Bab 6

Kriptografi Stream Cipher

Deretan kunci untuk enkripsi *one-time pad* dapat dipandang sebagai suatu *keystream* yang tidak mempunyai periode. Kriptografi *stream cipher* mencoba menggunakan konsep ini tetapi dengan *keystream* yang mempunyai periode dan menggunakan *generator* yang relatif pendek berupa kunci. Baik enkripsi *one-time pad* maupun *stream cipher* mendapatkan naskah acak dari *exclusive or* (XOR) naskah asli dengan *keystream*, jadi keduanya merupakan apa yang dinamakan *Vernam cipher* (lihat tabel 6.1) yang ditemukan oleh Gilbert Vernam tahun 1917. Bedanya hanya pada pembuatan *keystream*:

- Untuk *one-time pad*, *keystream* didapat langsung dari *key generation* (*random number generation*).
- Untuk *stream cipher*, *keystream* didapat dari *pseudo-random number generation* menggunakan kunci enkripsi.

10010111001011101001...	naskah asli
01001110001101001101...	*keystream*
11011001000110100100...	naskah acak

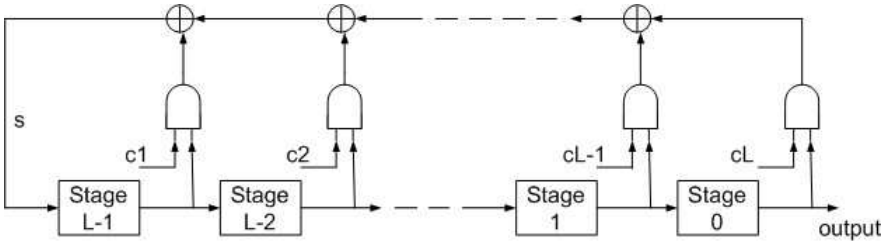
Tabel 6.1: *Vernam Cipher*

Berbeda dengan *keystream* untuk *one-time pad* yang tidak mempunyai periode, *keystream* untuk *stream cipher* mempunyai periode (kecuali jika naskah acak dijadikan *feedback*), meskipun periode sangat panjang. Ini karena *pseudo-random number generator* adalah suatu *deterministic finite state automaton*, jadi karena jumlah *state finite* dan kunci juga *finite*, maka setelah seluruh kunci diproses, banyaknya *state* yang dapat dikunjungi terbatas dan *automaton* akan

kembali ke *state* yang pernah dikunjungi, dan karena *automaton* bersifat *deterministic*, maka siklus akan diulang. Beberapa cara *pseudo-random number generation* untuk mendapatkan *keystream* antara lain:

- menggunakan *linear feedback shift register* (LFSR),
- menggunakan *block cipher* dalam *feedback mode* (lihat bagian 7.2), dan
- menggunakan *state automaton* dalam *software* (contohnya RC4).

Gambar 6.1 memperlihatkan suatu *linear feedback shift register* (LFSR). Setiap



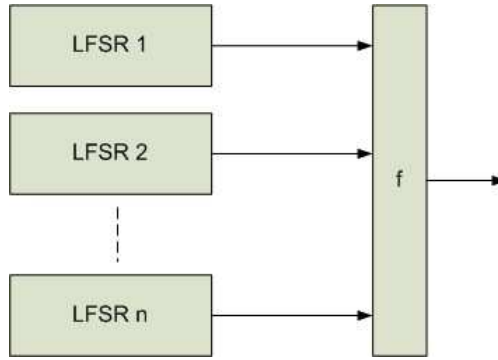
Gambar 6.1: Linear feedback shift register

stage i, $0 \leq i \leq L-1$ dipisahkan menggunakan *delay flip-flop*, atau disebut juga *latch*. Setiap $cL - i$ dimana $0 \leq i \leq L-1$ menentukan apakah *stage* i ditambahkan ke *feedback* s (modulo 2), jika $cL - i$ mempunyai nilai 1 maka *stage* i ditambahkan, sedangkan jika $cL - i$ mempunyai nilai 0 maka *stage* i tidak ditambahkan. Setiap *clock cycle*, *output* 1 bit didapat dari *stage* 0. Untuk membuat *pseudo-random number generator*, beberapa LFSR biasanya dikombinasikan menggunakan fungsi non-linear. Gambar 6.2 memperlihatkan kombinasi LFSR menggunakan fungsi non-linear f.

