Studi dan Analisis Penerapan Fungsi *Hash* dan Algoritma RSA pada *Download Manager*

Muhammad Anwari Leksono / 13508037
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
if18037@students.if.itb.ac.id

***Abstract*— Pada masa sekarang ini banyak aplikasi *download manager* yang beredar di internet. Aplikasi *download manager* berguna untuk mencari semua sumber file yang akan diunduh dengan tujuan meningkatkan kecepatan proses pengunduhan suatu *file* karena semakin banyak sumber *file* maka kecepatan akan semakin tinggi. Aplikasi *download manager* mengunduh *file* dengan cara mengunduh semua potongan *file* dan setelah semua potongan terkumpul, kumpulan potongan *file* tersebut digabung menjadi satu. Sumber yang berjumlah banyak dan jaringan komunikasi yang sibuk membuat potongan *file* yang diunduh dapat rusak atau disadap oleh pihak yang tidak diinginkan. Salah satu cara untuk memeriksa keaslian suatu *file* adalah dengan menggunakan fungsi *hash* dan salah satu cara untuk menjaga kerahasiaan suatu *file* adalah dengan cara mengenkripsi *file* tersebut. Penerapan beberapa alogitma kriptografi tentu akan memberikan dampak pada performa aplikasi tersebut dan pada *file* yang bersangkutan.**

***Index Terms*—download manager, hash, RSA, performa aplikasi.**

I. PENDAHULUAN

. Aplikasi download manager secara luas digunakan untuk memudahkan penggunanya mengunduh banyak file dari internet. Kemampuan untuk mengunduh biasanya secara *default* telah tesedia pada aplikasi *web browser* namun kelemahan fitur *download* pada web browser adalah proses pengunduhan yang biasanya tidak bisa dilanjutkan setelah dihentikan sebelumya atau kecepatan download yang sangat tidak stabil. Kekurangan yang lain adalah ketika file yang diunduh lebih dari satu maka aka nada satu proses download yang mati atau dengan kata lain kecepatannya mencapai angka nol.

Aplikasi download menawarkan kemampuan untuk mengatur proses download untuk banyak file dengan kecepatan yang relative lebih stabil dan dapat diatur. Selain itu aplikasi ini juga memudahkan pengguna untuk dapat melanjutkan proses download yang telah dihentikan sebelumnya.

Aplikasi download manager sangat berguna untuk file berukuran besar karena dengan kecepatan jaringan internet di Indonesia, aktivitas *pause-and-resume* proses download menjadi hal yang biasa dilakukan oleh pengguna download manager.

Keamanan jaringan internet merupakan hal yang penting untuk dipertimbangkan karena kemanan jaringan internet berpengaruh pada keamanan pengiriman pesan. Tidak semua pesan merupakan pesan terbuka untuk umum. Banyak pesan yang sebenarnya merupakan pesan rahasia/informasi rahasia sehingga pihak yang boleh mengetahui hanya pihak pengirim dan penerima saja.

Kecepatan relative stabil yang dimiliki aplikasi download manager membuat informasi yang sedang diambil dari internet menjadi lebih mudah untuk disadap atau dicuri. Hal ini disebabkan karena pola pengiriman pesan menjadi lebih jelas dan teratur sehingga kemudahan pencurian semakin meningkat.

Kriptografi menawarkan berbagai macam cara untuk menjaga kerahasiaan pesan sehingga meskipun pesan dicuri atau disadap, pihak pencuri tidak akan mendapatkan informasi apapun dari pesan yang ia curi. Selain itu kriptografi juga memberikan tawaran solusi untuk menjamin keaslian pesan. Pada proses download yang kita lakukan, file yang kita download belum tentu merupakan file yang kita inginkan. Tidak menutup kemungkinan bahwa selama perjalannya, file tidak rusak atau sengaja dirusak untuk suatu hal yang tidak baik. Kriptografi menyediakan cara untuk menyelesaikan masalah ini, salah satunya dengan menggunakan fungsi *hash.*

II. DASAR TEORI

Download Manager

Aplikasi *download manager* adalah sebuah program komputer yang digunakan untuk *download* (atau *upload*) file yang berdiri sendiri dan tidak terkait Internet untuk penyimpanan. Aplikasi ini dapat diintegrasi dengan *web browser* tertentu. Pada makalah ini contoh aplikasi *download manager* yang digunakan adalah Internet Download Manager atau sering disebut IDM dan BitTorrent. Aplikasi ini dapat diintegrasi dengan Mozilla Firefox, Opera, Google Chrome, Internet Explorer, Safari, Netscape, Flock, Orca, dan Mozilla Firebird.

