Studi Mengenai Algoritma Tiger Hash

Dewangga Respati – NIM : 13503120

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if13120@students.if.itb.ac.id

Abstrak

Selama ini, fungsi hash pada kriptografi yang tidak berbasis pada blok chiper, MD4 dan snefru terlihat cukup atraktif dan berguna untuk aplikasi yang membutuhkan kecepatan perangkat lunak untuk menjalankan hashing. Bagaimanapun juga kekurangan pada snefru akhirnya ditemukan pada tahun 1990, dan akhirnya kekurangan pada MD4 ditemukan juga akhir-akhir ini. Dengan ditemukannya kekurangan ini, semakin timbul keraguan akan kekuatan dari varian dari fungsi-fungsi hash lain seperti *RIPE-MD, MD5, SHA, SHA1 and Snefru-8* untuk bertahan.

Lebih jauh lagi, semua dari fungsi-fungsi di atas memang didesain untuk prosesor 32 bit, dan tidak dapat diimplementasikan secara efisien pada generasi prosesor yang baru yaitu prosesor 64 bit sepeti DEC Alpha. Karena alasan itu, dibuatlah algoritma fungsi hash yang baru yang diyakini sangat aman. Dan fungsi hash ini memang didesain untuk berjalan cepat pada prosesor 64 bit. Fungsi hash ini dinamakan Tiger Hash.

Kata Kunci: *Tiger Hash, hash, chiper, prosesor.*

1. Pendahuluan

Tiger adalah fungsi hash terbaru yang cepat, didesain untuk berjalan secara cepat pada komputer modern, dan secara khusus untuk komputer yang berbasis pada 64 bit (seperti DEC-Alpha), dan juga algoritma ini masih tidak lebih lambat dari fungsi hash lain yang disarankan dalam mesin 32-bit (meskipun sekarang sudah tidak lagi, semenjak MD5 dan SHA-1 telah ditemukan kelemahannya).

Dalam DEC-Alpha, Tiger melakukan hash lebih dari 132 Mbits per detiknya (diukur pada Alpha 7000, Model 660, dan dalam satu prosesor). Dalam mesin yang sama, MD5 melakukan hash dengan kecepatan hanya 37 Mbps (ini kemungkinan bukan kode dari MD5 yang paling optimal). Dalam mesin 32-bit, kode dari Tiger tidak berjalan dengan optimal. Tapi tetap saja Tiger melakukan hash lebih cepat daripada MD5 dalam 486s dan Pentiums. Kami memperkirakan bahwa Tiger menjalankan hash lebih cepat daripada MD5 bahkan pada mesin 16-bit.

Tiger tidak ada pembatasan penggunaan ataupun suatu paten tertentu. Tiger dapat digunakan secara gratis, dengan suatu referensi implementasi, dengan implementasi yang lain atau dengan modifikasi dari referensi implementasi (selama hal ini masih mengimplementasikan Tiger). Kami hanya meminta pembaca untuk mengetahui implementasi dan menyebutkan asal mula dari Tiger dan referensi implementasi dari Tiger.

Kami mendorong orang untuk mempelajari kekuatan dari algoritma Tiger, kami juga akan membahas tentang penyerangan terhadap Tiger beserta dengan analisisnya.

Dalam kriptografi, Tiger adalah suatu fungsi hash yang didesain oleh Ross Anderson dan Eli Biham pada tahun 1995 untuk kepentingan efisiensi pada platform 64-bit. Ukuran nilai hash dari Tiger adalah 192 bits. Versi 128-bits dan 160-bit dari algoritma ini juga ada, dikenal dengan nama Tiger/128 dan Tiger/160. Kedua variasi di atas menghasilkan nilai yang merupakan potongan dari nilai hash Tiger/192.

Tiger2 adalah variasi lain dari Tiger yang menggunakan *message-end padding* yang sama dengan MD5 dan SHA, dibandingkan dengan sikap melalaikan perbedaan *padding* yang digunakan pada MD4. Spesifikasi formal dari Tiger2 belum dirilis sampai saat ini, tapi tes vector sudah tersedia. Tiger didesain menggunakan oendekatan yang mirip dengan paradigma “Merkle-Damgard”. Fungsi kompresi menggunakan kombinasi campuran dari operasi XOR dan penambahan, perputaran, dan juga *S*-

box lookups. Tiger umumnya menggunakan bentuk *Merkle hash tree*, dimana pohon itu biasanya disebut dengan TTH (Tiger Tree Hash). TTH digunakan pada banyak klien pada jaringan *file sharing* "Direct Connect" dan "Gnutella". Tiger. Tiger dipertimbangkan juga untuk digunakan pada standart OpenPGP, tapi dibatalkan karena adanya kemunculan RIPEMD-160. Program "testtiger" yang ada pada *homepage* pemilik dimaksudkan untuk melakukan pengetesan yang mudah terhadap *source code*, dibandingkan untuk mendefinisikan urutan eksekusi. Sebenarnya spesifikasi dari Tiger sendiri tidak mendefinisikan cara dari keluaran Tiger itu dicetak, meskipun test vector untuk NESSIE menggunakan *quasi-standart* metode pengeprintan untuk hashnya dimana merupakan *big endian* urutan byte (urutan pencetakan ini juga digunakan oleh MD5 dan SHA1 untuk keluaran hashnya, sehingga pengguna dapat pula membaca nomornya dari kiri ke kanan).

Oleh karena itu, dalam contoh di bawah ini, hash Tiger yang 192-bit (24 byte) direpresentasikan sebagai 48-digit nomor hexadecimal dalam urutan byte big endian. Berikut ini mendemokan 43-byte masukan ASCII dan hash Tiger yang berhubungan :

```
Tiger("The quick brown fox jumps
over the lazy dog") =
6d12a41e72e644f017b6f0e2f7b44c62
85f06dd5d2c5b075

Tiger2("The quick brown fox
jumps over the lazy dog") =
976abff8062a2e9dcea3a1ace966ed9c
19cb85558b4976d8
```

Bahkan perubahan kecil terhadap pesan akan menghasilkan hash yang benar-benar berbeda, sebagai contoh kita mengubah dari abjad d menjadi c:

```
Tiger("The quick brown fox jumps
over the lazy cog") =
a8f04b0f7201a0d728101c9d26525b31
764a3493fcd8458f

Tiger2("The quick brown fox
jumps over the lazy cog") =
09c11330283a27efb51930aa7dc1ec62
4ff738a8d9bdd3df
```

Hash yang dihasilkan adalah :

```
Tiger("") =
3293ac630c13f0245f92bbb1766e1616
7a4e58492dde73f3

Tiger2("") =
4441be75f6018773c206c22745374b92
4aa8313fef919f41
```

2. Motivasi dan Design Requirement

Fungsi hash pada kriptografi memiliki peranan yang sangat penting dalam protokol kriptografi. Ketika digunakan dengan skema signature, peran dari fungsi hash adalah untuk mengurangi ukuran data yang harus ditandai dan untuk memecah beberapa properti seperti *multiplicative homomorphism* yang kemungkinan akan dieksploitasi oleh saingan kita. Secara singkat, fungsi hash diharapkan memiliki dua *requirement* penting yaitu efisien dan aman; dan dalan banyak aplikasi komersil, fungsi hash diharapkan dapat berjalan dengan cepat dalam perangkat lunak di semua platform hardware yang umum digunakan.

Beberapa fugsi hash berbasis pada mode umpan balik yang maju terhadap block cipher, tapi pesaing utamanya adalah gungsi yang berbasis pada MD4, yang di dalamnya mengandung MD5, RIPE-MD, SHA, dan SHA-1. Keluarga hash yang lain adalah Snefru dan turunannya yang disebut Snefru-8. Namun, kelemahan pada Snefru ditemukan pada 1990 dan akhirnya kelemahan pada MD4 pun juga ditemukan. Serangan ini membuat timbulnya keraguan akan keamanan dari anggota keluarga algoritma ini yang lain. Salah satunya mungkin hanya spekulasi pada berapa lama lagi setiap fungsi akan bertahan dari penyerangan, meskipun juga akan sangat menguntungkan jikakita mulai bekerja untuk menemukan pengganti dari algoritma yang telah ada.

