
	 * random prime numbers -- for use in the Blum-Blum-Shub algorithm.
	 *
	 * @param bits
	 *            The number of bits of security
	 * @param rand
	 *            A random instance to aid in generating primes
	 * @return A BigInteger, the <i>n</i>.
	 */
	public static BigInteger generateN(int bits, Random rand) {
		BigInteger p = getPrime(bits/2, rand);
		BigInteger q = getPrime(bits/2, rand);

		// make sure p != q (almost always true, but just in case, check)
		while (p.equals(q)) {
			q = getPrime(bits, rand);
		}
		return p.multiply(q);
	}

	/**
	 * Constructor, specifing bits for <i>n</i>
	 *
	 * @param bits number of bits
	 */
	public BlumBlumShub(int bits) {
		this(bits, new Random());
	}

	/**
	 * Constructor, generates prime and seed
	 *
	 * @param bits
	 * @param rand
	 */
	public BlumBlumShub(int bits, Random rand) {
		this(generateN(bits, rand));
	}

	/**
	 * A constructor to specify the "n-value" to the Blum-Blum-Shub algorithm.
	 * The inital seed is computed using Java's internal "true" random number
	 * generator.
	 *
	 * @param n
	 *            The n-value.
	 */
	public BlumBlumShub(BigInteger n) {
		this(n, SecureRandom.getSeed(n.bitLength() / 8));
	}
```

```
	/**
	 * A constructor to specify both the n-value and the seed to the
	 * Blum-Blum-Shub algorithm.
	 *
	 * @param n
	 *            The n-value using a BigInteger
	 * @param seed
	 *            The seed value using a byte[] array.
	 */
	public BlumBlumShub(BigInteger n, byte[] seed) {
		this.n = n;
		setSeed(seed);
	}

	/**
	 * Sets or resets the seed value and internal state
	 *
	 * @param seedBytes
	 *            The new seed.
	 */
	public void setSeed(byte[] seedBytes) {
		// ADD: use hardwired default for n
		BigInteger seed = new BigInteger(1, seedBytes);
		state = seed.mod(n);
	}

	/**
	 * Returns up to numBit random bits
	 *
	 * @return int
	 */
	public int next(int numBits) {
		// TODO: find out how many LSB one can extract per cycle.
		//   it is more than one.
		int result = 0;
		for (int i = numBits; i != 0; --i) {
			state = state.modPow(two, n);
			result = (result << 1) | (state.testBit(0) == true ? 1 : 0);
		}
		return result;
	}

	/**
	 * A quickie test application for BlumBlumShub.
	 */
	public void main(String[] args) {
		// First use the internal, stock "true" random number
		// generator to get a "true random seed"
		SecureRandom r = new SecureRandom();
		System.out.println("Generating stock random seed");
		r.nextInt(); // need to do something for SR to be triggered.

		// Use this seed to generate a n-value for Blum-Blum-Shub
		// This value can be re-used if desired.
		System.out.println("Generating N");
		int bitsize = 512;
		BigInteger nval = BlumBlumShub.generateN(bitsize, r);

		// now get a seed
		byte[] seed = new byte[bitsize/8];
		r.nextBytes(seed);

		// now create an instance of BlumBlumShub
		BlumBlumShub bbs = new BlumBlumShub(nval, seed);

		// and do something
		System.out.println("Generating 10 bytes");
		for (int i = 0; i < 10; ++i) {
			System.out.println(bbs.next(8));
		}

		// OR
		// do everything almost automatically
		BlumBlumShub bbs2 = new BlumBlumShub(bitsize /*,+ optional random
instance */);

		// reuse a nval (it's ok to do this)
		BlumBlumShub bbs3 = new BlumBlumShub(nval);
	}
}
```

4. Yarrow

Algoritma Yarrow adalah sebuah *cryptographically secure pseudo-random number generator*. Nama ini diambil dari tanaman Yarrow, yang batangnya dikeringkan dan digunakan sebagai *randomising agent* pada zaman dinasti I Ching [10].

Dirancang oleh Bruce Schneier, John Kelsey, dan Niels Ferguson dari Counterpane Labs (Kelsey et al, 1999). Algoritma Yarrow secara eksplisit tidak dipantenkan dan bebas royalty, dengan kata lain tidak diperlukan izin apapun untuk menggunakannya. Yarrow digunakan dalam Mac OS X dan FreeBSD sebagai alat /dev/random mereka.

Dalam website resmi Yarrow (http://www.schneier.com/yarrow.html) [2] terdapat *source code* Yarrow dengan versi 0.8.71. Ada 6 *package* dalam *source* Yarrow yaitu:

1. entropyhooks – Rutin untuk koleksi entropy. Mengandung entropy DLL, yang menyediakan rutin *hook* untuk mengkoleksi waktu *mouse* dan *keyboard* sama seperti rutin komunikasi berdasarkan arsip-arsip peta-*memory* dan *event flags.*
2. prngcore – Rutin PRNG utama. Kini dalam bentuk DLL, dengan beberapa fungsi yang disediakan untuk *end user*, dan beberapa fungsi lainnya disediakan untuk *high level user*.
3. frontend – Koleksi entropy dan program setup prng. Aplikasi ini harus dijalankan pada saat *startup* dan pada sistem apapun yang ingin menjalankan rutin prng. Usaha untuk memanggil rutin prng tanpa menjalankan frontend terlebih dahulu akan berujung ke *error*.
4. testapp – Sebuah aplikasi trivial untuk menguji "*client end*" dari prng ("*server end*" adalah setup dan dikendalikan oleh frontend). Jalankan frontend lalu aplikasi ini untuk mendemonstrasikan setup Yarrow sepenuhnya.
5. smf – Secure Malloc/Free. Mengandung DLL untuk manajemen *memory*. *Memory* diambil dari *system paging file* lalu dipetakan ke dalam *address space* dari proses yang sedang berjalan. *Memory* tersebut kemudian dengan aman dihapus ketika *freeing*.
6. zlib – Versi terkompilasi dari *zlib compressiong library*. zlibr.lib adalah versi *release* dan zlibd.lib adalah versi *debug.*

Dari package prngcore terdapat beberapa arsip utama, salah satunya arsip yarrow.h yang berisi sebagai berikut:

```
/*
        yarrow.h
        Main header file for Counterpane's Yarrow Pseudo-random number generator.
*/

#ifndef YARROW_H
#define YARROW_H

/* Error Codes */
typedef enum prng_error_status {
         PRNG_SUCCESS = 0,
         PRNG_ERR_REINIT,
         PRNG_ERR_WRONG_CALLER,
         PRNG_ERR_NOT_READY,
         PRNG_ERR_NULL_POINTER,
         PRNG_ERR_LOW_MEMORY,
         PRNG_ERR_OUT_OF_BOUNDS,
         PRNG_ERR_COMPRESSION,
         PRNG_ERR_NOT_ENOUGH_ENTROPY,
         PRNG_ERR_MUTEX,
         PRNG_ERR_TIMEOUT,
         PRNG_ERR_PROGRAM_FLOW
} prng_error_status;

/* Declare YARROWAPI as __declspec(dllexport) before
   including this file in the actual DLL */
#ifndef YARROWAPI
#define YARROWAPI __declspec(dllimport)
#endif

/* Public function forward declarations */
YARROWAPI int prngOutput(BYTE *outbuf,UINT outbuflen);
YARROWAPI int prngStretch(BYTE *inbuf,UINT inbuflen,BYTE *outbuf,UINT outbuflen);
YARROWAPI int prngInput(BYTE *inbuf,UINT inbuflen,UINT poolnum,UINT estbits);
YARROWAPI int prngForceReseed(LONGLONG ticks);
YARROWAPI int prngAllowReseed(LONGLONG ticks);
YARROWAPI int prngProcessSeedBuffer(BYTE *buf,LONGLONG ticks);
YARROWAPI int prngSlowPoll(UINT pollsize);

#endif
```

Kemudian ada arsip prng.h dan prng.c yang merupakan inti dari algoritma Yarrow.

```
/*
        prng.h

        Main private header for the Counterpane PRNG. Use this to be able access the
        initialization and destruction routines from the DLL.
*/

#ifndef YARROW_PRNG_H
#define YARROW_PRNG_H

/* Declare YARROWAPI as __declspec(dllexport) before
   including this file in the actual DLL */
#ifndef YARROWAPI
#define YARROWAPI __declspec(dllimport)
#endif

/* Private function forward declarations */
YARROWAPI int prngInitialize(void);
YARROWAPI int prngDestroy(void);
YARROWAPI int prngInputEntropy(BYTE *inbuf,UINT inbuflen,UINT poolnum);

#endif
```

```
/*
        prng.c

        Core routines for the Counterpane PRNG
*/
#include "userdefines.h"
#include <windows.h>
#include <stdio.h>
#include "assertverify.h"

#ifdef WIN_NT
#include "ntonly.h"
#endif
#ifdef WIN_95
#include "95only.h"
#endif
#include "smf.h"
#include "sha1mod.h"
#include "entropysources.h"
#include "usersources.h"
#include "comp.h"

/* DLL Headers */
#define YARROWAPI __declspec(dllexport) /* must be declared before yarrow.h  and prng.h */
#include "yarrow.h"
#include "prng.h"
#include "prngpriv.h"

#define _MAX(a,b) (((a)>(b))?(a):(b))
#define _MIN(a,b) (((a)<(b))?(a):(b))

#pragma data_seg(".sdata")
static MMPTR mmp = MM_NULL;
static HANDLE Statmutex = NULL;
static DWORD mutexCreatorId = 0;
#pragma data_seg()
#pragma comment(linker,"/section:.sdata,rws")

/* Process-specific pointers */
PRNG* p= NULL;
HANDLE mutex = NULL;


BOOL WINAPI DllMain(HANDLE hInst, ULONG ul_reason_for_call, LPVOID lpReserved)
{
        HANDLE caller;

        switch(ul_reason_for_call)
        {
        case DLL_PROCESS_ATTACH:
                if(mmp != MM_NULL)
                {
                        p = (PRNG*)mmGetPtr(mmp);
                }
                if(Statmutex!=NULL)
                {
                        caller = OpenProcess(PROCESS_DUP_HANDLE,FALSE,mutexCreatorId);

        DuplicateHandle(caller,Statmutex,GetCurrentProcess(),&mutex,SYNCHRONIZE,FALSE,0);
                        CloseHandle(caller);
                }
```

```
        break;

    case DLL_THREAD_ATTACH:
        break;

    case DLL_THREAD_DETACH:
        break;

    case DLL_PROCESS_DETACH:
                if(p!=NULL)
                {
                        mmReturnPtr(mmp);
                        p = NULL;
                }
                if(mutex!=NULL) {CloseHandle(mutex);}
        break;
        }

   return TRUE;
}

/* Set up the PRNG */
int m_prngInitialize(void)
{
        UINT i;
        comp_error_status resp;
        int retval = PRNG_ERR_LOW_MEMORY;