Ketika *download manager* diintegrasi dengan *web browser* makan secara otomatis ketika pengguna akan mengunduh suatu file dari internet, URL dari file tersebut akan dilempar ke *download manager* dan kemudian

download manager akan mengunduh file yang berlokasi di URL yang telah ditentukan oleh pengguna.

Meskipun *download manager* memiliki fungsi yang sama, cara kerja tiap aplikasi *download manager* tidak selalu sama. Pada makalah ini ada dua contoh aplikasi *download manager*, Internet Download Manager (IDM) dan BitTorrent. Masing-masing memiliki kelebihan dan kekurangannya masing-masing.

Internet Download Manager

Aplikasi ini merupakan salah satu *download manager* yang sering digunakan oleh para peselancar dunia maya. Keunggulan aplikasi ini adalah dapat membagi satu file yang di-*download* menjadi beberapa bagian. Pada kondisi *default* aplikasi ini membagi sumber file menjadi 8 koneksi sehingga ketika dsatu file di-*download* dari internet *Internet Download Manager* atau IDM mengambil file tersebut dari 8 sumber.

Aplikasi IDM dapat menghentikan proses *download* sementara (*pause*) dan melanjutkan proses tersebut (*resume*). Fitur ini tidak selalu berhasil terutama jika URL file merupakan URL yang selalu berubah atau URL yang di-*generate* tiap kali ada permintaan untuk *download*. URL yang digunakan untuk mengambil file dapat berasal dari *Web Browser* atau input manual dari pengguna.

Kelebihan IDM adalah antara lain kemampuannya untuk membagi satu file menjadi beberapa segmen file dan masing-masing segmen akan ditangani oleh satu koneksi sehingga kecepatan *download* akan meningkat. Selain itu seperti yang telah dijelaskan di atas, IDM dapat menghentikan proses *download* sementara untuk dilanjutkan kembali.

BitTorrent

BitTorrent sendiri merupakan protocol sekaligus aplikasi yang memungkinkan satu file dibagi ke banyak computer dengan menggunakan jaringan internet. Protocol ini tergolong baru karena baru diperkenalkan pada tahun 2001 dan diimplementasi pada tahun 2002 oleh BitTorrent.Inc.

BitTorrent memiliki cara yang sedikit berbeda dengan IDM. Untuk me-*download* file menggunakan BitTorrent pengguna harus memiliki file berekstensi `.torrent` atau (dot)torrent.

File (dot)torrent berisi informasi mengenai potongan-potongan file yang akan diambil dari internet. Informasi mengenai potongan file tersebut minimal terdiri dari lokasi tiap masing-masing potongan filed dan informasi lain yang dibutuhkan untuk berbagi file.

Aplikasi BitTorrent akan mengambil file dari internet berdasarkan informasi yang tersimpan di file (dot)torrent tersebut. Aplikasi akan melacak lokasi masing-masing potongan file dan lokasi yang pertama kali ditemukan akan langsung digunakan dan diambil potongan file yang ada di sana. Lokasi tempat file disebut dengan *seed*. Karena urutan *download* tidak teratur maka potongan file yang didapat tidak selalu terurut sehingga pada akhir proses *download* aplikasi akan menyatukan potonga-potongan file tersebut.

Algoritma RSA

Algoritma ini pertama kali dipublikasikan oleh 3 orang : Ron Rivest, Adi Shamir, dan Len Adleman dari MIT. Algoritma RSA merupakan algortima kunci-publik. Kekuatan algoritma ini terdapat pada kesulitan memfaktorkan satu bilangan menjadi 2 faktor prima. Secara naïf pemecahan masalah pemfaktor ini bisa saja dilakukan namun cara yang efisien belum ditemukan.

Secara sederhana langkah-langkah dalam pembangkitan kunci algoritma RSA adalah sebagai berikut :

1. Pilih dua bilangan prima acak berbeda (p dan q).
2. Hitung nilai $N = p \times q$.
3. Hitung $\varphi = (p - 1) \times (q - 1)$.
4. Pilih bilangan bulat e dengan syarat $1 < e < \varphi$ dan e relative prima dengan φ.
5. Pilih satu bilangan d sehingga $d\,e = 1 \pmod{\varphi}$.