Beberapa kelemahan *stream cipher* antara lain:

- Jika naskah asli dan naskah acak diketahui, maka menggunakan xor kita bisa dapatkan *keystream*.
- *Stream cipher* rentan terhadap *tampering*. Seseorang yang mengetahui posisi data tertentu dalam naskah acak dan mengetahui nilai data tersebut bisa mengubahnya menggunakan xor. Contohnya, angka 50 bisa diubah menjadi angka 99 dengan melakukan $x \oplus (50 \oplus 99)$, dimana x merepresentasikan 50 dalam naskah acak.

Efek serupa dengan penggunaan kunci secara berulang pada enkripsi *one-time pad* juga mudah terjadi. Penggunaan *stream cipher* memang harus dengan sangat hati-hati, oleh sebab itu buku ini tidak merekomendasikan penggunaan *stream cipher*.



Gambar 6.2: Kombinasi non-linear LFSR

6.1 RC4

RC4 adalah *stream cipher* yang dirancang di RSA Security oleh Ron Rivest tahun 1987. Pada mulanya cara kerja RC4 dirahasiakan oleh RSA Security, akan tetapi ini dibocorkan di internet tahun 1994 di milis Cypherpunks. RSA Security tidak pernah merilis RC4 secara resmi, akibatnya banyak yang menyebutnya sebagai ARC4 (*alleged* RC4 atau tersangka RC4) untuk menghindari masalah *trademark*.

Berbeda dengan mayoritas *stream cipher* sebelumnya yang implementasinya dioptimalkan untuk *hardware* menggunakan *linear feedback shift registers*, RC4 dirancang agar dapat diimplementasikan di *software* secara sangat efisien. Ini membuat RC4 sangat populer untuk aplikasi internet, antara lain RC4 digunakan dalam standard TLS (*transport layer security*) dan WEP (*wireless equivalent privacy*).

Cara membuat *keystream* dalam RC4 adalah dengan *state automaton* dan terdiri dari dua tahap:

1. Tahap *key scheduling* dimana *state automaton* diberi nilai awal berdasarkan kunci enkripsi.

2. Tahap *pseudo-random generation* dimana *state automaton* beroperasi dan outputnya menghasilkan *keystream*.

Tahap pertama dilakukan menggunakan *key scheduling algorithm* (KSA). *State* yang diberi nilai awal berupa *array* yang merepresentasikan suatu permutasi dengan 256 elemen, jadi hasil dari algoritma KSA adalah permutasi awal. *Array* yang mempunyai 256 elemen ini (dengan indeks 0 sampai dengan 255) dinamakan `S`. Berikut adalah algoritma KSA dalam bentuk *pseudo-code* dimana

`key` adalah kunci enkripsi dan `keylength` adalah besar kunci enkripsi dalam *bytes* (untuk kunci 128 bit, `keylength` = 16):

```
for i = 0 to 255
  S[i] := i
j := 0
for i = 0 to 255
  j := (j + S[i] + key[i mod keylength]) mod 256
  swap(S[i],S[j])
```

Tahap kedua menggunakan algoritma yang dinamakan *pseudo-random generation algorithm* (PRGA). Setiap putaran, bagian *keystream* sebesar 1 byte (dengan nilai antara 0 sampai dengan 255) dioutput oleh PRGA berdasarkan *state* `S`. Berikut adalah algoritma PRGA dalam bentuk *pseudo-code*:

```
i := 0
j := 0
loop
  i := (i + 1) mod 256
  j := (j + S[i]) mod 256
  swap(S[i],S[j])
  output S[(S[i] + S[j]) mod 256]
```

Permutasi dengan 255 elemen mempunyai 255! kemungkinan. Ditambah dua indeks (i dan j) yang masing-masing dapat mempunyai nilai antara 0 dan 255, maka *state automaton* yang digunakan untuk membuat *keystream* mempunyai

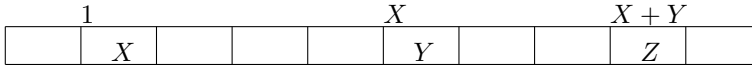
$$255! \times 255^2 \approx 2^{1700}$$

kemungkinan *internal states*. Karena banyaknya jumlah kemungkinan untuk *internal state*, sukar untuk memecahkan RC4 dengan menganalisa PRGA (teknik paling efisien saat ini harus menjajagi $> 2^{700}$ kemungkinan).

