PENGAKSESAN DAUN SECARA RANDOM PADA HASH TREE

Eka Yusrianto Toisutta - NIM : 13504116
Program Studi Teknik Informatika, Institut Teknologi Bandung
Jalan Ganesha 10, Bandung
email: if14116@students.if.itb.ac.id

Abstract *– Makalah ini membahas mengenai modifikasi terhadap salah satu varian dari fungsi hash yaitu Hash Tree.Hash Tree merupakan salah satu jenis fungsi hash yang menerapkan struktur pohon dalam melakukan proses hash.*
Data yang akan di Hash akan dipecah-pecah ke dalam blok-blok data yang berukuran statis. Blok-blok data tersebut lalu dimasukkan ke dalam daun-daun dari Hash Tree secara berurutan untuk di-Hash hingga ke root-nya.
Hash Tree sebenarnya telah cukup baik dalam melakukan hash walaupun tanpa adanya penerapan fungsi random pada pengaksesan daun. Hanya saja diharapkan dengan adanya fungsi random tersebut maka hash Tree ini akan lebih baik lagi.

Kata Kunci*: Hash Function, Hash Tree ,Modifikasi, Eksperimen.*

1. PENDAHULUAN

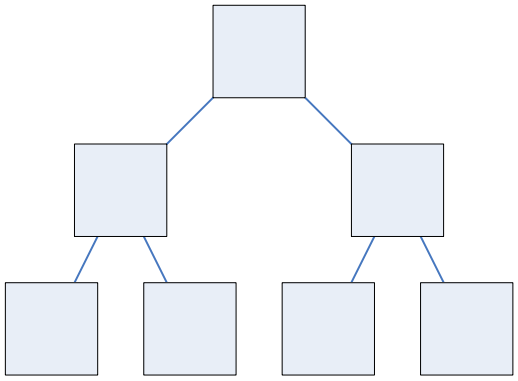
Perkembangan dunia digital saat ini membuat lalu lintas pengiriman data semakin ramai. Hampir setiap orang melakukan transaksi data setiap harinya. Hanya saja seiring bertambahnya jumlah data yang saling dipertukarkan, semakin banyak pula ancaman-ancaman terhadap transaksi data tersebut. Ancaman-ancaman tersebut antara lain *poisoning* dan *polluting*. *Poisoning* yaitu menyediakan data yang tidak sesuai dengan deskripsinya sedangkan *polluting* merupakan tindakan menyisipkan sesuatu yang jahat seperti virus ke dalam data yang dipertukarkan. Oleh karena itu dikembangkan fungsi *Hash* sebagai salah satu cara untuk membuktikan keotentikan suatu data yang dipertukarkan.

Hash Tree merupakan salah satu varian dari *Hash List* yang merupakan pengembangan dari fungsi *hash* yang telah ada sebelumnya. Sesuai namanya *Hash Tree* menggunakan struktur pohon dalam proses *hash* yang dilakukan.

Sepereti struktur pohon pada umumnya, *Hash Tree* juga memiliki akar (*root*), daun (*leaf*) , dan juga simpul internal (*internal node*). Untuk setiap daunnya akan diisi oleh potongan-potongan dari data yang akan diproses. Potongan-potangan data tersebut akan dikenakan proses *Hash* pada setiap daun dan simpul yang dilalui hingga mencapai pada simpul teratas atau pada akar dari pohon yang terbentuk. Hasil dari proses *hash* terakhir yang terjadi pada akar inilah yang akan dijadikan dicocokkan dengan nilai acuan yang telah diperoleh sebelumnya untuk mengecek kebenaran dari data yang dipertukarkan tersebut.

2. HASH TREE

Di dalam kriptografi, *Hash Tree* yang juga dikenal sebagai *Merkle Tree* merupakan sebuah struktur data yang mengandung sebuah struktur pohon yang berisikan rangkuman informasi mengenai data yang ukurannya lebih besar. *Hash Tree* merupakan merupakan ekstensi dari *Hash List* yang juga merupakan ekstensi dari fungsi *hash.*



Gambar 1: Contoh *hash tree*

2.1. Sejarah dan Kegunaan *Hash Tree*

Hash Tree ditemukan oleh Ralph Merkle pada tahun 1979. Pada saat itu tujuan utama dari *Hash Tree* ini adalah untuk membuat penanganan *Lamport one-time signatures* dalam jumlah banyak menjadi lebih efisien. Pada dasarnya sebuah *Lamport key* hanya dapat digunakan untuk menandatangani sebuah dokumen saja. Hanya saja jika digabungkan dengan *Hash Tree*, *Lamport key* tersebut dapat menangani banyak pesan dan pada akhirnya menjadi salah satu metoda penandatanganan digital yang cukup efisien.