Dari sisi performansi, semua fungsi yang telah disebutkan di atas didesain hanya untuk prosesor dengan 32-bit. Generasi selanjutnya dari prosesor memiliki 64-bit, dan termasuk seri DEC-Alpha demikian juga dengan prosesor yang akan dirilis oleh Intel, HP, dan IBM. Mungkin cukup beralasan untuk mengasumsikan bahwa mayoritas sistem akan menggunakan prosesor 64-bit selama lima tahun mendatang dengan perkecualian terhadap microcontroller yang digunakan pada *embedded application*.

Meskipun, dalam beberapa prosesor, fungsi hash pada keluarga algoritma di atas tidak dapat diimplementasikan dengan efisien. Sebagai contoh, keluarga dari MD menggunakan rotasi dan penambahan terhadap banyak 32-bit, dan register 64-bit hanya dapat menangani satu nilai 32-bit dalam sekali waktu, yang secara otomatis akan mengurangi kecepatan potensialnya dengan faktor perkalian dua. Lebih jauh lagi, arsiterktur dari Alpha tidak mempunyai segala macam operasi rotasi, baik terhadap 32-bit maupun 64-bit. Dari pertimbangan di atas, kami percaya bahwa fungsi hash generasi selanjutnya akan mempunyai beberapa spesifikasi yang harus dipenuhi, antara lain :

- Harus aman, setidaknya harus bebas dari *collision* dan *multiplication.*
- Harus berjalan dengan cepat pada 64-bit, dan juga tidak berjalan dengan lebih lambat pada mesin dengan prosesor 32-bit seperti pada Intel's 80486.
- Harus dapat diandalkan untuk menggantikan algoritma pendahulunya, yaitu MD4, MD5, SHA, dan SHA-1.

3. Proposal Pengajuan Algoritma Tiger

Dalam papernya, pemilik algoritma Tiger mengusulkan suatu fungsi hash yang baru, dan dinamakannya Tiger, sesuai dengan sifatnya yang kuat dan cepat: secepat SHA-1 pada prosesor 32-bit dan sekitar tiga kali lebih cepat dalam prosesor 64-bit (DEC-Alpha). Dan juga diharapkan dapat lebih cepat dibandingkan dengan SHA-1 pada prosesor 16-bit, meskipun SHA-1 dioptimasikan untuk penggunaan pada mesin 32-bit. Oleh karena itu, algoritma yang diusulkan didesain untuk bekerja dengan baik pada semua jenis prosesor.

Operasi utama yang dikerjakan adalah empat tabel S-box, masing terdiri antara 8 sampai 64 bit. Dalam mesin 32-bit hal ini dapat diimplementasikan sebagai pasangan dari tabel, dengan perhitungan offset hanya dilakukan sekali. Operasi lain yang dikerjakan adalah penambahan 64-bit dan substraksi, multiplikasi 64-bit dengan konstanta yang kecil (5,7, dan 9), 64-bit shifts dan operasi logika seperti XOR dan NOT. Semua operasi ini paling parah dua kali lebih lambat pada mesin 32-bit, dengan pengecualian pada operasi shifts dan multiplikasi dengan konstanta yang kecil dimana bisa mencapai empat atau lima kali lebih lambat. Untuk kepentingan *compatibility,* Tiger mengadopsi struktur umum dari keluarga MD4 : pesan diberi padding sebanyak satu bit '1' diikuti oleh string 0 dan akhirnya panjang dari pesan menjadi 64-bit. Hasilnya akan dibagi menjadi menjadi n buah 512 bit blok.

Ukuran dari nilai hash dan intermediate state adalah 3 words atau 192 bit. Nilai ini dipilih karena beberapa alasan, antara lain :

1. Jika kita menggunakan 64-bit words, ukurannya haruslah kelipatan dari 64.
2. Agar bisa kompatibel dengan aplikasi yang menggunakan SHA-1, ukuran dari hash harus paling tidak 160 bit.
3. Semua serangan yang diterapkan pada fungsi hash yang telah ada menyerang pada intermediate state, dibandingkan dengan menyerang pada nilai akhir dari hash. Penyerang biasanya memilih dua nilai untuk intermediate blok, dan hal ini menyebarkan ke suatu bentrokan dari semua fungsi yang ada. Bagaimanapun juga serangan ini tidak akan sukses jika nilai hash intermediate yang digunakan lebih besar.

Tiger dengan keluaran 192-bit biasa juga disebut Tiger/192, dan kami merekomendasikan untuk digunakan pada semua aplikasi baru yang ada. Ketika menggantikan fungsi lain pada aplikasi yang telah ada, penulis Tiger menyarankan dua variasi lain :

1. Tiger/160: nilai hash adalah 160 bit pertama dari hasil Tiger/192, dan digunakan untuk kompatibilitas dengan SHA dan SHA-1.
2. Tiger/128: nilai hash adalah 128 bit dari hasil Tiger/192, dan digunakan untuk kompatibilitas dengan MD4, MD5, RIPE-MD, variasi dari Snefru dan beberapa fungsi hash berbasis pada blok cipher.

Penulis Tiger memperkirakan bahwa tiga variasi dari Tiger bebas dari *collision*, oleh karena itu *collision* untuk Tiger/N tidak dapat ditemukan dengan usaha yang bernilai $O(2^{N/2})$. Penulis juga mempercayai bahwa algoritma ini *one-way* dan bebas dari multiplikasi. Efisiensi dari fungsi ini berdasarkan pada desainnya yang paralel. Dalam keluarga MD dan Snefru, setiap operasi bergantung secara langsung pada hasil dari operasi yang sebelumnya, dan juga prosesor RISC tidak dapat digunakan secara efisien dengan alasan pipeline. Dalam setiap putaran dalam Tiger, operasi *lookup* terhadap delapan tabel dapat dilaksanakan secara paralel, sehingga

compiler dapat membuat penggunaan yang optimal bagi pipeline. Desainnya juga memungkinkan implementasi perangkat keras yang efisien.

Ukuran memori yang dibutuhkan oleh Tiger tidak jauh beda dengan ukuran dari empat S box. Jika hal ini dapat diakomodasi dalam cache dari prosesor, komputasi akan berjalan dua kali lebih cepat (diukur pada DEC Alpha). Ukuran dari empat S box adalah 4x256x8 = 8096 = 8Kbytes, dimana ini merupakan ukuran dari cache pada kebanyakan mesin yang ada. Jika delapan S box yang digunakan, 16 Kbytes diperlukan, dan ini merupakan dua kali dari ukuran cache pada Alpha.

4. Spesifikasi Algoritma Tiger

Dalam Tiger, semua komputasi dilakukan pada 64-bit words, dalam representasi little endian . Tiger menggunakan tiga register 64-bit yang disebut sebagai a,b, dan c sebagai nilai hash intermediate. Register-register ini diinisialisasi ke komputasi h_0 dimana :

```
a = 0x0123456789ABCDEF
b = 0xFEDCBA9876543210
c = 0xF096A5B4C3B2E187
```

Setiap 512-bit blok pesan sebelumnya dipecah menjadi delapan 64-bit words x0, x1, …, x7, dan komputasinya berubah dari h_i menjadi h_{i+1}. Komputasi ini mengandung tiga tahapan dan di antara tahapan tersebut terdapat *key schedule* --- suatu transformasi dari data masukan yang mencegah para penyerang memaksakan masukan yang mirip pada tiga putaran tadi. Dan akhirnya ada suatu tahapan *feedforward* dimana nilai baru dari a, b, dan c dikombinasikan dengan nilai inisial untuk memberikan nilai h_{i+1}.

```
save_abc
pass(a,b,c,5)
key_schedule
pass(c,a,b,7)
key_schedule
pass(b,c,a,9)
feedforward
```

dimana

1. save_abc menyimpan nilai dari h_i

```
aa = a ;
bb = b ;
cc = c ;
```

2. pass (a,b,c,mul) adalah

```
round(a,b,c,x0,mul);
round(b,c,a,x1,mul);
round(c,a,b,x2,mul);
round(a,b,c,x3,mul);
round(b,c,a,x4,mul);
round(c,a,b,x5,mul);
round(a,b,c,x6,mul);
round(b,c,a,x7,mul);
```

dimana round (a,b,c,x,mul) adalah

```
c ^= x ;
a -= t1[c_0] ^ t2[c_2] ^
t3[c_4] ^ t4[c_6] ;
b += t4[c_1] ^ t3[c_3] ^
t2[c_5] ^ t1[c_7] ;
b *= mul;
```

dan dimana c_i adalah bit ke I dari c (0<i<7). Misalkan kita menggunakan notasi dari bahasa pemrograman C, dimana ^ menggambarkan operasi XOR, dan notasi X op= Y berarti bahwa X = X op Y, untuk semua operator op. S box t1 sampai dengan t4 akan menghabiskan sepuluh halaman untuk mempublikasikannya di sini. Oleh karena itu hal ini akan dipublikasikan secara elektronik bersamaan dengan source code, dan semuanya tersedia pada homepage dari penulis.