Kunci public yang dihasilkan adalah (e , N) dengan N merupakan nilai modulus dan kunci privat adalah (d, N). cara enkripsi dengan menggunakan kunci public adalah :

$ciphertext = (plaintext)^e \bmod N$,

dan untuk dekripsi digunakan cara sebagai berikut :

$plaintext = (ciphertext)^d \bmod N$.

Pesan yang akan dikiri harus dihitung terlebih dahulu nilainya sebelum dapat dihitung.

Fungsi *Hash*

Fungsi *hash* adalah fungsi satu arah yang didefinisikan secara khusus sehingga dapat mengonversi data berukuran besar menjadi satu nilai dengan panjang yang tetap. Nilai hasil perhitungan fungsi *hash* biasanya disebut nilai *hash*.

Perhitungan fungsi *hash* merupakan perhitungan yang peka akan perubahan sehingga perubahan sekecil apapun dalam data yang menjadi input akan berakibat perbedaan nilai hasil yang sangat besar.

Nilai *hash* biasanya digunakan sebagai tanda tangan digital, nilai dari suatu *password* yang tersimpan pada *database*, atau sebagai index dari suatu *record* pada *database*. Fungsi *hash* ada bermacam-macam. Contoh yang umum adalah antara lain : Md5, SHA1, dan SHA2. Langkah-langkah perhitungan fungsi *hash* berbeda untuk tiap fungsi *hash*.

III. PENERAPAN DAN ANALISA ALGORITMA RSA DAN FUNGSI *HASH*

Pada prinsipnya enkripsi dilakukan pada data yang akan dikirim dan dekripsi dilakukan pada data yang diterima. Pada aplikasi *torrent* fungsi enkripsi dipadang pada data yang siap dikirim. Fungsi enkripsi RSA membaca input berupa byte dari data yang akan dikirim. Kemudian hasil dari enkripsi akan dikirim ke klien. Fungsi *hash* digunakan untuk memeriksa keaslian suatu data.

Internet Download Manager

Bentuk aplikasi fungsi enkripsi dan dekripsi file pada IDM dapat dilakukan pada saat pengiriman file dan penerimaan. Jika IDM menggunakan RSA maka IDM harus dapat membangkitkan bilangan prima yang akan dipakai untuk membangkitkan kunci dengan kata lain IDM harus memiliki fungsi tambahan. Misalkan fungsi ini dinamakan `prime_generator()`. Fungsi ini akan mengembalikan deret bilangan prima besar terurut dari bilangan paling kecil ke bilangan yang paling besar. Algoritma pembangkit bilangan prima dapat menggunakan algortima *Sieve of Eratosthenes*.

Algoritma ini membangkitkan sederet bilangan prima dengan nilai maksimum yang ditentukan dan efektif untuk bilangan prima di bawah satu juta.

Berikut ini adalah tahap-tahap perjalanan *Sieve of Eratosthenes :*

1. Buat daftar angka dari 2 s.d. *n* secara berurut dengan *n* adalah nilai maksimal prima yang ditentukan.
2. Pada awalnya, misalkan $p = 2$.
3. Dari *p* cari semua bilangan yang merupakan kelipatan dari *p* (2*p*, 3*p*, 4*p*, dst.).
4. Temukan angka pertama yang bukan hasil dari pencarian pada langkah 3.
5. Ulangi langkah 3 dan 4 sampai p^2 lebih besar dari *n*.

Penggunaan *Sieve of Eratosthenes* efektif jika bilangan yang dicari kurang dari satu juga namun penggunaan cara ini akan menggunakan cukup banyak memori. Tiap bahasa pemrograman menggunakan alokasi memori yang berbeda untuk bilangan bulat (integer). Misalnya pada bahasa C#.NET integer (signed integer) ukuran integer adalah 32 bit atau 4 Byte, alokasi yang sama dengan bahasa C. jika ingin mencari deretan bilangan prima yang kurang dari 1 juta maka pada tahap awal harus dibuat *array* dari *integer* berukuran 1 juta dikurangi satu (perhitungan mulai dari angka 2) sehingga ada bilangan bulat yang berada dalam deret tersebut. Contoh bentuk table bilangan bulat yang digunakan adalah sebagai berikut :

2	3	4	5	6
7	8	9	10	11
12	13	14	15	…
61	62	63	64	65

Table 3.1 Tabel Bilangan Bulat 2 s.d 65

Dengan algoritma *Sieve of Eratosthenes* deretan bilangan prima yang didapat adalah sebagai berikut :

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	51	53	57	59	61

Table 3.2 Tabel Bilangan Prima antara 2 s.d 65

Semakin tinggi bilangan prima yang dicari semakin besar memori yang dipakai untuk membuat deret angka. Setelah itu, ambil dua angka untuk membangkitkan kunci untuk algoritma RSA tersebut. Pada makalah ini penulis tidak menggunakan angka yang besar karena tingkat kesulitan perhitungan yang tinggi. Penulis menggunakan angka-angka kecil.