Karena algoritma KSA relatif sangat sederhana, banyak ahli kriptografi yang fokus pada KSA dalam mencari kelemahan enkripsi RC4. Fokus ini membuahkan hasil dengan ditemukannya beberapa kelemahan KSA yang dapat digunakan untuk memecahkan aplikasi RC4, termasuk memecahkan penggunaan RC4 dalam protokol WEP. Metode pertama yang digunakan untuk memecahkan WEP dikembangkan oleh Fluhrer, Mantin dan Shamir [flu01], kita sebut saja FMS *attack*. Implementasi pertama FMS *attack* dilaporkan di [stu01]. Berikut kita bahas enkripsi RC4 dalam WEP dan metode FMS *attack* yang dapat digunakan untuk memecahkannya.

WEP menggunakan gabungan *initialization vector* (IV) dan kunci rahasia (yang diketahui oleh pengirim dan penerima) untuk mendapatkan kunci RC4 (kunci enkripsi dimulai dengan IV diikuti oleh kunci rahasia). Untuk setiap paket, IV sebesar 3 byte digunakan bersama kunci rahasia untuk mengenkripsi

paket. IV dikirim tanpa terenkripsi bersama paket yang dienkripsi, jadi IV dapat diketahui oleh pemecah. Byte pertama dalam paket yang dienkripsi juga selalu diketahui, yaitu byte dengan nilai $0xAA$ (10101010) di-XOR dengan byte pertama output dari PRGA. Byte pertama output PRGA hanya tergantung pada 3 elemen *state* S saat KSA baru saja selesai sebagai berikut, dimana nilai byte output pertama dilabel Z:



Metode pemecahan berfokus pada kasus dimana 3 elemen tersebut diatas ditentukan oleh komponen kunci. Saat KSA berada pada tahap $i \geq 1$, maka $X = S_i[1]$ dan $X + Y = S_i[1] + S_i[S_i[1]]$. Jika kita modelkan pertukaran elemen sebagai acak, maka ada kemungkinan yang meskipun kecil tetapi cukup signifikan (probabilitas ≈ 0.05) bahwa ketiga elemen tersebut diatas tidak ditukar. Untuk WEP dimana IV sebesar 3 byte, kunci enkripsi terdiri dari $(IV[0], IV[1], IV[2], K[0], K[1], \ldots, K[l-1])$ dimana K adalah kunci rahasia sebesar l byte. Untuk mendapatkan byte $K[B]$ kunci rahasia, maka setelah 3 langkah pertama KSA kita butuhkan

$$S_3[1] < 3 \text{ dan } S_3[1] + S_3[S_3[1]] = 3 + B. \tag{6.1}$$

Dengan probabilitas sekitar 0.05, setelah langkah $3 + B$, ketiga elemen yang menentukan byte pertama output PRGA tidak ditukar lagi[1]. Jika syarat 6.1 dipenuhi, kemungkinan terbesar untuk nilai byte pertama output adalah:

$$Out = S_{3+B-1}[j_{3+B}] = S_{3+B-1}[j_{3+B-1} + K[B] + S_{3+B-1}[3+B]].$$

Karena nilai j_{3+B-1} dan $S_{3+B-1}[3+B]$ dapat diketahui, maka nilai $K[B]$ dapat diprediksi sebagai berikut, dimana $S_i^{-1}[X]$ adalah indeks untuk elemen S_i yang mempunyai nilai X:

$$K[B] = S_{3+B-1}^{-1}[Out] - j_{3+B-1} - S_{3+B-1}[3+B].$$

Dengan syarat 6.1 terpenuhi, prediksi ini benar sekitar 5% dari semua percobaan (jika ada 100 percobaan acak, maka 5 percobaan menghasilkan prediksi yang benar). Jika cukup banyak percobaan yang dilakukan maka hasil yang benar akan menonjol dibandingkan hasil-hasil lainnya. Untuk mendapatkan hasil yang baik, diperlukan sekitar 60 IV dengan nilai X yang berbeda dengan format

$$(B + 3, 255, X).$$

Format ini diperlukan untuk memenuhi syarat 6.1. Untuk mendapatkan seluruh kunci rahasia, pencarian dimulai dengan $B = 0$, kemudian $B = 1$ dan