3. keyschedule adalah

```
x0 -= x7 ^
0xA5A5A5A5A5A5A5A5;
x1 ^= x0;
x2 += x1;
x3 -= x2 ^ ((~x1)<<19);
x4 ^= x3;
x5 += x4;
x6 -= x5 ^ ((~x4)>>23);
x7 ^= x6;
x0 += x7;
x1 -= x0 ^ ((~x7)<<19);
x2 ^= x1;
x3 += x2;
x4 -= x3 ^ ((~x2)>>23);
x5 ^= x4;
x6 += x5;
x7 -= x6 ^
0x0123456789ABCDEF;
```

dimana << dan >> adalah operasi logika shift left dan shift right.

4. feedforward adalah

```
a ^= aa ;
b -= bb ;
c += cc ;
```

Register a, b, c yang dihasilkan adalah 192 bit dari (intermediate) nilai value h_{i+1}.

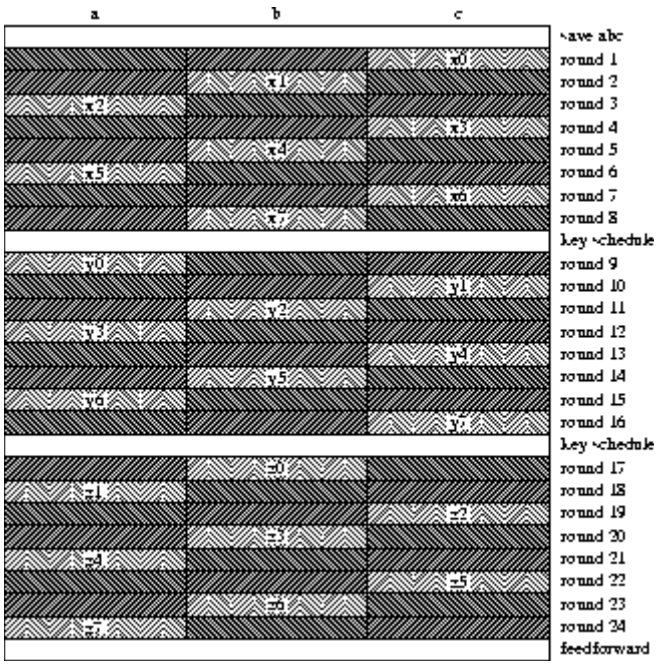

Figure 1: Gambaran dari fungsi kompresi Tiger.

Gambar di atas mendeskripsikan fungsi kompresi yang digunakan oleh Tiger. Dalam gambar di atas, area hitam menggambarkan register yang terkena efek, dimana garis miring menunjuk kepada byte yang terkena efek pada area putih. Variable y0, y1, ..., y7 dan z0, z1, ..., z7 mendefinisikan nilai dari x0, x1, ..., x7 dalam fase kedua dan ketiga. Dan akhirnya, nilai intermediate hash yang terakhir diambil dari hasil keluaran Tiger/192.

5. Keamanan Algoritma Tiger

Ada beberapa faktor keamanan yang dipertimbangkan dalam penemuan algoritma Tiger ini, antara lain:

1. Prinsip *nonlinearity* muncul kebanyakan dari S-box dengan jumlah 8 sampai 64 bit. Hal ini lebih baik dibandingkan jika kita hanya mengkombinasikan operasi penambahan dan operasi XOR (menggunakan *carry bits*), dan hal ini mengakibatkan efek pada semua bit keluaran bukan hanya bit tetangga saja.
2. Terdapat semacam longsoran yang kuat, dalam setiap bit pesan akan mengakibatkan efek semua tiga register setelah tiga putaran – lebih cepat daripada fungsi hash yang lain. Longsoran ini pada 64-bit words (dan 64-bit S box) berjalan dengan lebih cepat dibandingkan jika word yang lebih pendek digunakan.
3. Seperti yang telah dijelaskan di atas, semua serangan pada MD*/Snefru menargetkan pada salah satu blok intermediate, Dengan menaikkan nilai intermediate menjadi 192 bit membantu menghalangi terjadinya serangan ini.
4. *Key Schedule* memastikan bahwa perubahan pada sedikit jumlah bit pada pesan dapat mengakibatkan efek pada banyak bit yang muncul selama fase-fase selanjutnya. Bersama dengan point kedua di atas, hal ini membantu Tiger untuk melawan serangan yang sama dengan *Dobbertin's differential attack* pada MD4 (dimana merubah beberapa bit tertentu pada pesan dapat mengakibatkan efek pada paling banyak dua bit pada banyak putaran, dan perubahan kecil ini dapat dibuat untuk dibatalkan pada putaran terakhir).
5. Multiplikasi dari register b dalam setiap putaran juga berkontribusi untuk pertahanan terhadap serangan d atas, asalkan dipastikan bahwa bit yang dipakai sebagai masukan untuk S-box dalam putaran sebelumnya dicampurkan ke dalam S-box yang lain dengan baik, dan ke S-box yang sama dengan masukan yang berbeda. Multiplikasi ini juga mencegah *related-key attacks* pada fungsi hash, asalkan konstanta yang dipakai berbeda pada setiap putaran.
6. Feedforward yang digunakan dapat mencegah *meet-in-the-middle birthday attack* yang menemukan gambaran umum dari fungsi hash (meskipun kompleksitasnya akan menjadi sangat besar)

6. Rangkuman Algoritma Tiger

Dalam *paper* yang diajukan, penulis mengajukan suatu konsep fungsi hash yang baru, yang disebut dengan Tiger, dimana algoritma ini didesain untuk lebih cepat dan aman. Inti prosesnya adalah tiga putaran, dimana setiap putaran menggunakan delapan *lookups* menjadi 8 sampai 64 bit S-box untuk menyediakan semacam longsoran nonlinear yang kuat dan juga beberapa operasi register untuk meningkatkan *diffusion* dan membuat *differential attacks* menjadi lebih sulit.

Tiger dapat diimplementasikan dengan efisien pada mesin 32-bit dan 64-bit. Tiger sama cepatnya dengan algoritma pendahulunya yaitu SHA 1, tapi tidak seperti SHA 1, Tiger dapat menggunakan kekuatan penuh dari mesin 64-bit, dimana dapat diperkirakan sekitar 2,5 kali lebig cepat daripada SHA 1.

Tiger mengeluarkan 192-bit nilai hash. Untuk kompatibilitas dengan fungsi hash yang telah ada, Tiger menyarankan bahwa keluarannya dapat dipotong menjadi 160 atau 128 bit jika diperlukan untuk kompatibilitas dengan aplikasi yang telah ada. Tiger dapat dipercaya bahwa variasi ini akan lebih aman jika dibandingkan dengan fungsi yang tersedia untuk ukurang panjang keluaran yang sama; meskipun jika ada permintaan untuk menambah beberapa fase pada Tiger, dan hal itu dilakukan, dan Tiger menyarankan suatu konstanta multiplikasi 9 pada setiap fase tambahan. Penulis menyebutnya suatu varian TigerM, atau TigerM/N, dimana M adalah adalah jumlah fase yang digunakan dan N adalah jumlah bit dalam nilai hash.

Seperti biasanya ketika menyarankan suatu primitif kriptografi yang baru, penulis menyarankan orang-orang untuk mempelajari kekuatan dari Tiger; penulis akan menghargai serangan, analisis, dan komentar lain jika memang ada. Untuk informasi status Tiger saat ini ataupun juga info pengembangan Tiger lebih lanjut dapat ditemukan pada homepage penulis yaitu http://www.cs.technion.ac.il/~biham/ dan http://www.cl.cam.ac.uk/users/rja14/.