Misalkan dua angka prima yang diambil (*p* dan *q*) adalah 53 dan 61. Dari dua angka itu, hitung *N* yang merupakan hasil kali dari dua bilangan tersebut sehingga didapat $N = 3233$.

Setelah mendapatkan nilai *N*, hitung hasil kali dari masing-masing bilangan setelah masing-masing dikurangi dengan satu. Hasil kali ini disebut φ dan $\varphi = 3120$. Setelah itu pilih bilangan *e* sehingga $1 < e < \varphi$ dan bilangan *d* sehingga $d\,e = 1 \pmod N$. Pada contoh ambil $e = 2753$ dan $d = 17$ sehingga kunci public adalah (*e*, *N*) yaitu (2753, 3233)dan kunci privat adalah (*d*, *N*) yaitu (17, 3233). Dengan adanya kunci public dan privat maka proses enkripsi dan dekripsi dapat dilakukan.

Pihak yang membangkitkan kunci adalah pihak yang memiliki file dan tidak ingin file yang ia kirim ke pihak lain disadap dan disalahgunakan oleh pihak yang tidak ia inginkan. Dengan pembuatan kunci enkripsi ia dapat mengenkripsi pesan yang akan ia kirim. Pihak yang menerima pesan kemudian akan mendekripsi pesan dari pengirim dengan menggunakan kunci public pengirim, setelah penerima selesai mengunduh pesan tersebut dengan IDM. Hal tersebut sudah normal terjadi. Namun jika hal ini diimplementasi pada IDM maka hal lain yang akan terjadi.

Jika IDM memiliki fitur enkripsi dan dekripsi maka fitur yang akan terpakai adalah dekripsi saja karena IDM tidak digunakan untuk mengirim pesan tetapi hanya menerima pesan (*download* pesan), dengan kata lain generator kunci `prime_generator()` yang telah dijelaskan di atas tidak akan berguna. IDM yang dapat mendekripsi pesan harus mengetahui kunci untuk dekripsi tersebut dengan kata lain ketika IDM menerima pesan, IDM harus sudah memiliki kunci untuk dekripsi. Jika demikian maka ketika IDM meminta file dari internet, IDM harus terlebih dahulu meminta kunci public milik pemilik file. Setelah IDM mendapatkan kunci publiknya barulah IDM dapat mengunduh file tersebut. Secara umum kegiatan IDM dengan fitur dekripsi dapat tertera sebagai berikut :

1. Membuat koneksi dengan pemilik file
2. Meminta kunci public pemilik file
3. Mengunduh file setelah kunci public diterima

Dari sini ada beberapa kemungkinan yang dapat terjadi. Pertama, jika pemilik file tidak mengirimkan kunci public, apa yang akan dilakukan IDM? Jika IDM langsung melanjutkan aktivitasnya dengan mengunduh file tersebut tanpa kunci public, apakah file yang IDM *download* dapat dipergunakan dengan baik? IDM juga harus bisa membedakan file yang terenkripsi dengan yang tidak. Dengan kata lain IDM harus menyimpan semua informasi yang berkaitan sumber file yang pernah ia *download*. Dengan demikian IDM dapat mengetahui apakah ia harus meminta kunci public terlebih dahulu atau tidak. Masalah berikutnya adalah bagaimana jika sumber file tertentu yang pada awalnya tidak memiliki kunci public, tiba-tiba memiliki kunci public? Menghadapi kemungkinan ini, IDM *terpaksa* harus selalu meminta kunci public untuk semua file yang akan ia *download*. Hal seperti ini akan berpengaruh terhadap kecepatan *download* dari IDM itu sendiri.

Masalah mengenai permintaan kunci public dapat diselesaikan dengan menggunakan *timeout* yang berarti jika dalam batas waktu tertentu pemiliki file tidak mengirimkan kunci publiknya maka file yang dimiliki sang pemilik diasumsikan tidak terenkripsi dan IDM tidak perlu mendekripsi setelah proses *download* selesai. Namun jika dalam batas waktu tertentu pemilik file mengirimkan kunci publiknya maka IDM akan menyimpan kunci tersebut untuk mendekripsi file yang telah ia *download*. Masalah baru yang akan muncul adalah penentuan nilai *timeout* yang tepat sehingga kesalahan asumsi mengenai ada-tidaknya kunci public minimal.