        /* Create the mutex */
        if(mutexCreatorId!=0) {return PRNG_ERR_REINIT;}
        Statmutex = CreateMutex(NULL,TRUE,NULL);
        if(Statmutex == NULL) {mutexCreatorId = 0; return PRNG_ERR_MUTEX;}
        DuplicateHandle(GetCurrentProcess(),Statmutex,GetCurrentProcess(),&mutex,SYNCHRONIZE,F
ALSE,0);
        mutexCreatorId = GetCurrentProcessId();

        /* Assign memory */
        mmp = mmMalloc(sizeof(PRNG));
        if(mmp==MM_NULL)
        {
                goto cleanup_init;
        }
        else
        {
                p = (PRNG*)mmGetPtr(mmp);
        }

        /* Initialize Variables */
        for(i=0;i<TOTAL_SOURCES;i++)
        {
                p->poolSize[i] = 0;
                p->poolEstBits[i] = 0;
        }

#ifdef WIN_NT
        /* Setup security on the registry so that remote users cannot predict the slow pool */
        prng_set_NT_security();
#endif

        /* Initialize the secret state. */
        SHA1Init(&p->pool);
        prng_slow_init();/* Does a slow poll and then calls prng_make_state(...) */

        /* Initialize compression routines */
        for(i=0;i<COMP_SOURCES;i++)
        {
                resp = comp_init((p->comp_state)+i);
                if(resp!=COMP_SUCCESS) {retval = PRNG_ERR_COMPRESSION; goto cleanup_init;}
        }

        p->ready = PRNG_READY;

        return PRNG_SUCCESS;

cleanup_init:
        /* Program failed on one of the mmmallocs */
        mmFree(mmp);
        mmp = MM_NULL;
        CloseHandle(Statmutex);
        Statmutex = NULL;
        mutexCreatorId = 0;

        return retval; /* default PRNG_ERR_LOW_MEMORY */
}
```

```
/* Input a state into the PRNG */
int m_prngProcessSeedBuffer(BYTE *buf,LONGLONG ticks)
{
        CHECKSTATE(p);
        GENCHECK(p);
        PCHECK(buf);

        /* Put the data into the entropy, add some data from the unknown state, reseed */
        SHA1Update(&p->pool,buf,20);                                    /* Put it into the entropy
pool */
        prng_do_SHA1(&p->outstate);                /* Output 20 more bytes and     */
        SHA1Update(&p->pool,p->outstate.out,20);/* add it to the pool as well.  */
        prngForceReseed(ticks); /* Do a reseed */
        return prngOutput(buf,20); /* Return the first 20 bytes of output in buf */
}


/* Provide output */
int m_prngOutput(BYTE *outbuf,UINT outbuflen)
{
        UINT i;

        CHECKSTATE(p);
        GENCHECK(p);
        PCHECK(outbuf);
        chASSERT(BACKTRACKLIMIT > 0);

        for(i=0;i<outbuflen;i++,p->index++,p->numout++)
        {
                /* Check backtracklimit */
                if(p->numout > BACKTRACKLIMIT)
                {
                        prng_do_SHA1(&p->outstate);
                        prng_make_new_state(&p->outstate,p->outstate.out);
                }
                /* Check position in IV */
                if(p->index>=20)
                {
                        prng_do_SHA1(&p->outstate);
                }
                /* Output data */
                outbuf[i] = (p->outstate.out)[p->index];
        }

        return PRNG_SUCCESS;
}


/* Take some "random" data and make more "random-looking" data from it */
int prngStretch(BYTE *inbuf,UINT inbuflen,BYTE *outbuf,UINT outbuflen) {
        long int left,prev;
        SHA1_CTX ctx;
        BYTE dig[20];

        PCHECK(inbuf);
        PCHECK(outbuf);

        if(inbuflen >= outbuflen)
        {
                memcpy(outbuf,inbuf,outbuflen);
                return PRNG_SUCCESS;
        }
        else  /* Extend using SHA1 hash of inbuf */
        {
                SHA1Init(&ctx);
                SHA1Update(&ctx,inbuf,inbuflen);
                SHA1Final(dig,&ctx);
                for(prev=0,left=outbuflen;left>0;prev+=20,left-=20)
                {
                        SHA1Update(&ctx,dig,20);
                        SHA1Final(dig,&ctx);
                        memcpy(outbuf+prev,dig,(left>20)?20:left);
                }
                trashMemory(dig,20*sizeof(BYTE));

                return PRNG_SUCCESS;
        }

        return PRNG_ERR_PROGRAM_FLOW;
}


/* Add entropy to the PRNG from a source */
int m_prngInput(BYTE *inbuf,UINT inbuflen,UINT poolnum,UINT estbits)
{
        comp_error_status resp;
```

```
        CHECKSTATE(p);
        POOLCHECK(p);
        PCHECK(inbuf);
        if(poolnum >= TOTAL_SOURCES) {return PRNG_ERR_OUT_OF_BOUNDS;}

        /* Add to entropy pool */
        SHA1Update(&p->pool,inbuf,inbuflen);

        /* Update pool size, pool user estimate and pool compression context */
        p->poolSize[poolnum] += inbuflen;
        p->poolEstBits[poolnum] += estbits;
        if(poolnum<COMP_SOURCES)
        {
                resp = comp_add_data((p->comp_state)+poolnum,inbuf,inbuflen);
                if(resp!=COMP_SUCCESS) {return PRNG_ERR_COMPRESSION;}
        }

        return PRNG_SUCCESS;
}


/* Cause the PRNG to reseed now regardless of entropy pool */ /* Should this be public? */
int m_prngForceReseed(LONGLONG ticks)
{
        int i;
        LONGLONG start;
        LONGLONG now;
#ifdef WIN_NT
        FILETIME a,b,c,usertime;
#endif
        BYTE buf[64];
        BYTE dig[20];