7. Algoritma Umum Fungsi Kompresi Tiger

```
word64 t1[256] = {...};
word64 t2[256] = {...};
word64 t3[256] = {...};
word64 t4[256] = {...};

TIGER_compression_function
(state, block)
word64 state[3];
unsigned word64 block[8];
{
  word64 a = state[0], b =
state[1], c = state[2];
  word64 x0=block[0],
x1=block[1], x2=block[2],
x3=block[3],
         x4=block[4],
x5=block[5], x6=block[6],
x7=block[7];
  word64 aa, bb, cc;

#define save_abc aa = a; bb = b;
cc = c;

#define round(a,b,c,x,mul) \
      c ^= x; \
      a -= t1[((c)>>(0*8))&0xFF]
^ t2[((c)>>(2*8))&0xFF] ^ \
           t3[((c)>>(4*8))&0xFF]
^ t4[((c)>>(6*8))&0xFF] ; \
      b += t4[((c)>>(1*8))&0xFF]
^ t3[((c)>>(3*8))&0xFF] ^ \
           t2[((c)>>(5*8))&0xFF]
^ t1[((c)>>(7*8))&0xFF] ; \
      b *= mul;
#define pass(a,b,c,mul) \
      round(a,b,c,x0,mul) \
      round(b,c,a,x1,mul) \
      round(c,a,b,x2,mul) \
      round(a,b,c,x3,mul) \
      round(b,c,a,x4,mul) \
      round(c,a,b,x5,mul) \
      round(a,b,c,x6,mul) \
      round(b,c,a,x7,mul)

#define key_schedule \
      x0 -= x7 ^
0xA5A5A5A5A5A5A5A5; \
      x1 ^= x0; \
      x2 += x1; \
      x3 -= x2 ^ ((~x1)<<19); \
      x4 ^= x3; \
      x5 += x4; \
      x6 -= x5 ^ ((~x4)>>23); \
      x7 ^= x6; \
      x0 += x7; \
```

```
      x1 -= x0 ^ ((~x7)<<19); \
      x2 ^= x1; \
      x3 += x2; \
      x4 -= x3 ^ ((~x2)>>23); \
      x5 ^= x4; \
      x6 += x5; \
      x7 -= x6 ^
0x0123456789ABCDEF;

#define feedforward a ^= aa; b -
= bb; c += cc;

#define compress \
      save_abc \
      pass(a,b,c,5) \
      key_schedule \
      pass(c,a,b,7) \
      key_schedule \
      pass(b,c,a,9) \
      feedforward

  compress;

  state[0] = a; state[1] = b;
state[2] = c;
}
```

8. Kriptanalisis Terhadap Tiger

8.1. Pendahuluan

Dalam dua tahun terakhir ini, banyak hasil dari kriptanalis yang menemukan kelemahan pada hampir sebagian besar fungsi hash yang selama ini digunakan. Dilihat dari sisi desain, semua dari fungsi hash (termasuk MD5, RIPEMD, SHA0 dan SHA1) diturunkan dari fungsi MD4. Hal ini mendorong ketertarikan untuk menemukan alternatif desain fungsi hash lain yang benar-benar beda dengan fungsi hash yang selama ini, yang telah ditemukan cara penyerangannya oleh banyak kriptanalis. Salah satu alternatif konstruksi yang ditawarkan adalah tiger, yang didesain oleh Anderson dan Biham pada tahun 1996. Mirip dengan turunan dari MD4, Tiger melakukan iterasi suatu fungsi kompresi internal untuk melakukan hashing pesan panjang secara dinamis yaitu berubah-ubah tiap iterasi. Bagaimanapun juga, fungsi kompresi Tiger bisa dikatakan sangat berbeda dari fungsi kompresi dari keluarga MD4. Karena fungsi kompresi yang sangat berbeda secara internal, serangan terhadap keluarga MD4 akan berjalan tidak dengan semestinya jika digunakan langsung untuk melakukan serangan ke Tiger. Analisis kami menuju ke sesuatu informasi tambahan yang penting – teknik modifikasi pesan yang kita gunakan berbeda secara signifikan dengan yang kami gunakan pada MD4. Bagaimanapun juga, kita menggunakan teknik modifikasi pesan terhadap Tiger untuk tujuan yang sama seperti yang kita gunakan pada MD4 – untuk mengontrol perbedaan dalam beberapa putaran pertama dengan adanya pilihan terhadap nilai dari pesan, meskipun perbedaan pesan kami paksakan oleh analisis kami terhadap jadwal dari pesan. Tiger tampaknya membatasi pesan maksimum menjadi 2^{64} bit, berdasarkan pada ukuran dari *message length field* muncul secara langsung sehingga dapat dipakai sebagai pendekatan kita dalam melakukan serangan kepada Tiger. Di sisi lain, ini adalah pertanda yang memberi harapan; secara tidak langsung, hal ini menegaskan bahwa kita mempunyai harapan untuk mengambil teknik penyerangan yang diterapkan terhadap keluarga dari MD4, dan menerapkannya, dan menyesuaikannya, untuk membangun fungsi hash yang berbeda secara internal.

Dibawah ini, saya akan mendeskripsikan *collisio-finding attack* terhadap Tiger dikurangi menjadi 16 putaran. Seperti kita ketahui, Tiger beroperasi penuh sebanyak 24 putaran, serangan ini mengenai sebanyak dua pertiga dari Tiger, dengan kerja yang ekivalen dengan pembangkitan 2^{44} fungsi kompresi. Iger menghasilkan 192-bit hash, jadi collision akan secara ideal mengambil pembangkitan fungsi sebanyak 2^{96}. Demikian juga, saya mendeskripsikan serangan untuk memilih dua masukan berantai yang dengan jarak *Hamming* yang kecil (biasanya 6 bit), dimana menghasilkan *near-colliding* keluaran fungsi kompresi dengan jarak *Hamming* yang sama – meskipun pada kenyataannya *differential pattern* yang sama digunakan sebagai input. Serangan kedua ini menyerang pada lebih dari 80% dari Tiger (20 dari 24 putaran), dengan kerja yang ekivalen dengan kurang dari 2^{48} pembangkitan fungsi kompresi. 192-bit hash yang ideal diperlukan untuk mendukung hal di atas.

$$\sqrt{2^{192}\Big/\binom{192}{6}} \approx 2^{80} \quad \text{compression function invocations}$$

Untuk *near-collisions* dengan enam bit dari jarak Hamming, dibandingkan jika $< 2^{48}$.

Penjelasan selanjutnya dari bahasan ini adalah sebagai berikut. Subbab selanjutnya akan membahas deskripsi dari algoritma Tiger, secara detail untuk membahas serangan yang dilakukan. Subbab berikutnya akan membahas tentang *collision attack* dan mendeskripsikan detilnya. Dua subbab berikutnya akan membahas inti dari serangan yang dilakukan terhadap Tiger: Teknik Modifikasi Pesan. Subbab berikutnya akan membahas serangan *pseudo-near-collision* terhadap 20 putaran dari Tiger. Subbab terakhir akan membahas kesimpulan tentang keamanan dari Tiger berdasarkan serangan yang telah dikenakan pada Tiger.