Masalah kedua adalah mengenai performa dari IDM itu sendiri.pada perhitungan awal untuk membangkitkan kunci, nilai kunci public yang dihasilkan dan diberikan pada IDM oleh pemilik file adalah (2753, 3233), atau dengan kata lain IDM akan mendekripsi file yang telah ia *download* dengan kunci ini. File yang dienkripsi dengan RSA biasanya akan memiliki ukuran yang lebih besar dari ukuran aslinya karena pada proses enkripsi, nilai suatu pesan akan dipangkatkan dengan kunci privat yang digunakan pemilik pesan. Cara enkripsi dengan menggunakan RSA dengan kunci privat (17, 3233) seperti yang telah dijelaskan sebelumnya adalah

$$ciphertext = (plaintext)^{17} \bmod 3233.$$

Dari sini telah terlihat bahwa pesan akan dipangkatkan dengan angka dengan skala ribuan. Misalkan saja nilai dari *plaintext* tersebut adalah 123 maka nilai *ciphertext* yang dihasilkan adalah 855 dengan perhitungan berikut :

$$ciphertext = (123)^{17} \bmod 3233.$$

Ketika IDM selesai mengunduh file terenkripsi maka IDM akan mendekripsi dengan kunci public yang ia terima sebelumnya. Berikut ini perhitungan yang digunakan untuk dekripsi.

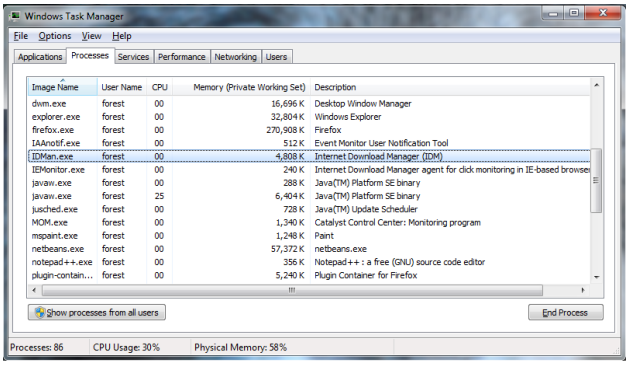
$$plaintext = (ciphertext)^{2753} \bmod 3233.$$

Nilai *ciphertext* yang selesai dinduh adalah 855 dan kunci public adalah (2753, 3233). Hasil dari dekripsi adalah hasil dari perhitungan

$$plaintext = (855)^{2753} \bmod 3233.$$

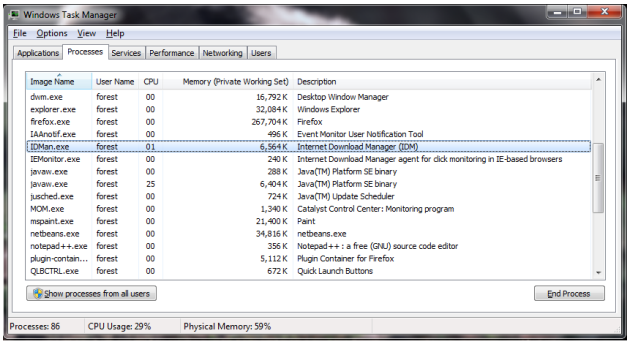
Nilai *plaintext* adalah 123, sama dengan nilai asli yang dienkripsi.

Pada proses enkripsi yang dilakukan pemilik file nilai perpangkatannya tidak terlalu besar, yaitu hanya 17. Namun pada proses dekripsi, IDM harus melakukan perpangkatan yang cukup besar, yaitu sebesar 2753. Pada kondisi normal, IDM yang berjalan pada system operasi Windows 7 menggunakan memori sebanyak sekitar 4808 K dan proses monitor IDM pada *web browser* berbasis Internet Explorer menggunakan memori sekitar 240 K. hal ini ditunjukkan dari gambar berikut.



Gambar 3.1 Tampilan Windows Task Manager untuk IDMan.exe dan IEMonitor.exe

Jika IDM sedang mengunduh satu file maka nilai memori yang digunakan akan seperti berikut :



Gambar 3.2 Tampilan Windows Task Manager untuk IDMan.exe dan IEMonitor.exe dalam Proses *Download*

Jumlah memori yang digunakan oleh proses IDMan.exe meningkat menjadi 6564 K. Kenaikan jumlah memori yang digunakan adalah sebesar 1756 K. Hal ini belum termasuk dengan proses dekripsi yang harus dilakukan jika IDM memiliki fitur dekripsi. Proses perpangkatan yang besar akan memakan memori lebih banyak lagi. Contoh yang diberikan penulis di makalah ini menggunakan bilangan dengan skala puluhan tapi pada prakteknya, bilangan prima pembangkit kunci berukuran besar dengan skala lebih dari ribuan.