[1]Kondisi ini disebut *resolved condition*.

seterusnya sampai dengan $B = 12$ untuk WEP. Eksperimen yang dilaporkan [stu01] menunjukkan bahwa sekitar 6 juta paket WEP diperlukan untuk mendapatkan kunci rahasia. Banyak jaringan Wi-Fi yang hanya menggunakan satu password untuk semua pengguna (jadi hanya ada satu kunci rahasia). Jika jaringan tersebut mempunyai kepadatan lalu-lintas yang moderat, kunci rahasia dapat ditemukan dalam waktu sehari.

Setelah FMS *attack* dipublikasikan, banyak peneliti yang mencoba menganalisa kelemahan WEP lebih lanjut. Ada beberapa peneliti yang berhasil memperkecil jumlah paket WEP yang diperlukan untuk mendapatkan rahasia sehingga WEP dapat dipecahkan dalam waktu yang cukup singkat (bisa dibawah 60 detik). Kita akan bahas dua *attack* yang kinerjanya cukup menakjubkan yaitu *chopchop attack* dan PTW *attack*.

Chopchop attack sebetulnya bukan *attack* untuk mendapatkan kunci WEP, tetapi merupakan *attack* untuk mendekripsi paket WEP tanpa mengetahui kunci WEP. *Attack* cerdik ini dikembangkan oleh seseorang yang menggunakan nama KoreK dan dipublikasikan dalam milis netstumbler (dapat diakses di website `http://www.netstumbler.org`). *Attack* ini didasarkan pada pengamatan bahwa dalam suatu jaringan WiFi, *access point* menindak-lanjuti suatu paket hanya jika paket tersebut lolos pengecekan CRC (*cyclic redundancy check*). Jika byte terahir dari paket dihilangkan, hampir dapat dipastikan bahwa CRC paket harus diubah. Bagaimana CRC harus diubah menentukan nilai asli dari byte yang dihilangkan. Dengan mencoba semua kemungkinan pengubahan CRC dan mengamati reaksi *access point* terhadap paket yang telah diubah CRC-nya, penyadap dapat mengetahui bagaimana CRC harus diubah, dan dengan demikian dapat mengetahui nilai asli byte. Setelah CRC dan nilai byte terahir didapat, penyadap dapat mengulang proses dengan juga menghilangkan byte kedua dari terahir, dan seterusnya. Matematika yang digunakan untuk CRC tentunya adalah aritmatika *polynomial field* (lihat bagian 5.7). Jika pesan diinterpretasikan sebagai *polynomial* P dalam $\mathbf{GF}(2)[x]$, maka

$$P \bmod R_{CRC} = \sum_{i=0}^{31} x^i$$

dimana R_{CRC} adalah *polynomial* untuk CRC32:

$$\begin{aligned} R_{CRC} &= x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} \\ &\quad + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1. \end{aligned}$$

Karena R_{CRC} *irreducible*, maka aritmatika modulo R_{CRC} adalah aritmatika *polynomial field*. Pesan P dapat ditulis sebagai

$$P = Q \cdot x^8 + P_7$$

dimana P_7 adalah $P \bmod x^8$, jadi P_7 merepresentasikan nilai asli byte terahir. Karena P mempunyai *checksum* yang benar, maka

$$(Q \cdot x^8 + P_7) \bmod R_{CRC} = \sum_{i=0}^{31} x^i.$$

Menggunakan aritmatika *polynomial* modulo R_{CRC} kita bisa dapatkan $(x^8)^{-1}$, jadi

$$Q \equiv (x^8)^{-1}(P_7 + \sum_{i=0}^{31} x^i) \pmod{R_{CRC}}.$$

Jadi kita perlu koreksi Q menjadi Q' untuk mendapatkan *checksum* yang benar. Ini dapat dilakukan dengan menambahkan (modulo R_{CRC}) P_{CORR} ke Q, dimana

$$P_{CORR} = ((x^8)^{-1}(P_7 + \sum_{i=0}^{31} x^i) + \sum_{i=0}^{31} x^i) \bmod R_{CRC}.$$

Dengan mencoba semua kemungkinan nilai P_7 (dari 0 sampai dengan 255), dan mengamati reaksi dari *access point*, penyadap dapat menentukan nilai asli byte terahir.