8.2. Deskripsi High Level Tiger Hash

Fungsi kompresi dari Tiger berbasis pada menerapkan fungsi internal ”block cipher like”, yang mengambil 192-bit ”plaintext” dan 512-bit kunci untuk menghitung 192-bit ”ciphertext”. Fungsi ”block cipher like” diterapkan berdasarkan konstruksi Davies-Meyer: 512-bit blok pesan digunakan sebagai kunci untuk melakukan enkripsi nilai 192-bit yang berantai, dan kemudian masukan nilai berantai dimasukkan ke depan untuk membuat semua fungsi tidak dapat dibalik. Dalam sisa bahasan dalam subbab ini, saya akan mendeskripsikan Tiger dengan cukup detail untuk mengikuti alur dari teknik kriptanalis yang digunakan. Harus ditegaskan bahwa jika untuk setiap nilai berantai masukan, kita dapat menemukan *collision* dari Tiger. Tiger didesain dengan arsiterktur 64-bit secara asumsi. Oleh karena itu, kita akan mendefinisikan 64-bit *unsigned integer* sebagai istilah ”word”. Penulis akan merepresentasikan word sebagai angka heksadesimal. Tiger menggunakan operasi aritmetik (penambahan, pengurangan, pengalian dengan konstanta kecil), bit-wise XOR, NOT, operasi logika shift, dan aplikasi S-box. Operasi aritmatika terhadap word adalah merupakan modulo 2^{64}. Nilai berantai direpresentasikan secara internal dengan tiga word 64-bit, blok pesan sendiri adalah 64-bit word. Kemudian, tiga word A,B,C mendeskripsikan nilai masukan berantai dan delapan word pesan X_0, ..., X_7 dimasukkan ke dalam fungsi kompresi, dimana akan mengeluarkan tiga word A’,B’,C’ mendeskripsikan nilai keluaran yang berantai. Keluaran final dari fungsi kompresi A”,B”,C” dikeluarkan oleh fungsi feedforward :

$$A'' := A \oplus A',$$
$$B'' := B - B', \text{ and}$$
$$C'' := C + C'.$$

8.2.1. Tiger Round Function

Dalam terminologi, blok cipher dari Tiger mirip dengan fungsi dalam “target-heavy unbalanced Feistel Cipher”. Blok dibagi menjadi tiga words, yang diberi label A, B, dan C. dalam setiap putaran, word pesan X di-XOR dengan C:

$$C := C \oplus X.$$

Sehingga A dan B dimodifikasi menjadi:

$$A := A - \mathbf{even}(C),$$
$$B := B + \mathbf{odd}(C),$$
$$B := B * (\text{const}),$$

Hasilnya akan dikenakan operasi shift lagi, sehingga A, B, C akan menjadi B, C, A. Perhatikan pada Gambar 1 di bawah ini:

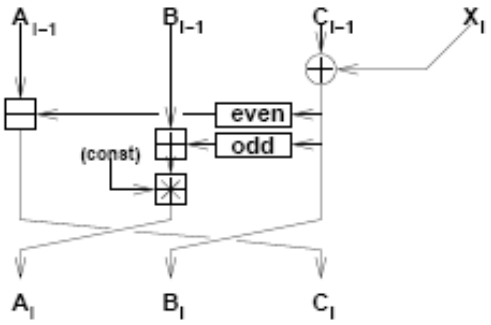

Fig. 1. The round function of Tiger

Untuk definisi dari even dan odd, coba pikirkan untuk memecah word C menjadi 8 byte C[0], ..., C[7], dengan *most significant byte* adalah C[0]. Fungsi even dan odd menerapkan empat S-box T1, ..., T4 : $\{0,1\}^8 \rightarrow \{0,1\}^{64}$ dengan aturan berikut:

$$\mathbf{even}(C) := T_1(C[0]) \oplus T_2(C[2]) \oplus T_3(C[4]) \oplus T_4(C[6]) \quad \text{and}$$
$$\mathbf{odd}(C) := T_1(C[7]) \oplus T_2(C[5]) \oplus T_3(C[3]) \oplus T_4(C[1]).$$

Byte “even” dan byte “odd” dari word W didefinisikan sebagai:

$$W[\mathbf{even}] = (W[0], W[2], W[4], W[6]) \in (\{0,1\}^8)^4 \quad \text{and}$$
$$W[\mathbf{odd}] = (W[7], W[5], W[3], W[1]) \in (\{0,1\}^8)^4.$$

Fungsi putaran di atas menyebarkan perubahan dengan sangat cepat – satu bit perbedaan yang dikenakan pada C dalam putaran pertama akan mengubah sekitar setengah bit dari blok dalam akhir putaran ketiga. Tiger memperlihatkan bahwa dia lebih unggul dalam hal ini bila dibandingkan dengan anggota dari keluarga MD4. Sangat mudah untuk memproduksi *local collisions* untuk fungsi perputaran pada Tiger, dengan menggunakan semacam pattern (α, β, 0, ά). Dimana, α adalah masukan yang berbeda untuk bit genap dari S-box, β adalah perbedaan XOR yang akan membatalkan hasil dari perbedaan pada fungsi genap, dan ά adalah α dikalikan dengan konstanta, diharapkan untuk membatalkan perubahan yang asli dari α. Bagaimanapun juga, *local collision* dalam bentuk ini akan sangat sulit untuk digunakan dalam serangan lebih dari delapan putaran dari Tiger – *key schedule* tampaknya akan sangat efektif untuk menghancurkan pattern di atas.

8.2.2. The Key Schedule

Tiger mengandung 24 putaran. Dalam setiap putaran menggunakan satu word pesan Xi sebagai kunci dalam putaran tersebut. Delapan kunci putaran pertama X0, . . . ,X7 identik dengan 512-bit kunci cipher (atau dengan 512-bit blok pesan). Enam belas kunci putaran sisanya dihasilkan dengan menerapkan suatu fungsi *key schedule:*

$$(X_8, \ldots, X_{15}) := \mathrm{KeySchedule}(X_0, \ldots, X_7)$$
$$(X_{16}, \ldots, X_{23}) := \mathrm{KeySchedule}(X_8, \ldots, X_{15})$$

Key schedule menggunakan logika shift terhadap word, dinotasikan dengan << dan >>, misalkan:

- $1111\,5555\,9999\,\mathrm{FFFF} \ll 5 = 222\mathrm{A}\,\mathrm{AAB3}\,333\mathrm{F}\,\mathrm{FFE0}$, and
- $222\mathrm{A}\,\mathrm{AAB3}\,333\mathrm{F}\,\mathrm{FFE0} \gg 9 = 0011\,1555\,5999\,9\mathrm{FFF}$.

Lebih jauh lagi, Tiger menggunakan fungsi bit-wise NOT, misalkan untuk X = EEEE AAAA 6666 0000, negasi dari X adalah 1111 5555 9999 FFFF. *Key schedule* memodifikasi inputnya (x0, . . . , x7) dalam dua fase:

first pass	second pass
1. $x_0 := x_0 - (x_7 \oplus \mathrm{Const}_1)$	9. $x_0 := x_0 + x_7$
2. $x_1 := x_1 \oplus x_0$	10. $x_1 := x_1 - (x_0 \oplus (\overline{x_7} \ll 19))$
3. $x_2 := x_2 + x_1$	11. $x_2 := x_2 \oplus x_1$
4. $x_3 := x_3 - (x_2 \oplus (\overline{x_1} \ll 19))$	12. $x_3 := x_3 + x_2$
5. $x_4 := x_4 \oplus x_3$	13. $x_4 := x_4 - (x_3 \oplus \overline{x_2} \gg 23))$
6. $x_5 := x_5 + x_4$	14. $x_5 := x_5 \oplus x_4$
7. $x_6 := x_6 - (x_5 \oplus (\overline{x_4} \gg 23))$	15. $x_6 := x_6 + x_5$
8. $x_7 := x_7 \oplus x_6$	16. $x_7 := x_7 - (x_6 \oplus \mathrm{Const}_2)$

Nilai akhir (x0, . . . , x7) digunakan sebagai keluaran dari *key schedule*. Konstantanya adalah Const1 = A5A5 . . .A5A5 and Const2 = 0123 . . .CDEF.

8.3. Serangan Terhadap Tiger

Ada kriptanalis yang mengajukan *differential attack* terhadap Tiger dalam tiga bagian. Melalui serangan itu, kriptanalis menukar antara *XOR-difference* dan difference dalam penambahan. Secara umum, menukar antara *differences* menghasilkan suatu peluang yang nilainya tidak nol; sebagai contoh, *addictive-difference* dari 1 dapat direpresentasikan sebagai *XOR-difference* dari 1, dengan nilai kemungkinan ½ menuju benar.

8.3.1. Conventions

Mengubah suatu *difference* ke suatu bentuk yang lain mungkin untuk dilakukan, tapi untuk beberapa nilai tertentu, hal ini mempunyai nilai kemungkinan satu.