Pada penerapan fungsi *hash* sebenarnya IDM juga tidak memiliki fitur ini. Ketika pengguna mengunduh file berukuran besar seperti file *disc image*, file dengan ekstensi .iso dsb., biasanya pada *website* tempat URL file tersebut tersedia, tertera pula nilai *hash* dari file tersebut. Sebagai contoh, pada halaman *web* yang menampilkan link untuk mengunduh Linux Mint 10 (*http://www.linuxmint.com/edition.php?id=67*) ada keterangan mengenai nilai *hash* menggunakan Md5, yaitu `08db04d607172d3a073543b447ebc919`.

Nilai *hash* diberikan sebagai referensi untuk pengunduh file *disc image* Linux Mint 10 tersebut. Pengunduh diminta untuk memeriksa nilai *hash* file yang ia *download* dengan cara menjalankan kembali fungsi Md5. Fungsi Md5 ini dapat dicari pengunduh dan didapatkan secara gratis.

Jika fungsi *hash* diintegrasi dengan *download manager*, dalam hal ini IDM, ada beberapa masalah yang akan timbul selain ada keuntungan yang akan didapat.

Masalah yang akan timbul pertama kali ada pemilihan fungsi *hash* yang sesuai. Pada contoh di atas fungsi *hash* yang digunakan adalah Md5. Tao Xie dan Dengguo Feng[10] mempublikasikan adanya *collision* pada Md5 dan hal ini tentu memberikan pertanyaan mengenai alasan penggunaan Md5 untuk memeriksa nilai *hash* dari *disc image* Linux Mint 10. Di lain pihak Fedora (*http://fedoraproject.org*) memberikan nilai *hash* untuk produknya dengan menggunakan fungsi SHA256. Fungsi Md5 memiliki *collision* dengan kompleksitas $2^{20.96}$ sedangkan SHA256 belum ditemukan.

Masalah yang lain adalah cara pemeriksaan yang akan dilakukan, yaitu apakah pemeriksaan nilai *hash* dilakukan tiap data yang berhasil diunduh atau pemeriksaan secara otomatis dilakukan setelah proses *download* selesai.

Jika pemeriksaan nilai *hash* dilakukan pada tiap data/potongan data yang berhasil diunduh maka data harus dikirim dalam bentuk paket data, dengan komponen tambahan yaitu nilai *hash* dari data yang dikirim. secara sederhana bagan paket data yang harus dikirim dan diterima oleh IDM adalah sebagai berikut

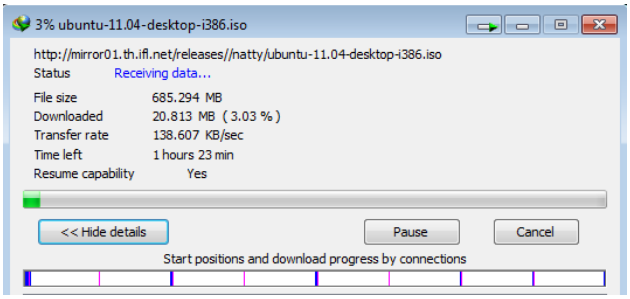
Data	Nilai *hash*

Gambar 3.3 Bagan Paket Data dengan Komponen Nilai *Hash*

Bagan 3.3 dibuat dengan asumsi bahwa saat ini potongan file yang diterima oleh IDM berbentuk *data*. Dengan demikian ada tindakan ekstra yang harus dilakukan oleh IDM untuk ekstraksi data dari paket data dan kemudian IDM menghitung nilai *hash* dari data hasil ekstraksi tersebut.

Pemeriksaan dengan cara ini dapat memakan waktu lebih lama karena hal ini berkaitan dengan kecepatan *download* IDM itu sendiri yang juga berkaitan dengan kecepatan jaringan atau ISP (Internet Service Provider) yang digunakan.

Misalkan kecepatan *download* adalah seperti yang ditunjukkan oleh gambar di bawah ini.