Hash Tree dapat digunakan untuk memproteksi berbagai jenis data, baik yang disimpan, ditangani maupun disebarluaskan antar komputer. Hingga saat ini *Hash Tree* umumnya digunakan untuk memastikan blok-blok data yang diterima dari komputer lain melalui jaringan *peer-to-peer* tidak rusak maupun tidak berubah. Bahkan *Hash Tree* juga dapat berfingsi

untuk mengecek apakah pihak yang lain yang terlibat tidak berbohong mengenai data yang dikirimkannya.

2.2. Cara Kerja *Hash Tree*

Seperti namanya, *Hash Tree* adalah suatu pohon yang terbentuk dari satu atau lebih proses *hash* yang terjadi di dalam setiap simpul dan daun yang dimiliki oleh pohon tersebut. Pada setiap daun dari *Hash Tree* ini akan diisi oleh blok-blok data yang ukurannya telah ditentukan sebelumnya. Pada setiap daun lalu dilakukan *hash*ing terhadap blok-blok data yang terdapat pada daun tersebut. Hasil *hash* yang dihasilkan oleh setiap daun lalu diteruskan kepada *parent*-nya. Pada *parent* tersebut juga dilakukan *hash*ing terhadap hasil *hash* yang dihasilkan oleh *children*-nya. Hal ini lalu dilakukan secara rekursif hingga menyentuh simpul paling atas atau *root* untuk kemudian dilakukan *hash*ing terakhir.

Langkah-langkah yang dilakukan pada suatu proses *Hash Tree*, antara lain:

1. Menginisialisasi tiap-tiap daun dengan blok-blok data secara berurutan,
2. Lakukan *hash*ing terhadap nilai-nilai yang terdapat pada tiap-tiap simpul/daun,
3. Inisialisasi nilai parent dengan gabungan hasil *hash* yang diperoleh dari children yang dimilikinya,
4. Ulangi langkah 2 dan 3 hingga sudah tidak terdapat lagi parent yang dapat diakses dari simpul-simpul yang diproses (posisi simpul adalah akar atau *root*),
5. Hasil *hash* terakhir akan menjadi keluaran akhir dari *Hash Tree*.

Umumnya struktur pohon yang digunakan dalam *Hash Tree* berupa pohon biner (*binary tree*). Akan tetapi tidak berarti *hash tree* hanya menggunakan pohon biner saja. Jika dibutuhkan, *Hash Tree* dapat saja menggunakan struktur pohon lainnya seperti *n-ary tree*. Selain strukturnya, algoritma yang digunakan dalam proses *hash* yang dilakukan pada setiap simpulnya juga tidak melulu menggunakan satu jenis algoritma saja. Algoritma-algoritma *hash* seperti *SHA-1*, *Whirlpool*, atau *Tiger* dapat dipergunakan dalam *Hash Tree*. Bahkan jika tujuan digunakannya *Hash Tree* hanya untuk mencegah kerusakan yang tidak disengaja dalam transfer data, metode *checksum* yang tidak terlalu aman juga dapat diterapkan pada setiap simpul dan daunnya.

2.3. *Tiger Tree Hash*

Seperti yang dijelaskan sebelumnya bahwa algoritma yang diterapkan di dalam *Hash Tree* dapat berbeda-beda antara satu dengan lainnya. Hanya saja pada kenyataannya, terdapat suatu bentuk dari *Hash Tree* yang paling umum digunakan yaitu *Tiger Tree Hash*.