- Jika X – Y = 2^i, maka Pr[X + Y = 2^i] = ½. Dengan pengecualian pada i = 63, dimana nilainya menjadi 1.
- Jika I = 2^{63}. menukar antara *additive-difference* I, *XOR-difference* dari I akan sukses dengan nilai probabilitas satu. Dengan kata lain, jika berusrusan dengan *difference* I, kita tidak peduli dengan tipe dari *difference* yang sedang digunakan. Serangan kita akan berguna pada semacam fakta yang simpel.
- Harus diperhatikan bahwa *difference* I dalam word W menyisakan hal yang sama, bahkan jika W dikalikan oleh beberapa konstanta yang bernilai ganjil, seperti yang dilakukan pada fungsi kompresi dari Tiger.

Kami memulai menghitung putaran dari nol, dan kami menulis Xi sebagai word pesan masukan pada putaran ke-i, dan Ai,Bi,Ci untuk keluaran pada putaran ke-i – dimana ini merupakan nilai

yang berantai untuk putaran yang ke-i+1. Berdasarkan hal di atas, nilai masukan untuk putaran ke-0 (putaran pertama) adalah A−1,B−1,C−1. *Difference* pada word pesan kebanyakan terlihat pada *XOR-difference,* asalkan word pesan (atau kunci putaran) Xi telah dikenakan operasi XOR pada saat status tersebut. *Additive-difference* adalah apa yang kita butuhkan untuk mengetahui kapan harus berhadapan dengan dua word target dalam suatu putaran (dua word yang akan diubah), karena *difference* aritmatika semuanya adalah merupakan mod 2_{64}. Untuk masukan S-Box dalam langkah modifikasi pesan, *XOR-differences* sangat berguna. Untuk seterusnya, saya akan menggunakan notasi berikut untuk mendefinisikan *differences* yang dialami oleh word W:

- $\Delta^{+}(W) = W - W^{*} \bmod 2^{64}$ for additive differences and
- $\Delta^{\oplus}(W) = W \oplus W'$ for word-wise differences.

8.3.2. Garis Besar Serangan

Serangan yang dilakukan dapat dipecah menjadi tiga bagian:

1. Karakteristik differential (I, I, I, I, 0, 0, 0, 0) → (I, I, 0, 0, 0, 0, 0, 0) dalam *key schedule*.
2. Karakteristik differential (I, I, 0) → (0, 0, 0) dalam putaran 6-9 dari fungsi putaran. (Karena word pesan dalam putaran 10-15 tetap tidak berubah, hal ini menyebabkan *collision* setelah putaran ke-16).
3. Modifikasi pesan untuk memaksa *difference* dalam fungsi putaran setelah putaran ke-6 menjadi (I, I, 0).

8.3.3. Key Schedule Differences

Pertimbangkan jika ada perbedaan dari bentuk (I, I, I, I, 0, 0, 0, 0) dalam word pesan. Fase pertama dari *key-schedule* mengubah bentuk ini menjadi *intermediate difference pattern* (I, 0, I, 0, 0, 0, 0, 0, 0). Pada fase kedua mengubah bentuk tadi menjadi (I, I, 0, 0, 0, 0, 0, 0). Ini adalah pattern *differential* yang akan kita gunakan untuk serangan kita; ini akan memberikan nilai kemungkinan satu, dan mengover word pesan tambahan yang digunakan dalam putaran 0-15. *Colliding messages* akan berbeda hanya dalam bit order yang tinggi untuk empat word yang pertama. Word pesan tambahan akan berbeda untuk putaran ke-8 dan ke-9, hanya dalam bit order yang tinggi.

Word pesan tambahan 10-15 tidak akan mempunyai perbedaan. Hal ini berarti jika status dari fungsi kompresi memproses dua pesan akan menjadi sama setelah putaran ke-9, dia akan bertahan sama sampai akhir putaran ke-15, dan akan menghasilkan *collision* pada putaran ke-16.

8.3.4. Round Function Differences

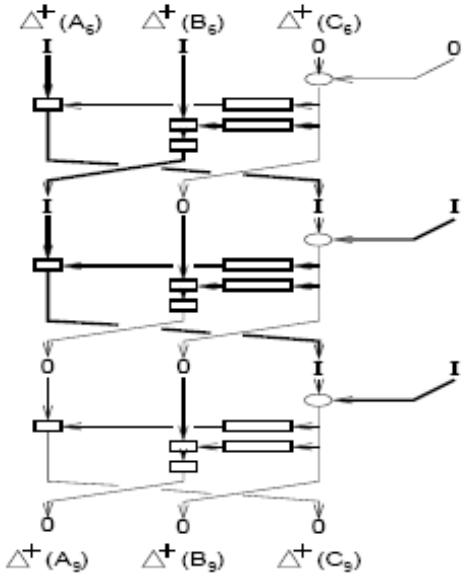

Fig. 2. Probability one characteristic from round 6-9.

Diberikan suatu karakteristik *key schedule* differential seperti di atas, kami dapat menspesifikasikan karakteristik differential untuk fungsi putaran dari akhir dari putaran ke-6 sampai akhir putaran ke-9, berjalan dari (I, I, 0) → (0, 0, 0) dengan membatalkan *differences* pada putaran ke-8 dan ke-9. Word pesan tambahan dari putaran 10-15 tidak akan mempunyai perbedaan, dan untuk seterusnya *collision* pada putaran ke-9 akan menjadi *collision* untuk 16 putaran dari Tiger. Gambar di atas menunjukkan karakteristik ini.

8.3.5. Modifikasi Pesan

Kesulitan utama dalam melakukan serangan adalah langkah dalam modifikasi pesan. Misalkan target *differences* kita pada akhir dari putaran ke-6 adalah

$$\Delta^{+}(A_6) = I,\ \Delta^{+}(B_6) = I, \Delta^{+}(C_6) = 0.$$

Secara independent, kita dapat mengetahui $\Delta^+(C_5)$ and $\Delta^+(C_4)$. Jika $\Delta^+(X_6) = 0$, kita membutuhkan $\Delta^+(C_5) = \Delta^+(B_6) = I$. Secara sama, kita akan mengetahui suatu hubungan $\Delta^+(C_4) = I + \Delta^+(\mathbf{odd}(B_6))$.

8.4. Local Message Modification dengan Meeting in the Middle

Misalkan kita mengetahui input (A_{i-1}, B_{i-1}, C_{i-1}) dan (A'_{i-1}, B'_{i-1}, C'_{i-1}), dan beberapa differences dalam word pesan X_{i-1} dan X_{i+1}. Kita ingin untuk memaksa beberapa difference penambahan $\delta^* = \Delta^+(C_{i+1}) = C_{i+1} - C'_{i+1}$. Seperti yang telah digambarkan pada gambar di bawah ini, *difference* $\Delta^+(C_{i+1})$ bergantung pada $\Delta^+(B_{i-1})$, keluaran dari *difference* penambahan dari fungsi ganjil pada putaran ke-i, dan keluaran dari *difference* penambahan dari fungsi genap pada putaran ke-i+1.

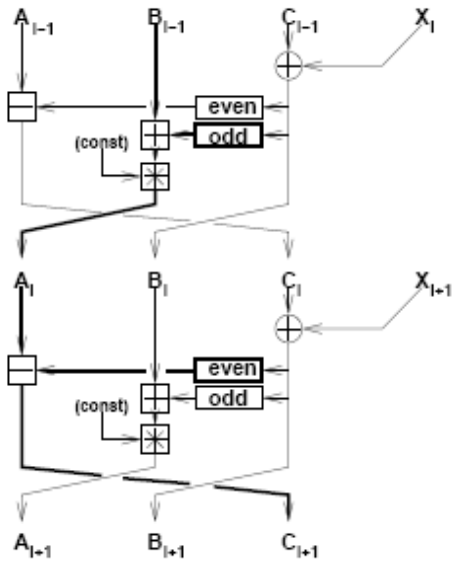

Fig. 3. The information flow from B_{i-1} to C_{i+1}.