Gambar 3.4 Tampilan IDM untuk Transfer Rate

Kecepatan *download* pada gambar di atas adalah 138.607 KB/sec dengan kata lain IDM menerima data sebesar 138.607 KB per detik dan jika pemeriksaan dilakukan untuk tiap data yang datang maka aka nada pemeriksaan sebanyak ukuran file per *transfer-rate*, yaitu 685.294 MB / 138.607 KB yang sama dengan 5062.811 ≈ 5063 kali dengan asumsi kecepatan konstan. Jika kecepatan tidak konstan, jumlah pemeriksaan mungkin bertambah jika *transfer-rate* turun, atau mungkin berkurang jika *transfer-rate* naik.

Jika *transfer-rate* tinggi ada kemungkinan pemeriksaan akan terjadi sekali saja. Untuk mencapai hal ini *transfer-rate* harus lebih besar atau sama dengan ukuran file utuh ditambah dengan ukuran nilai *hash*.

Cara lain adalah, seperti yang telah disebutkan sebelumnya, dengan menghitung nilai *hash* setelah proses *download* selesai. Hal tambahan yang harus dilakukan jka cara ini diterapkan adalah IDM setelah berhasil membuat koneksi dengan sumber file, IDM harus meminta nilai *hash* dari file yang akan IDM *download*. Hal ini tidak selamanya berhasil karena tidak ada jaminan bahwa semua *server* dapat mengenali permintaan nilai *hash* yang diajukan IDM. Hal lain yang dapat terjadi adalah pertimbangan mengenai waktu yang digunakan IDM untuk menunggu jawaban dari *server* tentang ada-tidaknya nilai *hash* yang diminta. Solusi yang dapat ditawarkan untuk masalah ini sama dengan masalah IDM dengan permintaan kunci public, *timeout*. IDM menunggu dalam batas waktu tertentu dan jika dalam batas waktu tersebut *server* tidak mengirimkan nilai *hash* makan IDM tidak akan memeriksa nilai *hash* tapi jika IDM menerima nilai *hash* yang diminta, IDM akan memverifikasi hasil perkerjaannya. Hal seperti ini tentu akan memakan waktu lebih lama terutama jika ukuran file yang IDM *download* berukuran besar.

BitTorrent

Berbeda dengan IDM, BitTorrent mengandalkan banyak sumber untuk mengunduh satu file saja. Dengan kata lain potongan data yang diterima datang dari bebagai *server*, tidak dari satu *server* saja. Hal ini membuat fungsi *hash* menjadi penting untuk digunakan untuk memeriksa bahwa potongan data yang diterima adalah potongan data yang benar karena bukan tidak mungkin potongan dengan nomor urut tertentu di *server* tertentu adalah sama dengan potongan dengan nomor urut yang sama di *server* yang lain.

Penerapan fungsi *hash* pada BitTorrent ada pada saat paket diterima. BitTorrent memeriksa apakah nilai *hash* yang dihasilkan dengan yang tertera di paket data sama. Jika sama maka BitTorrent akan meminta potongan file selanjutnya tapi jika tidak sama maka BitTorrent akan meminta pengiriman ulang dari *server*.

Berbeda dengan IDM, tiap *client* BitTorrent menggunakan aplikasi yang sama untuk berbagi file karena file yang akan dibagi oleh BitTorrent hanya bisa terbagi kepada pengguna (*client*) BitTorrent yang lain sehingga jenis fungsi *hash* yang digunakan pasti sama.

Penggunaan fungsi *hash* secara awam memang akan menggunakan *resource* CPU dan waktu lebih tapi hal ini seimbang dengan kepastian potongan file yang didapat karena sumber potongan file berasal dari banyak *server*.

Untuk masalah penggunaan algoritma enkripsi/dekripsi, hal ini telah sempat dibahas di beberapa situs di internet. Pada awalnya protocol Torrent dimodifikasi dengan mengenkripsi *header* dari paket data yang dikirim tapi hal ini tidak mengamankan pengiriman sepenuhnya. Lalu lintas data masih dapat dideteksi.

Penggunaan RSA untuk enkripsi/dekripsi utuh memang menjanjikan keamanan tapi ukuran data yang dienkripsi akan menjadi sangat besar. Proses pembuatan kunci untuk RSA pada BitTorrent dapat menggunakan cara yang sama seperti yang telah dijelaskan pada bagian enkripsi/dekripsi untuk IDM. Namun demikian kunci public hasil pembuatan kunci harus dipublikasikan kepada *client* lain sehingga jumlah paket data yang akan tumpah ke jaringan akan bertambah.