PTW *attack* (Pyshkin, Tews dan Weinmann) (lihat [tew07]) dikembangkan dari analisa yang dilakukan oleh Andreas Klein (lihat [kle07]). Selain kelemahan pada KSA, analisa juga menunjukkan kecenderungan pada algoritma PRGA. Kecenderungan ini dirumuskan dalam teorema berikut yang membuat korelasi antara i dan *internal state*.

Teorema 27 *Jika probabilitas internal state untuk RC4 terdistribusi secara uniform, dengan setiap pilihan i, kita dapatkan berbagai probabilitas sebagai berikut:*

1. *Dengan $k = S[i] + S[j]$ dan $n = 256$, kita dapatkan*

$$P(S[j] + S[k] \equiv i \pmod{n}) = \frac{2}{n}.$$

2. *Untuk $c \not\equiv i \pmod{n}$, kita dapatkan*

$$P(S[j] + S[k] \equiv c \pmod{n}) = \frac{n-2}{n(n-1)}.$$

Pembuktian teorema 27 adalah sebagai berikut. Untuk bagian 1, kita buktikan bahwa untuk setiap $x \in \{0, 1, \ldots, 255\}$,

$$P(S[j] + S[k] \equiv i \pmod{n} \mid S[j] = x) = \frac{2}{n}.$$

Kita hitung banyaknya *internal state* dimana $S[j] + S[k] \equiv i \pmod{n}$ dan $S[j] = x$. Karena $k = (S[i] + S[j]) \bmod n$, maka $S[j] + S[k] \equiv i \pmod{n}$ dapat ditulis sebagai $k + S[k] \equiv i + S[i] \pmod{n}$ dengan penjelasan sebagai berikut:

$$\begin{aligned} S[j] + S[k] &\equiv i \pmod{n}, \\ k + S[j] + S[k] &\equiv S[i] + S[j] + i \pmod{n}, \\ k + S[k] &\equiv i + S[i] \pmod{n}. \end{aligned}$$

Kita bagi menjadi dua kasus yaitu kasus $i = k$ dan $i \neq k$.

1. Untuk $i = k$, karena $S[i] = S[k]$, tidak ada syarat lainnya, persamaan langsung tercapai. Untuk sisa $S[y]$ dimana $y \neq i = k$, terdapat $(n-1)!$ kemungkinan permutasi.

2. Untuk $i \neq k$, kita harus buat $S[k] = (i - x) \bmod n$ dan $S[i] = (k + S[k] - i) \bmod n$. Terdapat $n - 1$ pilihan untuk k dan sisanya (setelah i dan k terpilih) masih ada $(n-2)!$ kemungkinan permutasi. Jadi untuk $i \neq k$ ada $(n-1)(n-2)! = (n-1)!$ kemungkinan permutasi.

Total untuk $i = k$ dan $i \neq k$ terdapat $2(n-1)!$ kemungkinan permutasi, sedangkan *internal state* secara keseluruhan mempunyai $n!$ kemungkinan permutasi. Jadi probabilitas yang kita dapatkan adalah

$$\frac{2(n-1)!}{n!} = \frac{2}{n}.$$

Untuk bagian 2, kita buktikan bahwa untuk setiap $x \in \{0, 1, \ldots, 255\}$,

$$P(S[j] + S[k] \equiv c \pmod{n} \mid S[j] = x) = \frac{n-2}{n(n-1)}.$$

Kita bagi menjadi 3 kasus:

1. Kasus $i = k$. Karena $c \neq i = k$, maka tidak mungkin kondisi $k + S[k] \equiv c + S[i]$ tercapai, jadi kasus ini tidak memberi kontribusi terhadap penghitungan.

2. Kasus $c = k$. Kondisi $k + S[k] \equiv c + S[i] \pmod{n}$ juga tidak mungkin tercapai karena $S[k] \equiv S[i] \pmod{n}$ menjadi sesuatu yang tidak mungkin. Jadi kasus ini juga tidak memberi kontribusi terhadap penghitungan.