        CHECKSTATE(p);
        POOLCHECK(p);
        ZCHECK(ticks);

        /* Set up start */
#ifdef WIN_NT
        GetThreadTimes(GetCurrentThread(),&a,&b,&c,&usertime);
        start = (usertime.dwHighDateTime<<32 | usertime.dwLowDateTime) * 10000; /* To get # of
ticks */
#endif
#ifdef WIN_95
        start = GetTickCount();
#endif
        do
        {
                /* Do a couple of iterations between time checks */
                prngOutput(buf,64);
                SHA1Update(&p->pool,buf,64);
                prngOutput(buf,64);
                SHA1Update(&p->pool,buf,64);
                prngOutput(buf,64);
                SHA1Update(&p->pool,buf,64);
                prngOutput(buf,64);
                SHA1Update(&p->pool,buf,64);
                prngOutput(buf,64);
                SHA1Update(&p->pool,buf,64);
                /* Set up now */
#ifdef WIN_NT
        GetThreadTimes(GetCurrentThread(),&a,&b,&c,&usertime);
        now = (usertime.dwHighDateTime<<32 | usertime.dwLowDateTime) * 10000; /* To get ticks
*/
#endif
#ifdef WIN_95
        now = GetTickCount();
#endif
        } while ( (now-start) < ticks) ;
        SHA1Final(dig,&p->pool);
        SHA1Update(&p->pool,dig,20);
        SHA1Final(dig,&p->pool);

        /* Reset secret state */
        SHA1Init(&p->pool);
        prng_make_new_state(&p->outstate,dig);

        /* Clear counter variables */
        for(i=0;i<TOTAL_SOURCES;i++)
        {
                p->poolSize[i] = 0;
                p->poolEstBits[i] = 0;
        }
```

```
        /* Cleanup memory */
        trashMemory(dig,20*sizeof(char));
        trashMemory(buf,64*sizeof(char));

        return PRNG_SUCCESS;
}


/* If we have enough entropy, allow a reseed of the system */
int m_prngAllowReseed(LONGLONG ticks)
{
        UINT temp[TOTAL_SOURCES];
        UINT i,sum;
        float ratio;
        comp_error_status resp;


        CHECKSTATE(p);

        for(i=0;i<ENTROPY_SOURCES;i++)
        {
                /* Make sure that compression-based entropy estimates are current */
                resp = comp_get_ratio((p->comp_state)+i,&ratio);
                if(resp!=COMP_SUCCESS) {return PRNG_ERR_COMPRESSION;}
                /* Use 4 instead of 8 to half compression estimate */
                temp[i] = (int)(ratio*p->poolSize[i]*4);
        }
        /* Use minumum of user and compression estimate for compressed sources */
        for(i=ENTROPY_SOURCES;i<COMP_SOURCES;i++)
        {
                /* Make sure that compression-based entropy estimates are current */
                resp = comp_get_ratio((p->comp_state)+i,&ratio);
                if(resp!=COMP_SUCCESS) {return PRNG_ERR_COMPRESSION;}
                /* Use 4 instead of 8 to half compression estimate */
                temp[i] = _MIN((int)(ratio*p->poolSize[i]*4),(int)p->poolEstBits[i]);
        }
        /* Use user estimate for remaining sources */
        for(i=COMP_SOURCES;i<TOTAL_SOURCES;i++) {temp[i] = p->poolEstBits[i];}

        bubbleSort(temp,TOTAL_SOURCES);
        for(i=K,sum=0;i<TOTAL_SOURCES;sum+=temp[i++]); /* Stupid C trick */
        if(sum>THRESHOLD)
                return prngForceReseed(ticks);
        else
                return PRNG_ERR_NOT_ENOUGH_ENTROPY;

        return PRNG_ERR_PROGRAM_FLOW;
}

/* Call a slow poll and insert the data into the entropy pool */
int m_prngSlowPoll(UINT pollsize)
{
        BYTE *buf;
        DWORD len;
        prng_error_status retval;

        CHECKSTATE(p);

        buf = (BYTE*)malloc(pollsize);
        if(buf==NULL) {return PRNG_ERR_LOW_MEMORY;}
        len = prng_slow_poll(buf,pollsize);        /* OS specific call */
        retval = prngInputEntropy(buf,len,SLOWPOLLSOURCE);
        trashMemory(buf,pollsize);
        free(buf);

        return retval;
}


/* Delete the PRNG */
int m_prngDestroy(void)
{
        UINT i;

        if(GetCurrentProcessId()!=mutexCreatorId) {return PRNG_ERR_WRONG_CALLER;}
        if(p==NULL) {return PRNG_SUCCESS;} /* Well, there is nothing to destroy... */

        p->ready = PRNG_NOT_READY;

        for(i=0;i<COMP_SOURCES;i++)
        {
                comp_end((p->comp_state)+i);
        }
```

```
        mmFree(mmp);
        mmp = MM_NULL;
        p = NULL;

        CloseHandle(Statmutex);
        Statmutex = NULL;
        mutexCreatorId = 0;

        return PRNG_SUCCESS;
}

#include "prng.mut" /* This file wraps the above functions for use with the mutex */

/* Utility functions  -  Cannot be called from outside*/
/* All error checking should be done in the function that calls these */
void prng_do_SHA1(GEN_CTX *ctx)
{
        SHA1_CTX sha;