8.4.1. Plain Message Modification

Untuk pertama, pertimbangkan terlebih dahulu fungsi genap, dimana setelah menghitung $B_{i+1} := C_i \oplus X_{i+1}$, dievaluasi sebagai

$$\mathrm{even}(B_{i+1}) := T_1(B_{i+1}[0]) \oplus T_2(B_{i+1}[2]) \oplus T_3(B_{i+1}[4]) \oplus T_4(B_{i+1}[6]).$$

Untuk setiap nonzero *difference* XOR antara word B_{i+1} dan B'_{i+1}, kami memperkirakan sekitar 2^{32} perbedaan *difference* pada keluaran hasil penambahan dalam bentuk even = even(B_{i+1}) – even(B'_{i+1}). Secara sama, jika kita berhadapan dengan fungsi ganjil

$$\mathbf{odd}(B_i) := T_1(B_i[7]) \oplus T_2(B_i[5]) \oplus T_3(B_i[3]) \oplus T_4(B_i[1]),$$

Kita mengharapkan dekat dengan 2^{32} keluaran penambahan *difference* yang berbeda dalam bentuk odd = odd(B_i) – odd(B'_i).

Seterusnya, jika *differences* dalam B_{i+1}[even] dan dalam B_i[odd] keduanya adalah nonzero, kita dapat menerapkan pendekatan *meet-in-the-middle* untuk memaksa

$$\Delta^+(B_{i-1}) + \delta_{\mathbf{odd}} - \delta_{\mathbf{even}} = \delta^*$$

- Simpan 2^{32} kandidat untuk $\delta_{\mathbf{odd}}$ dalam tabel.
- Untuk semua 2^{32} kandidat untuk $\delta_{\mathbf{even}}$, tes jika $\delta_{\mathbf{odd}}$ ada dengan

$$\delta_{\mathrm{even}} = (\Delta^+(B_{i-1}) + \delta_{\mathrm{odd}}) * (\mathrm{const}) - \delta^*,$$

or rather

$$\delta_{\mathrm{odd}} = (\delta_{\mathrm{even}} + \delta^*)/(\mathrm{const}) - \Delta^+(B_{i-1}) \qquad (1)$$

(note that since (const) is odd, division by (const) mod 2^{64} is well-defined).

Teknik ini mengambil beberapa 2^{32} evaluasi untuk setiap fungsi genap dan ganjil, dimana ekivalen dengan sekitar 2^{28} evaluasi dari fungsi kompresi – dan pastinya beberapa 2^{23} unit dari storage space. Kami menghitung untuk setiap nilai $\Delta^+(B_{i-1})$ and δ^* yang diberikan, pendekatan *meet-in-the-middle* sukses dengan nilai probabilitas mendekati ½. Dalam skenario serangan, kita akan mengulang pendekatan dengan $\Delta^+(B_{i-1})$ yang lain atau *difference* target yang lain δ^*, jika diperlukan. Misalkan X_i[even] telah diperbaiki dan MITM telah mengirim $\delta_{\mathbf{even}}$ and $\delta_{\mathbf{odd}}$ yang memenuhi pernyataan pertama di atas; $\delta_{\mathbf{even}}$ mendefinisikan B_{i+1} [even], dan $\delta_{\mathbf{odd}}$ mendefinisikan B_i [odd]. Akhirnya, kita akan mungkin untuk menghitung 64 bit pesan lokal:

$$X_i[\mathbf{odd}] := C_{i-1}[\mathbf{odd}] \oplus B_i[\mathbf{odd}] \quad \text{and}$$
$$X_{i+1}[\mathbf{even}] := C_i[\mathbf{even}] \oplus B_{i+1}[\mathbf{even}].$$

Perlu diingat jika C_i telah didefinisikan oleh X_i [even] yang telah diperbaiki. Dalam serangan ini, kita menggunakan dua variasi dalam menerapkan idenya.

8.4.2. Modifikasi Pesan untuk Mendapatkan XOR Difference

Dalam langkah ketiga dari serangan, kita membutuhkan *XOR difference* yang spesifik dalam C_3. Bagaimanapun juga, teknik *meet-in-the-middle* di atas hanya dapat bekerja untuk *difference* penambahan. Solusi kita terhadap masalah ini adalah dengan menjalankan komputasi secara brute force terhadap masalah ini: Untuk setiap *XOR difference* dari Hamming dengan bobot k, kita akan secara simpel menuju pencarian *meet-in-the-middle* untuk setiap *difference* penambahan dimana dapat dihasilkan oleh *XOR difference,* sampai kita kehabisan pilihan atau menemukan *difference* penambahan yang bernilai 8 dimana keduanya cocok dan menghasilkan *XOR difference* yang diharapkan ketika kita menghitung secara maju. *Difference* penambahan dimana dapat memacu ke k-bit *XOR difference* yang diberikan mempunyai peluang sekitar 2^{-k} untuk dilakukan. Jadi hal ini berarti kita membutuhkan usaha untuk mencoba sekitar 2^k *difference* penambahan yang konsisten dengan k-bit *XOR difference* sebelum kita sukses untuk menemukan pasangan *difference* penambahan sekitar setengah waktu yang diperlukan, kita akan membutuhkan secara total 2^{k+1} langkah MIM. Bagaimanapun juga, kita dapat mengoptimasi ini dengan cara yang sederhana, dengan hanya mengerjakan ulang satu sisi dari pencarian MIM untuk setiap target *difference* penambahan yang baru. Kerja yang diperlukan akan menjadi sejumlah 2^{28+k}.

8.4.3. Modifikasi Pesan dengan Constraints

Dua dari langkah pada MIM dalam serangan di atas harus hidup dengan *constraints* dalam melakukan seleksi terhadap bit pesan. *Constraints* berasal dari transisi antara *XOR difference* dalam C_3 dan *difference* penambahan dalam B_4. Jika *XOR difference* memiliki sejumlah k bit yang aktif , dan *difference* penambahan konsisten dengan hanya satu set dari nilai bit tersebut, k bit dari word pesan X_4 akan terkena constraints. Modifikasi pesan dengan kondisi ini akan terlihat relatif simpel: Dibandingkan dengan jika kita mencari lebih dari 2^{32} kemungkinan *difference* penambahan dari setiap sisi, kita mencari jumlah yang kecil, dengan bit yang terkonstraint dari pesan untuk disesuaikan dengan nilai yang dibutuhkan. Untuk kepentingan dari kesederhanaan, kita mengasumsikan bahwa k/2 bit dikenakan constraint pada bit yang genap, dan k/2 yang lainnya pada bit yang ganjil. Bagaimanapun juga, kemungkinan untuk sukses berkurang; dengan hanya 2^{28} pilihan dalam satu sisi, dan 2^{23} dari yang lainnya, kita mengharapkan sekitar 2^{-4} kemungkinan untuk cocok. Dan kemudian, kita diharapkan untuk mengulang pencarian MIM dengan 4 bit constraint sebanyak 16 kali.

8.5 Skenario Modifikasi Pesan

1. Kerjakan satu kali prekomputasi untuk menemukan *difference* penambahan L dengan bobot Hamming yang rendah berhubungan dengan XOR *difference* dimana kita dapat membatalkan dengan pilihan kita terhadap bit genap dari X_6. Hal ini memakan cost ekivalen dengan sebanyak 2^{27} fungsi hash Tiger-16, dan kita membutuhkan ini untuk menghasilkan suatu *difference* penambahan dimana konsisten dengan 8-bit *XOR-difference.* (9-bit *XOR difference* dimana satu dari bit aktif adalah bit *high order* yang memberikan hasil yang identik dalam sisa serangan yang terakhir).
2. Pilih X_0 dan X_1[even] untuk memastikan bahwa C_0 dan C_1 mempunyai *difference* yang berguna. Harus diperhatikan juga bahwa pada akhir setiap langkah, kita mengetahui $\Delta^{\oplus}(C_1)$ and $\Delta^{\oplus}(C_2)$. Kita menggunakan dua hal ini untuk langkah selanjutnya. Apa yang dikerjakan di tahap ini tidak memiliki arti yang besar.
3. Pilih X_1[odd] dan X_2[even] untuk memastikan bahwa C_2 mempunyai *difference* yang berguna. Harus diperhatikan bahwa pada akhir langkah ini, kita mengetahui nilai $\Delta^{\oplus}(C_2)$. Kita menggunakan *XOR difference* ini untuk langkah selanjutnya. Apa yang dikerjakan di sini juga tidak memiliki arti yang besar.
4. Lakukan langkah modifikasi pesan untuk mendapatkan *XOR difference* $\Delta^{\oplus}$dalam C_3 dimana hal ini konsisten dengan *difference* penambahan L. Hal ini telah dijelaskan di atas. Kerja yang dibutuhkan di sini adalah ekivalen dengan 2^{36} hash Tiger-16, dan kita mendefinisikan $X_2^{\mathbf{odd}}, X_3^{\mathbf{even}}$.
5. Pertemukan dengan constraint pada langkah tengah, pilih X3[odd],X4[even] untuk

mendapatkan $\Delta C_4 = I$. Kita mengharapkan untuk terdapat empat bit constraint, yang berarti bahwa kita diharapkan untuk mencoba hal ini sebanyak 16 kali sebelum menemukan kecocokan. Setiap usaha yang gagal mengharuskan kita mundur ke langkah 2. Lalu kita membutuhkan kompleksitas yang ekivalen dengan $2^4 2^{36} = 2^{40}$ Tiger-16 untuk menyelesaikan tahap ini dalam serangannya.