3. Kasus $i \neq k$ dan $c \neq k$. Untuk kasus ini, kita harus buat $S[k] = (i - x) \bmod n$ dan $S[i] = (k + S[k] - c) \bmod n$. Terdapat $n - 2$ pilihan untuk k dan sisanya (setelah i dan k terpilih) masih ada $(n-2)!$ kemungkinan permutasi. Jadi total ada $(n-2)(n-2)!$ kemungkinan permutasi.

Total terdapat $(n-2)(n-2)!$ kemungkinan permutasi, sedangkan *internal state* secara keseluruhan mempunyai $n!$ kemungkinan permutasi. Jadi probabilitas yang kita dapatkan adalah

$$\frac{(n-2)(n-2)!}{n!} = \frac{n-2}{n(n-1)}.$$

Selesailah pembuktian teorema 27.

Secara garis besar, PTW *attack* bertahap mencari $K[l]$ setelah mengetahui $K[0], K[1], \ldots, K[l-1]$. Karena $K[0], K[1], \ldots, K[l-1]$ diketahui, maka l putaran permutasi pertama menggunakan KSA dapat disimulasi. Kita gunakan notasi $S_m[x]$ untuk nilai $S[x]$ pada putaran m setelah permutasi dilakukan, dimana untuk putaran pertama $m = 1$. (Yang mungkin agak membingungkan adalah dalam putaran permutasi KSA, i mulai dari 0, sedangkan dalam putaran permutasi PRGA, i mulai dari 1.) Dengan simulasi, kita bisa dapatkan seluruh *internal state* S_l. Dalam putaran permutasi ke-$l+1$, nilai $S[l]$ dan nilai $S[(j + S[l] + K[l]) \bmod n]$ saling dipertukarkan. Tepatnya

$$S_{l+1}[l] = S_l[(j_l + S_l[l] + K[l]) \bmod n].$$

Untuk singkatnya, kita gunakan $t = S_{l+1}[l]$. Jika nilai t diketahui, karena simulasi bisa menghasilkan seluruh *internal state* S_l, maka nilai $K[l]$ dapat dicari dengan rumus

$$K[l] = (S_l^{-1}[t] - (j_l + S_l[l])) \bmod n$$

dimana $S_l^{-1}[t]$ adalah indeks untuk S_l yang memberikan t, dengan kata lain

$$S_l[S_l^{-1}[t]] = t.$$

Esensi dari PTW *attack* adalah mencari nilai t dari pengamatan output, yang kemudian digunakan untuk menentukan $K[l]$. Kita kaitkan t dengan output menggunakan persamaan

$$t \equiv l - X[l-1] \pmod{n}.$$

Berikutnya kita akan bahas probabilitas bahwa persamaan ini berlaku.

Setelah putaran $l+1$, maka terdapat $n-2$ putaran permutasi lagi sebelum i kembali mempunyai nilai l ($n-l-1$ putaran permutasi pada tahap KSA dan $l-1$ putaran permutasi pada tahap PRGA). Selama $n-2$ putaran itu, nilai $S[l]$ hanya akan berubah jika j menunjuknya. Setiap putaran, probabilitas $j = l$ adalah $1/n$, jadi probabilitas bahwa nilai $S[l]$ tidak berubah dalam $n-2$ putaran adalah

$$\left(1 - \frac{1}{n}\right)^{n-2}.$$

Probabilitas bahwa nilai $S[l]$ ditukar adalah $1 - (1 - 1/n)^{n-2}$. Untuk kasus dimana $S[l]$ tidak berubah ($S_{n+l-1}[l] = t$), menggunakan teorema 27 kita dapatkan probabilitas p_1 bahwa t yang didapat adalah benar

$$\begin{aligned} p_1 &= P(t \equiv l - X[l-1] \pmod n \mid t = S_{n+l-1}[l]) \\ &= P(S_{n+l-1}[l] + X[l-1] \equiv l \pmod n) \\ &= P(S_{n+l}[j_l] + S_{n+l}[k] \equiv i \pmod n) \\ &= \frac{2}{n}, \end{aligned}$$

dimana $i = l$, $S_{n+l}[j_l] = S_{n+l-1}[i]$ (karena permutasi), $X[l-1] = S_{n+l}[k]$, dan $k = S_{n+l}[i] + S_{n+l}[j_l]$. Untuk kasus $S_{n+l-1}[l] \neq t$ kita dapatkan probabilitas p_2 bahwa t yang didapat adalah benar