        SHA1Init(&sha);
        SHA1Update(&sha,ctx->IV,20);
        SHA1Update(&sha,ctx->out,20);
        SHA1Final(ctx->out,&sha);
        p->index = 0;
}

void prng_make_new_state(GEN_CTX *ctx,BYTE *state)
{
        SHA1_CTX sha;

        memcpy(ctx->IV,state,20);
        SHA1Init(&sha);
        SHA1Update(&sha,ctx->IV,20);
        SHA1Final(ctx->out,&sha);
        p->numout = 0;
        p->index = 0;
}


/* Initialize the secret state with a slow poll */
#define SPLEN 65536  /* 64K */

void prng_slow_init(void)
/* This fails silently and must be fixed. */
{
        SHA1_CTX* ctx = NULL;
        MMPTR mmctx = MM_NULL;
        BYTE* bigbuf = NULL;
        MMPTR mmbigbuf = MM_NULL;
        BYTE* buf = NULL;
        MMPTR mmbuf = MM_NULL;
        DWORD polllength;

        mmbigbuf = mmMalloc(SPLEN);
        if(mmbigbuf == MM_NULL) {goto cleanup_slow_init;}
        bigbuf = (BYTE*)mmGetPtr(mmbigbuf);

        mmbuf = mmMalloc(20);
        if(mmbuf == MM_NULL) {goto cleanup_slow_init;}
        buf = (BYTE*)mmGetPtr(mmbuf);

        mmctx = mmMalloc(sizeof(SHA1_CTX));
        if(mmctx == MM_NULL) {goto cleanup_slow_init;}
        ctx = (SHA1_CTX*)mmGetPtr(mmctx);


        /* Initialize the secret state. */
        /* Init entropy pool */
        SHA1Init(&p->pool);
        /* Init output generator */
        polllength = prng_slow_poll(bigbuf,SPLEN);
        SHA1Init(ctx);
        SHA1Update(ctx,bigbuf,polllength);
        SHA1Final(buf,ctx);
        prng_make_new_state(&p->outstate,buf);

cleanup_slow_init:
        mmFree(mmctx);
        mmFree(mmbigbuf);
        mmFree(mmbuf);

        return;
}
```

```
void trashMemory(void* mem,UINT len)
/* This function should only be used on data in RAM */
{
        /* Cycle a bit just in case it is one of those weird memory units */
        /* No, I don't know which units those would be */
        memset(mem,0x00,len);
        memset(mem,0xFF,len);
        memset(mem,0x00,len);
}

/* In-place modifed bubble sort */
void bubbleSort(UINT *data,UINT len)
{
        UINT i,last,newlast,temp;