Dari segi struktur pohon yang digunakan, *Tiger Tree Hash* menggunakan struktur pohon biner (*binary tree*), dimana untuk setiap daunnya dimasukkan blok-blok data yang berukuran 1024 bytes. Untuk proses *hash*ingnya sendiri menggunakan algoritma *Tiger Hash* yang dinyatakan aman dalam konteks kriptografi

2.4. Keunggulan *Hash Tree*

Dengan struktur yang tidak jauh berbeda dengan *Hash* List, *Hash Tree* mempunyai keunggulan yang membuatnya lebih banyak digunakan untuk mendukung koneksi *peer-to-peer*. *Hash Tree* membuat pengecekan integritas terhadap suatu blok data dapat segera dilakukan ketika blok data tersebut telah diterima (diproses pada daun yang bersangkutan) walaupun belum seluruh bagian dari pohon tersebut diterima. Hal ini membuat proses recovery dari blok data yang rusak menjadi lebih mudah karena hanya blok data yang rusak saja yang akan di unduh ulang bukan keseluruhan data.

3. PENGAKSESAN DAUN SECARA RANDOM PADA HASH TREE

3.1. Overview

Pengaksesan daun secara berurutan (*sequential*) pada *Hash Tree* sebenarnya telah terbukti mampu menjamin hasil yang diperoleh. Hanya saja apabila digunakan sebagai sarana enkripsi, pengaksesan secara berurutan ini dapat memudahkan proses pemecahan hasil *hash*. Oleh karena itu pengaksesan daun secara acak (*random*) akan coba untuk diterapkan pada *Hash Tree*.

3.2 Lingkungan Pelaksanaan Eksperimen

Eksperimen penerapan pengaksesan daun secara random pada *Hash Tree* ini dilakukan pada lingkungan sebagai berikut :

- AMD® Athlon™ 64 3000+
- Memory DDR 1536 Mb
- Windows XP SP2 with Framework 2.0

Adapun perangkat lunak yang digunakan dalam eksperimen ini antara lain :

- Notepad++ dengan menggunakan bahasa pemrograman Java sebagai kakas pengembangan.
- Jacksum-1.7.0 sebagai kakas pengujian

3.3 Tahapan Pelaksanaan

3.3.1 Analisa Algoritma Dasar *Tiger Tree Hash*

Algoritma dasar yang digunakan merupakan algoritma *Tiger Tree Hash* yang diadaptasi untuk penggunaan pada perangkat lunak jacksum-1.7.0. Di mana terdapat kelas *TigerTree* yang mendefinisikan *Tiger Tree Hash*

di dalam perangkat lunak tersebut

```
Class TigerTree extends
MessageDigest{

/*variabel-variabel lokal*/

   public TigerTree(String name)
throws NoSuchAlgorithmException
   {
      //konstruktor
   }

/*fungsi-fungsi dan prosedur
kelas*/
   int engineGetDigestLength(){}
   void engineUpdate(byte in){}
   void engineUpdate(byte[] in,
int offset, int length){}
   byte[] engineDigest(){}
   int engineDigest(byte[] buf,
int offset, int len){}
   void engineReset(){}
   void blockUpdate(){}
}
```

Berikut adalah penjelasan singkat mengenai fungsi-fungsi dan prosesdur-prosedur yang terdapat pada kelas *TigerTree* :

- Fungsi engineGetDigestLength() merupakan fungsi yang mengembalikan jumlah byte dari hasil proses *hash* yang dilakukan.
- Prosedur engineUpdate(byte in) mengisi blok-blok pada daun apabila byte yang diolah datang satu per satu.
- Prosedur engineUpdate(byte[] in, int offset, int length) mengisi blok-blok pada daun apabila inputnya berupa array of byte.
- Fungsi engineDigest() mengembalikan array of byte yang merupakan hasil *hash* dari array of byte yang kosong.
- Fungsi engineDigest(byte[] buf, int offset, int len) mengembalikan integer yang merupakan hasil dari operasi *Hash Tree*.
- Prosedur engineReset() mengembalikan nilai seluruh atribut ke kondisi awal (0 atau null).
- Prosedur block Update() melakukan update terhadap blok-blok data dalam hal byte-byte tambahan untuk membedakan antara simpul dan daun.