$$\begin{aligned} p_2 &= P(t \equiv l - X[l-1] \pmod n \mid t \neq S_{n+l-1}[l]) \\ &= P(t + X[l-1] \equiv l \pmod n \mid t \neq S_{n+l-1}[l]) \\ &= P(t + S_{n+l}[k] \equiv l \pmod n \mid t \neq S_{n+l-1}[l]) \\ &= P(S_{n+l-1}[l] + d + S_{n+l}[k] \equiv l \pmod n) \\ &= P(S_{n+l}[j_l] + S_{n+l}[k] \equiv i - d \pmod n) \\ &= P(S_{n+l}[j_l] + S_{n+l}[k] \equiv c \pmod n) \\ &= \frac{n-2}{n(n-1)}, \end{aligned}$$

dimana $i = l$, $S_{n+l}[j_l] = S_{n+l-1}[i]$ (karena permutasi), $X[l-1] = S_{n+l}[k]$, $d = t - S_{n+l-1}[l]$, $c = i - d \not\equiv i \pmod n$, dan $k = S_{n+l}[i] + S_{n+l}[j_l]$. Secara keseluruhan, probabilitas bahwa $t \equiv l - X[l-1] \pmod n$ adalah

$$\left(1 - \frac{1}{n}\right)^{n-2} \frac{2}{n} + \left(1 - \left(1 - \frac{1}{n}\right)^{n-2}\right) \frac{n-2}{n(n-1)} \approx \frac{1.36}{n}.$$

Hasil yang benar, meskipun prosentasinya kelihatan kecil, akan menonjol dibandingkan hasil-hasil yang salah yang masing-masing mempunyai probabilitas kurang dari $\frac{1}{n}$. Tentunya untuk mendapatkan hasil yang baik diperlukan *sample* yang cukup banyak. Untuk tingkat kesuksesan diatas 50 persen, diperlukan *sample* sebesar 43 ribu paket. Ini jauh lebih sedikit dibandingkan *sample* yang dibutuhkan FMS *attack* yaitu 9 juta paket. Untuk tingkat kesuksesan diatas 95 persen dibutuhkan *sample* sebesar 70 ribu paket. Jadi jelas PTW *attack* lebih efisien dibandingkan FMS *attack*.

Penggunaan RC4 dalam WEP mempunyai kelemahan yang menyebabkan WEP menjadi tidak aman. Kelemahan ini terutama berada pada bagian KSA, tetapi terdapat juga pada bagian PRGA. Kunci rahasia dapat terbongkar karena "jejak" dari kunci rahasia masih terdapat pada *keystream* dibagian awal.

PRGA harus berjalan lebih lama lagi sebelum "jejak" kunci rahasia menjadi sulit untuk terdeteksi. Itulah sebabnya dianjurkan agar bagian awal dari *keystream* (256 byte pertama) dibuang dan tidak digunakan untuk mengenkripsi, satu anjuran yang diikuti oleh WPA (Wi-Fi *protected access*). Tetapi meskipun bagian awal dibuang, karena terdapat kecenderungan pada PRGA, RC4 tetap memiliki kelemahan yang dapat dieksploitasi.

Banyak contoh lain penggunaan RC4 yang tidak aman. Jika tidak hati-hati, memang sangat mudah untuk menggunakan *stream cipher* secara tidak aman. Itulah sebabnya, tidak dianjurkan untuk menggunakan *stream cipher*, apalagi jika tidak paham dengan kelemahan algoritma yang digunakan. Daftar penggunaan RC4 secara tidak aman dimasa lalu sangat panjang. Semua produk Microsoft yang diketahui pernah menggunakan RC4, pernah menggunakannya secara tidak aman. Browser Netscape juga pernah mengimplementasikan RC4 secara tidak aman untuk keperluan SSL (*Secure Socket Layer*).

6.2 Ringkasan

Bab ini telah membahas *stream cipher* sebagai teknik enkripsi yang menyerupai enkripsi *one-time pad* dengan *keystream* yang dibuat menggunakan kunci enkripsi. Contoh yang dibahas adalah RC4 karena merupakan *stream cipher* yang banyak digunakan. Sangat mudah untuk menggunakan *stream cipher* secara tidak aman, contohnya penggunaan RC4 dalam WEP. Oleh sebab itu buku ini tidak merekomendasikan penggunaan *stream cipher*.