        last = len-1;
        while(last!=-1)
        {
                newlast = -1;
                for(i=0;i<last;i++)
                {
                        if(data[i+1] > data[i])
                        {
                                newlast = i;
                                temp = data[i];
                                data[i] = data[i+1];
                                data[i+1] = temp;
                        }
                }
                last = newlast;
        }
}
```

5. Fortuna

Fortuna juga sebuah *cryptographically secure pseudo-random number generator* yang dibuat oleh Bruce Schneier dan Niels Ferguson. Fortuna diambil dari nama dewa Roma yang menguasai kesempatan. Lebih tepat lagi, Fortuna adalah anggota keluarga dari *pseudo-random number generator* yang aman; rancangannya memberikan beberapa pilihan untuk para implementornya. Fortuna terdiri dari beberapa bagian [6]:

- *Generator* itu sendiri, yang sekali diberi *seed* akan memproduksi sejumlah data *pseudo-random* yang tak terbatas.
- *Entropy Accumulator*, yang mengumpulkan data acak asli dari berbagai sumber dan menggunakannya untuk memberi *seed* pada *generator* ketika tingkat acak yang cukup telah hadir.
- Arsip *seed*, yang menyimpan banyak status yang memungkinkan komputer untuk memulai produksi bilangan-bilangan acak sesegera mungkin setelah dinyalakan.

Generator berdasarkan *block chiper* manapun yang baik. Practical Cryptography menyarankan AES, Serpent, atau Twofish. Ide dasarnya adalah untuk menjalankan *chiper* dalam *counter mode*, mengenkripsi nilai-nilai *counter* yang terus bertambah. Dengan sendirinya, hal ini akan memproduksi *statistically identifiable deviations* dari tingkat acak; sebagai contoh, menciptakan 2^{65} 128-bit *block* yang acak akan memproduksi rata-rata satu pasang block yang identik, tapi ada *block* yang berulang sama sekali pada 2^{128} *block* pertama yang diproduksi oleh 128-bit *chiper* dalam *counter mode*. Maka dari itu, kunci diganti secara berkala: tidak lebih dari 1MB data yang diciptakan tanpa perubahan kunci. Kunci juga diganti setelah permintaan tiap data (betapa pun kecilnya), jadi sebuah kunci tidak akan membahayakan keluaran RNG yang lama.

Entropy accumulator dirancang agar tahan terhadap serangan "injection", tanpa memerlukan peramal entropy yang rumit (dan secara tak langsung tak dapat diandalkan). Terdapat beberapa "pools" entropy; masing-masing sumber entropy mendistribusikan entropy yang sama rata ke pools; dan (di sinilah ide pokok) pada bilangan ke *n* yang menjadi *seed generator*, pool *k* digunakan hanya jika 2^k membagi *n*. Lalu, pool yang ke *k* digunakan hanya $1/2^k$ dari keseluruhan waktu. Pool yang bernomor lebih tinggi, dengan kata lain, (1) berkontribusi untuk membuat *seed* lebih jarang tapi (2) mengumpulkan jumlah entropy yang lebih banyak antar *seed*. Membuat *seed* dilakukan dengan me-hash pool entropy yang telah dispesifikasikan ke dalam kunci *block chiper* menggunakan dua iterasi SHA-256.

Kecuali si penyerang dapat mengontrol semua sumber entropy yang mengalir ke dalam sistem (dalam kasus di mana tidak ada satupun

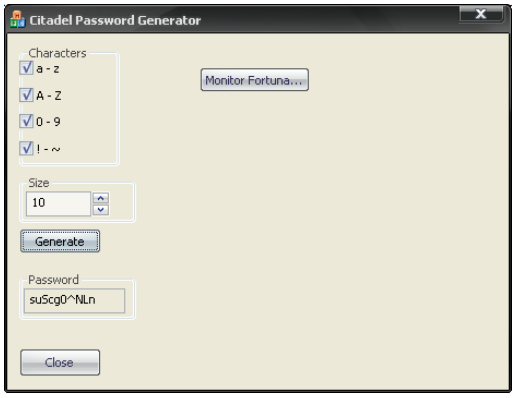
algoritma yang menyelamatkannya dari *compromise*), akan ada beberapa *k* di mana pool yang ke *k* mengumpulkan entropy yang cukup antar *seed* akan menjamin keamanan. Dan pool itu akan digunakan pada saat interval yang proporsional dengan jumlah entropy yang dipertanyakan. Maka dari itu, sistem akan selalu menyembuhkan diri dari serangan *injection*, dan waktu yang diperlukan untuk melakukannya adalah sebuah *constant factor* yang lebih besar dari *theoretical time* yang dapat diambilnya jika kita dapat mengidentifikasi sumber mana dari entropy yang rusak dan mana yang tidak.

Kesimpulan ini bergantung pada cukupnya jumlah pool. Fortuna menggunakan 32 pool, dan membatasi pembuatan *seed* hingga terjadi 10 kali per detik. Kehabisan pool akan memakan waktu sekitar 13 tahun, yang Ferguson dan Schneier rancang cukup lama untuk tujuan praktek. Implemetor yang paranoid, atau mereka yang memerlukan penciptaan data acak dengan tingkat kolosal dan pembuatan *seed* yang sering, dapat menggunakan jumlah pool yang lebih banyak.

Fortuna berbeda dari algoritma Yarrow terlebih dalam penanganan *entropy accumulator*. Yarrow memerlukan tiap sumber entropy ditemani oleh sebuah mekanisme untuk memperkirakan jumlah tepat entropy yang disuply, dan menggunakan hanya dua pool; dan menyarankan *embodiment* (dinamakan Yarrow-160) menggunakan SHA-1 daripada iterasi pada SHA-256.

Ada beberapa website yang memuat source fortuna [7], tapi tidak ada dituliskan dalam paper ini karena pada dasarnya mirip dengan Yarrow, selain itu juga untuk menghemat ruang penulisan.

Namun demikian penulis memperoleh salah satu implementasi dari algoritma Fortuna yang digunakan untuk membangkitkan kata kunci bagi penggunanya. Diperoleh dari Citadel Software, di bawah ini adalah *screenshot* dari program (lihat **Gambar 1**). Sebagai catatan tambahan, program bersifat *open source* yang dapat diperoleh secara cuma-cuma dari [7].