6. Pertemukan dengan constraint pada langkah pertengahan, pilih X4[odd],X5[even] untuk mendapatkan $\Delta C_5 = I$. Seperti sebelumnya, kita mengharapkan hal ini dikenakan constraint sebanyak 4 bit, lalu perlu untuk diulangi sebanyak 16 kali. Setiap kegagalan membutuhkan usaha kita untuk kembali ke langkah 2. Langkah ini akan selesai setelah melakukan kerja yang ekivalen dengan $2^4 2^{40} = 2^{44}$ hash dari Tiger-16.
7. Diberikan nilai dari C_5, kita menggunakan hasil dari langkah 1 untuk menemukan nilai byte genap dari X_6. Ini adalah pekerjaan yang tidak berguna dan tidak akan pernah gagal.

Hasil dari semua ini adalah *difference* penambahan dalam keluaran dari putaran terakhir I, 0, I. Dengan nilai kemungkinan 1, hal ini membatalkan karakteristik dari *key schedule,* menuju 16-round collision.

8.6. Kesimpulan dan Pertanyaan

Dalam tulisan ini, penulis menemukan *collision attack* terhadap 16 putaran dari Tiger dengan menggunakan teknik modifikasi pesan, membutuhkan kerja yang ekivalen dengan 2^{44} komputasi dari fungsi kompresi. Penulis juga telah lebih jauh melakukan eksplorasi teknik ini untuk melakukan serangan *near-collision* (dengan suatu input masukan tertentu) terhadap 20 putaran terakhir dari Tiger, juga sekitar 2^{48} komputasi dari fungsi kompresi.

8.6.1. Keamanan dari Tiger

Semua hasil yang kita dapatkan berbasis pada teknik modifikasi pesan, yang berarti bahwa kita memilih dua *difference* yaitu *XOR-difference* pada pesan dan juga nilai yang spesifik untuk sebagian besar atau semua bit pesan. Hal ini yang menghalangi serangan dalam berbagai jalan. Sebagai contoh, kita dapat melihat bahwa tidak ada jalan untuk mengadaptasi teknik kita yang sekarang untuk *collisions* terhadap aplikasi yang menggunakan 16 putaran Tiger dalam konstruksi HMAC – kelangkaan kita akan pengetahuan terhadap nilai berantai akan membuat pendekatan kita menjadi tidak mungkin. Serangan s*econd-preimage* dalam komputasi fungsi kompresi tunggal akan tampak sulit dilakukan dengan menggunakan teknik kita. Dua hal perbedaan baik *colliding message block* maupun nilai yang spesifik dari pesan dihalangi oleh serangan kita; nampaknya akan menjadi sangat sulit untuk bekerja mundur dengan menemukan *colliding message block* dari blok pesan spesifik dengan suatu keluaran hash. Gambaran serangan kedua ini mencoba untuk menemukan lebih dari delapan putaran, dan tampaknya memungkinkan untuk menemukan sampai dengan sebelas putaran mengunakan *local collision*, tetapi kami belum melakukan eksplorasi secara mendetail terhadap masalah ini. Kami lebih berkonsentrasi dengan kemungkinan untuk mengembangkan *collision attack* terhadap lebih banyak putaran. Oleh karena Tiger memiliki 24 putaran, maka dengan menyerang 16-20 putaran sudah cukup dirasakan mengancam. Perbaikan yang kecil dapat membuat teknik serangan kita berguna terhadap fungsi hash yang penuh. Kami pastinya belum mempercayai bahwa teknik serangan yang dijelaskan telah dieksploitasi secara penuh.

Kami menemukan bahwa *pseudo-and near collisions* dapat menjadi lebih terlihat lemah. Beberapa serangan terhadap cipher dari keluarga MD4 menggunakan *pseudo-and near-collisions* dalam skenario serangan dengan lebih dari satu blok pesan, untuk menemukan *collisions* yang dipakai untuk fungsi hash tersebut.

8.6.2. Tentang Tiger

Kami menggambarkan dua pelajaran yang cukup luas tentang analisisnya sejauh ini. Pertama, kami mempercayai bahwa Tiger memiliki putaran yang terlalu sedikit. Teknik modifikasi pesan memungkinkan kita untuk hampir secara penuh mengendalikan apa yang terjadi pada putaran ketiga dari fungsi hash, dan juga memperbolehkan kita untuk menempatkan *differences* dalam sisa putaran yang tersisa dengan hampir tanpa halangan. Kedua, penggunaan dari S-box yang besar dan campuran antara operasi XOR dan penambahan adalah strategi yang sangat bagus untuk membangun blok cipher, tapi hal ini beroperasi dengan sangat

berbeda di dalam fungsi hash. S-box yang besar diharapkan mempunyai suatu set yang besar tentang *differential* yang bagus, tetapi *differential* mana yang akan melewati putaran berikutnya tergantung pada nilai dari fungsi hash yang ada; penyerang menghadapi blok cipher dengan semacam S-box yang besar harus menebak fiffrerntial mana yang harus dicoba; penyerang menghadapi fungsi hash selalu memilih nilainya untuk membuat differential yang digunakan bekerja, atau paling tidak melihat ke dalam status dari fungsi hash untuk menemukan jalur differential yang harus dicoba.

8.6.3. Kegunaan dari Tools yang Digunakan dalam Serangan Keluarga MD4

Kita juga telah melihat beberapa overlap dalam tools yang digunakan untuk menyerang keluarga dari MD4, dan hasil kita dalam *reduced-round* Tiger. Secara lebih luas, kami menganalisis penambahan pesan untuk fungsi hash, dan membentuk karakteristik differential dimana, jika dimasukkan setelah putaran ketujuh, akan menuju ke suatu *collision* dalam fungsi hash secara penuh. Kita kemudian menggunakan modifikasi pesan untuk untuk memaksa hash untuk memproses sepasang pesan dengan *difference* yang berdasar keinginan kita menjadi karakteristik differential setelah putaran ketujuh. Hal ini mirip dengan teknik yang telah dijelaskan di atas, yaitu dengan menggunakan teknik modifikasi pesan tetapi dengan level yang lebih tinggi. Secara sama, variasi dari teknik neutral bit digunakan untuk membuat serangan *pseudo-near-collision* 20 putaran kita lebih efisien. Sementara detail dari tools serangan ini akan berbeda jika diterapkan terhadap Tiger, *high-level similarities* dalam pendekatannya menyarankan kita untuk belajar teknik serangan yang pada umumnya berguna terhadap fungsi hash dari hasil sementara dari fungsi hash yang ada pada keluarga MD4.

DAFTAR PUSTAKA

[1] Kelsey, John. (2006). *http://th.informatik.uni-mannheim.de/People/Lucks/papers/Tiger_FSE_v10.pdf*. Tanggal akses: 16 Desember 2006 pukul 14:00.

[2] Munir, Rinaldi. (2004). Bahan Kuliah IF5054 Kriptografi. Departemen Teknik Informatika, Institut Teknologi Bandung.

[3] Biham,Eli. (2004). *http://www.cs.technion.ac.il/~biham/Reports/Tiger/tiger/tiger.html*. Tanggal akses: 14 Desember 2006 pukul 13:00.