3.3.2. Pengubahan Algoritma Dasar

Dari algoritma dasar yang telah ditampilkan sebelumnya, diketahui bahwa prosedur engineUpdate merupakan prosedur yang berperan untuk mengisi daun-daun pada *Hash Tree* untuk selanjutnya diproses lebih lanjut pada fungsi engineDigest. Oleh karena itu proses pengubahan akan dilakukan pada prosedur engineUpdate tersebut.

Perubahan pada prosedur-prosedur awal kelas antara lain :

- Penambahan fungsi random pada engineUpdate untuk menentukan blok mana yang harus diisi setelah suatu blok telah terisi.
- Pengubahan pada blok update dengan menerima input berupa integer untuk mengupdate daun dengan nomer urut sesuai inputnya

Selain perubahan pada prosedur engineUpdate perlu juga ditambahkan prosedur/fungsi tambahan, antara lain :

- Fungsi notVisitedLeaves(int i) mengembalikan boolean untuk menentukan apakah daun dengan nomer urut i sudah terakses atau belum. Ini digunakan untuk memastikan setiap daun hanya diisi satu kali dalam proses *Hash Tree*.
- Fungsi availableLeaves() berfungsi sebagai penentu jumlah daun yang harus disediakan untuk menghandle suatu data dengan ukuran tertentu
- Fungsi rearrangeBlocks() berfungsi untuk mengatur ulang blok-blok data yang telah urutannya telah dikacaukan oleh fungsi random. Hanya digunakan jika hasil *hash* harus dapat dikembalikan ke bentuk aslinya.

Adapun beberapa prekondisi yang harus sudah dipenuhi antar lain :

- Ukuran data yang akan dipertukarkan harus telah diketahui dengan pasti untuk menentukan jumlah daun.
- Pemrosesan nilai pada suatu simpul dilakukan ketika simpul-simpul yang berada di bawahnya telah selesai diproses.
- Apabila hasil *hash* harus dapat dikembalikan ke dalam bentuk awalanya, maka masukan pada setiap daun perlu ditambahkan byte yang menunjukkan urutan kedatangan dari blok data tersebut. Hal ini ditujukan agar dapat dilakukan penyusunan ulang terhadap blok-blok data tersebut.

Berikut adalah culikan dari prosedur engine Update setelah dilakukan perubahan sesuai dengan poin-poin di atas. Selain itu juga ditampilkan overview dari kelas *TigerTree* setelah dilakukan perubahan-perubahan yang disebutkan di atas.

```
void engineUpdate (byte in){
   int I = new Random();
   int j = I % availableLeaves();
   byteCount += 1;
   buffer[bufferOffset++] = in;
   if (bufferOffset==BLOCKSIZE){
      while(!notVisitedLeaf(j)){
         j = I.next();
      }
      blockUpdate(j);
      bufferOffset = 0;
   }
}
```

```
Class TigerTree extends
MessageDigest{

/*variabel-variabel lokal*/

   public TigerTree(String name)
throws NoSuchAlgorithmException
   {
      //konstruktor
   }

/*fungsi-fungsi dan prosedur
kelas*/
   int engineGetDigestLength(){}
   void engineUpdate(byte in){}
   void engineUpdate(byte[] in,
int offset, int length){}
   byte[] engineDigest(){}
   int engineDigest(byte[] buf,
int offset, int len){}
   void engineReset(){}
   void blockUpdate(int i){}
   bool notVisitedLeaf(int i){}
   int availableLeaves(){}
   byte[] rearrengeBlocks(){}
}
```

3.3.3. Eksperimen

Pada eksperimen terhadap hasil pengubahan di atas digunakan tiga jenis data yaitu :

1. File text kosong
2. File text dengan ukuran lebih besar dari 5 kB
3. Dokumen *Word*

Pada setiap file akan dilakukan satu kali pemrosesan dengan menggunakan *Tiger Tree Hash* standard dan dua kali dengan menggunakan *Tiger Tree Hash* dengan fungsi random. Hal ini ditujukan sebagai bahan komperasi antar ketiga tahapan tersebut.