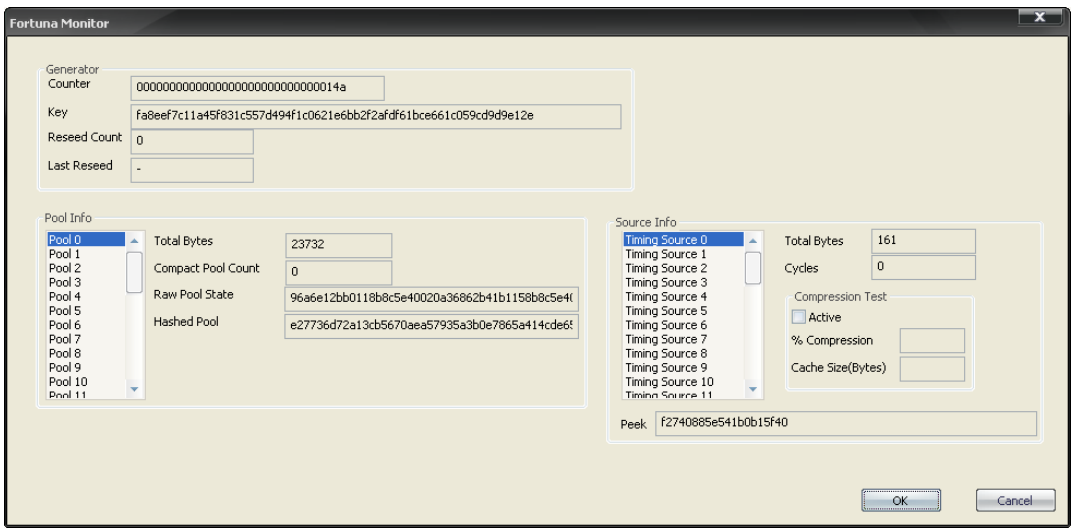
Gambar 1 Tampilan Awal Program

Program memuat pilihan karakter yang ingin ditampilkan dalam kata kunci yang akan diciptakan. Selain itu juga diberi pilihan untuk memasukkan panjang karakter yang ingin diciptakan, maksimum sepanjang 999 karakter.

Jika kita mengklik tombol ‘Generate’, maka program akan menciptakan sebuah kata kunci sesuai dengan pilihan-pilihan yang telah kita tentukan sebelumnya.

Kini akan lebih detil dilihat mengenai kerja Fortuna dalam program ini. Jika diklik tombol ‘Monitor Fortuna’ (lihat **Gambar 2** di bawah).

Dalam Monitor Fortuna tersebut dapat dilihat



Gambar 2 Tampilan Monitor Fortuna

adanya Generator yang menghitung *counter* seperti dijelaskan sebelumnya, ada Pool Info yang menunjukkan ke 32 *pool* dalam Fortuna dan keterangannya masing-masing, serta ada Source Info yang memuat keterangan sumber-sumber yang digunakan dalam program ini.

Menurut pengembang program [5], tujuan dari perancangan implementasi Fortuna ini adalah untuk mengembangkan sebuah PRNG yang cocok digunakan oleh *web server*. Satu fitur yang khas dari *web server* adalah pada umumnya tidak banyak masukkan yang diperoleh dari *mouse* atau *keyboard*. Maka dari itu, implementasi dari fortuna ini menggunakan informasi yang berasal dari Windows sebagai sumber entropy bagi si *generator*. Perlu dicatat bahwa generator tipe ini lebih tepat dinamakan '*chaotic deterministic*' daripada '*pseudo random*' karena sumber-sumbernya tidak sepenuhnya acak. Beberapa entropy dari implementasi Fortuna ini datang dari penggunaan *timer* berkualitas tinggi untuk menghitung waktu *call* dari sistem operasi. Perlu diperhatikan bahwa *timing data* bukan entropy yang sebenarnya karena nilainya deterministik. Ketidakmampuan untuk memprediksi hasil keluaran dari PRNG adalah hasil yang diinginkan.

Tujuan perancangan yang lain dari implementasi Fortuna ini adalah untuk membuat PRNG menjadi lebih sulit diserang. Hal ini dilakukan dengan:

- Adanya volum data yang amat besar yang dimasukkan ke dalam pool, dengan beberapa entropy yang tinggi dan beberapa lainnya rendah, Si penyerang akan menemukan kesulitan untuk menghitung status dari sebuah pool karena volum data yang besar ini.
- Data dari *registry* Windows digunakan hingga data dari aplikasi menjadi spesifik di mana aplikasi itu dijalankan. Hal ini dimasukkan sebagai sumber entropy yang rendah untuk *accumulator*.
- Informasi status dari masing-masing proses (*page faults, memory, I/O Reads and Writes*) digunakan sebagai sumber entropy.
- Penggunaan yang ekstensif dari *timer* dengan tingkat presisi yang tinggi sebagai sumber entropy.
- Menghindari penyimpanan data sederhana untuk kunci data acak yang digunakan dalam PRNG untuk menghindari *memory scanning attacks*.
- Berhati-hati dalam mengacak data acak dalam *memory* ketika data tersebut tidak digunakan lagi.

Perlu dicatat bahwa sumber entropy yang sesungguhnya memerlukan sumber fisik.

Jika:

- Keamanan jaringan bisa dipercaya.
- Yakin bahwa penyerang tidak dapat mengakses *registry* sepenuhnya dan tidak dapat melihat seluruh status yang sedang berjalan di komputer
- Yakin bahwa penyerang tidak dapat menjalankan perangkat lunaknya sendiri pada mesin untuk menentukan isi dari 32 data pool dan menyerang generator.

Maka PRNG ini cocok digunakan.

Untuk dapat menyerang implementasi Fortuna ini, penyerang harus dapat mengakses seluruh *registry*, melihat seluruh proses, dan lain sebagainya. Seorang penyerang harus dapat menyerang mesin seutuhnya. Karena hal ini, dipercaya implementasi dari Fortuna dapat diterima oleh banyak aplikasi berbasis web.

6. Jenis-jenis Serangan Pada PRNG

Begitu banyak serangan yang dapat dilakukan terhadap sebuah PRNG adalah alasan diperlukannya sebuah PRNG yang secara kriptografi aman terhadap serangan. Dari banyaknya cara serangan, terdapat tiga kelas serangan [8] yang dapat diterapkan pada sebuah PRNG.

1. **Serangan Kripanalisis Langsung**. Ketika seorang penyerang secara langsung dapat membedakan hasil keluaran PRNG dan hasil keluaran acak, serangan ini dinamakan serangan kripanalisis langsung. Jenis serangan ini dapat diaplikasikan pada banyak, tapi tak semua, penggunaan PRNG. Sebagai contoh, sebuah PRNG digunakan hanya untuk menggenerasi kunci *triple*-DES mungkin tak mempan terhadap serangan ini semenjak keluaran PRNG tidak pernah dapat dilihat secara langsung.
2. **Serangan Berbasiskan Masukkan (*input*).** Sebuah serangan berbasiskan masukkan terjadi ketika seorang penyerang dapat menggunakan pengetahuan atau kendali dari masukkan PRNG untuk mengkripanalisis PRNG tersebut, misalnya, untuk membedakan antara