Trik yang dapat diajukan di sini adalah dengan mengirim paket dalam ukuran yang kecil sehingga hasil enkripsinya tidak terlalu besar. Namun hal ini berakibat *traffic* jaringan akan menjadi sangat tinggi dan resiko jaringan *crash* akan semakin besar. Hal ini berujung pada penggunaan algoritma enkripsi/dekripsi simetris seperti AES. Namun cara seperti ini pun memiliki kelemahan lain, yaitu pemakaian CPU yang semakin tinggi, waktu yang semakin lama, dan perlu cara untuk bertukar kunci antara *client*, misalnya dengan menggunakan algoritma Diffie-Hellman.

Protocol Torrent menggunakan jaringan yang sangat luas karena menggunakan banyak *server* sebagai sumber data. Dengan sibuknya jaringan maka *latency* akan semakin tinggi dan *delay* akan meningkat sehingga *transfer-rate* akan mengecil. Lebih lanjut lagi, ketika ada satu *server* yang *crash* mendadak maka akan ada banyak paket data di jaringan yang tidak bertujuan dan hal itu hanya membuat lalu lintas jaringan semakin buruk.

Masalah lain adalah kompatibilitas *server* mengenai algoritma enkripsi/dekripsi yang digunakan. hal ini berujung pada banyaknya macam aplikasi berbasis protocol Torrent. BitTorrent menggunakan enkripsi yang berbeda dengan aplikasi sejenis lainnya. Hal menyebabkan file yang dibagi menggunakan BitTorrent tidak bisa digunakan oleh aplikasi sejenis lainnya.

Terlepas dari beberapa kelebihan dan kekurangan yang ditimbulkan karena implementasi algoritma enkripsi/dekripsi, aplikasi berbasis protocol Torrent seperti uTorrent memberikan pilihan penggunanya untuk menggunakan fitur enkripsi atau tidak.

IV. KESIMPULAN

Penggunaan fungsi enkripsi/dekripsi pada protocol BitTorrent dapat menjaga keamanan data yang dibagi pada umum tapi hal ini tidak efektif karena tujuan dari protocol BitTorrent adalah untuk memberikan kebebasan berbagi file. Selain itu BitTorrent memiliki *client* yang berbeda-beda sehingga kompatibilitas antar *client* akan menjadi masalah utama.

Implementasi fungsi enkripsi/dekripsi pada IDM juga tidak efektif karena IDM merupakan aplikasi yang digunakan oleh pihak umum, bukan pribadi. Selain itu kunci public yang digunakan IDM harus disebarluaskan dan hal itu berlaku juga untuk kunci privat sehingga enkripsi/dekripsi tidak berguna sama sekali.

Kedua aplikasi akan berjalan lebih lambat dibandingkan dengan tidak menggunakan fungsi enkripsi/dekripsi. Hal ini disebabkan karena RSA melibatkan bilangan prima yang sangat besar. Selain itu file yang diunduh juga jadi berukuran sangat besar.

Fungsi *hash* dapat digunakan untuk menjamin bahwa file/potongan file yang diunduh dari internet merupakan file/potongan file yang benar sehingga mengurangi resiko kesalahan file yang sudah selesai diunduh. Hal menjadi kebutuhan pada aplikasi berbasis protocol Torrent.

REFERENCES

[1] http://www.internetdownloadmanager.com/ (diakses pada tanggal 26 April 2011 pukul 9.20 PM).

[2] Preimage Attacks on 41-Step SHA-256 and 46-Step SHA-512, Yu Sasaki1*et al*..

[3] Cohen, Bram (2001-07-02). "BitTorrent — a new P2P app". Yahoo eGroup.

[4] A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, R.L. Rivest, A. Shamir, and L. Adleman.

[5] Randomized Hashing and Digital Signatures , Shai Halevi and Hugo Krawczyk.

[6] J. Wang, "Fundamentals of erbium-doped fiber amplifiers arrays (Periodical style—Submitted for publication)," *IEEE J. Quantum Electron.*, submitted for publication.

[7] C. J. Kaufman, Rocky Mountain Research Lab., Boulder, CO, private communication, May 1995.

[8] http://www.bittorrent.com (diakses pada tanggal 6 Mei 2011 pukul 9.40 PM)

[9] http://www.easycalculation.com/prime-number.php(diakses pada tanggal 6 Mei 2011 pukul 9.40 PM)

[10] Tao Xie, Dengguo Feng, "How To Find Weak Input Differences For MD5 Collision Attacks"

PERNYATAAN

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

Bandung, 9 Mei 2011
ttd

Muhammad Anwari L
13508037