3.3.3.1. File Teks Kosong

Pada pemrosesan file teks kosong atau berukuran 0 kB diperoleh hasil sebagai berikut :

***Tiger Tree Hash* Standar**
LWPNACQDBZRYXW3VHJVCJ64QBZNGHOH
HHZWCLNQ
***Tiger Tree Hash* dengan fungsi Random**
Percobaan 1:
LWPNACQDBZRYXW3VHJVCJ64QBZNGHOH
HHZWCLNQ
Percobaan 2:
LWPNACQDBZRYXW3VHJVCJ64QBZNGHOH
HHZWCLNQ

3.3.3.2. File Teks dengan Ukuran > 5kB

***Tiger Tree Hash* Standar**
27ZHRAHGAHN3BTZ37KA3XLADWK4RES5
2QHR3KR4A
***Tiger Tree Hash* dengan fungsi Random**
Percobaan 1:
VLM2XXZEUX5LVAG6UWJ4LWILQG35VOT6
JSK20SA
Percobaan 2:
D642QTVJ67ZCXAJCOBQRWCBBVQISFQJSI
UPGYKI

3.3.3.3. Dokumen Word

***Tiger Tree Hash* Standar**
ZVB5WHSPZAHSSNDUWWPXSCE6BM6PYR
KJGBW2IFI
***Tiger Tree Hash* dengan fungsi Random**
Percobaan 1:
NSLD522PACCMN7MSJ6L3RW5NYZRLSO45
VEWW4BY
Percobaan 2:
JAQGEMNPI2EU3UEJMO7ZN2365L3LGJPWC
JOURZI

3.3.4. Analisa

Dari keseluruhan rangkaian pengujian di atas, dapat disimpulkan bahwa :

1. Hasil *Hash* terhadap file teks kosong yang diperoleh dengan menggunakan *Tiger Tree Hash* Standar sama dengan *Tiger Tree Hash* dengan fungsi Random. Dikarenakan terdapat penanganan khusus terhadap file-file berukuran < dari batas ukuran blok yaitu 1024 bytes.
2. Terhadap file-file yang tidak kosong hasil yang diperoleh dari *Tiger Tree Hash* dengan fungsi Random selalu tidak sama. Sehingga terdapat inkonsistensi hasil.
3. Karena hasil keluaran dibatasi, maka hasil yang dicapai oleh kedua metode tersebut cenderung sama dalam hal jumlah keluaran dan karakter yang mengisinya.

Secara keseluruhan penerapan fungsi random pada

pengaksesan daun dalam *Hash Tree* cenderung menigkatkan keamanan dari segi pengembalian bentuk dari *message digest* ke dalam bentuk asalnya. Dengan adanya pengaksesan random, pihak ketiga menjadi tidak dapat menebak bentuk asal dari hasil *hash* tersebut.

Hanya saja jika digunakan untuk *checksum*, penggunaan fungsi random sangat tidak dianjurkan. Hal ini disebabkan oleh inkonsistensi hasil *hash* membuat tidak memungkinkan untuk membandingkan hasil *hash* yang diperoleh dari transfer data dengan hasil *hash* yang disediakan oleh pihak yang terpercaya sebagai acuan untuk memastikan keaslian file tersebut.

4. KESIMPULAN DAN SARAN

Dari pembahasan diatas maka kesimpulan yang dapat diambil, antara lain:

1. Penggunaan fungsi random pada *hash tree* membuat *hash tree* menjadi lebih kuat untuk menghalau ancaman *man in the middle*.
2. Inkonsistensi hasil yang dihasilkan membuat penerapan *hash tree* yang menggunakan fungsi random tidak baik digunakan dalam *checksum* atau validasi file.

Adapun saran antara lain:

1. Agar dapat digunakan sebagai *checksum*, perlu dipastikan pengaksesan daun yang dilakukan dalam transfer data sama persis dengan pengaksesan daun yang dilakukan oleh source acuan.
2. Penggunaan *Hash Tree* dalam *checksum* sangat bergantung pada sumber *hash root* yang digunakan. Oleh karena itu perlu dipastikan bahwa sumber *hash root* tersebut merupakan sumber yang terpercaya.

DAFTAR REFERENSI

[1]Munir, Rinaldi, *Diktat Kuliah IF5054 Kriptografi*, Sekolah Teknik Elektro dan Informatika Institut Teknologi Bandung, 2006.

[2]http://en.wikipedia.org/*Hash_tree*

[3]http://www.jonelo.de/java/jacksum/

[4]http://en.wikipedia.org/*Tiger*(cryptography)