keluaran PRNG dengan nilai-nilai acak. Serangan berbasiskan masukkan dapat lebih lanjut dibedakan menjadi serangan *known-input*, *replayed-input*, dan *chosen-input*. Serangan *chosen-input* mungkin lebih praktis digunakan untuk *smart-cards* dan alat-alat yang tahan pemalsuan fisik/kripanalisis lainnya; serangan jenis ini juga praktis untuk aplikasi yang membaca masukkan pesan, kata kunci pilihan pengguna, statistik jaringan, dan sebagainya sebagai sumber entropy-nya. Serangan *replayed-input* lebih kurang praktis untuk situasi yang sama, tapi memerlukan kendali atau kerumitan yang lebih sedikit dari pihak si penyerang. Serangan *known-input* mungkin praktis pada situasi di mana beberapa dari masukkan PRNG, yang seharusnya sulit diprediksi, menjadi amat mudah diprediksi. (Contoh yang amat nyata dari kasus ini adalah ketika sebuah aplikasi yang menggunakan *hard-drive latency* sebagai beberapa masukkan PRNG, tapi berjalan di atas *drive* jaringan yang *timing*-nya dapat diketahui oleh penyerang.)

3. ***State Compromise Extention Attacks*.** Serangan ini berusaha untuk memperpanjang kesempatan dari usaha yang berhasil sebelumnya yang telah berhasil memperoleh nilai *S* selama mungkin. Diasumsikan bahwa, entah dengan alasan apapun – penetrasi sementara dari keamanan komputer, kebocoran yang tak terduga, kesuksesan kripanalisis, dan lain sebagainya – si penyerang telah berhasil untuk mempelajari status internal, *S*, dalam suatu waktu. Serangan jenis ini berhasil ketika si penyerang berhasil mengetahui keluaran PRNG yang sebelumnya tidak diketahui (atau membedakan antara keluaran PRNG dengan nilai-nilai acak) sejak sebelum *S compromised*, atau memperoleh keluaran setelah PRNG mengumpulkan *sequence* masukkan yang penyerang tidak tahu. Serangan ini akan berjalan ketika sebuah PRNG dimulai pada status yang tidak aman (mudah ditebak) karena entropy awal yang tidak memadai. Serangan ini juga akan berjalan ketika *S* telah di-*compromised* oleh serangan-serangan di bawah ini, atau dengan metode lainnya. Dalam prakteknya, secara umum dapat diasumsikan bahwa *compromise* dari status *S* secara berkala terjadi; untuk mempertahankan kemangkusan dari sistem, PRNG harus dapat menolak serangan ini.
   a. **Backtracking Attacks.** Serangan ini menggunakan compromise dari status PRNG *S* pada saat *t* untuk mempelajari keluaran PRNG sebelumnya.
   b. **Permanent Compromise Attacks.** Serangan ini terjadi jika, sekali si penyerang *compromise S* pada saat *t*, semua nilai *S* baik itu di masa lalu atau masa yang akan datang, menjadi rentan terhadap serangan.
   c. **Iterative Guessing Attacks.** Sebuah serangan menebak yang iteratif menggunakan pengetahuan atas *S* pada saat *t*, dan keluaran-keluaran PRNG di antaranya, untuk mempelajari *S* pada saat $t + e$, ketika masukkan dikumpulkan pada rentang waktu ini menjadi mudah ditebak (tapi tidak diketahui) oleh si penyerang.
   d. **Meet-in-the-Middle Attacks.** Serangan ini pada prinsipnya adalah sebuah kombinasi dari *iterative guessing attacks* dengan *backtracking attacks*. Pengetahuan atas *S* pada saat *t* dan $t + 2e$ mengizinkan penyerang untuk memperoleh *S* pada saat $t + e$.

7. Diskusi dan Kesimpulan

Sebuah PRNG amat dibutuhkan oleh berbagai aplikasi yang ada saat ini. Tak jarang aplikasi-aplikasi ini membutuhkan tingkat keamanan yang tingi dan hasil bilangan acak dari sebuah PRNG bisa mempengaruhi hal ini. Oleh karena itu dibutuhkan sebuah PRNG yang secara kriptografi aman terhadap berbagai macam serangan.

Dari berbagai jenis PRNG, beberapa yang diketahui dan terbukti aman terhadap serangan kriptografi adalah Blum Blum Shub, Yarrow, dan Fortuna. Blum Blum Shub adalah algoritma yang sudah lama ada dan menggunakan perhitungan matematis yang rumit sebagai pertahanan terhadap berbagai serangan yang mungkin. Yarrow diciptakan beberapa tahun kemudian untuk menjawab beberapa kekurangan dari algoritma-algoritma

yang sudah ada sebelumnya, termasuk Blum Blum Shub. Perbaikan dari Yarrow lahir dengan nama Fortuna, diciptakan oleh beberapa orang yang sama sebagai pencipta Yarrow, yang menggunakan 32 pool dan sumber-sumber entropy yang semakin berkembang pada saat ini.

8. Daftar Referensi

[1] Blum Blum Shub. http://en.wikipedia.org/wiki/Blum_Blum_Shub

[2] Bruce Schneier. Yarrow – A secure pseudorandom number generator. http://www.schneier.com/yarrow.html

[3] com.modp.random – A collection of pseudo-random number generators implemented in Java. http://modp.com/release/javarng/

[4] Cryptographically secure pseudo-random number generator. http://en.wikipedia.org/wiki/Cryptographically_secure_pseudo-random_number_generator

[5] Fortuna - A Cryptographically Secure Pseudo Random Number Generator. http://www.codeproject.com/cpp/FortunaCSI.asp

[6] Fortuna (PRNG). http://en.wikipedia.org/wiki/Fortuna_%28PRNG%29

[7] Fortuna prng. http://www.citadelsoftware.ca/fortuna/Fortuna.htm

[8] Kelsey, John; Schneier, Bruce; Wagner, David. Cryptanalytic Attacks on Pseudorandom Number Generator. [paper]

[9] NIST Special Publication 800-22. May 15, 2001. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.

[10] Yarrow algorithm. http://en.wikipedia.org/wiki/Yarrow_algorithm