STUDI MENGENAI HASHCASH DAN PENERAPANNYA UNTUK MENCEGAH *SPAMMING*

Weno Adji Syahdana – NIM : 13503084

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if13084@students.if.itb.ac.id

Abstrak

Fungsi *hash* adalah fungsi yang menerima masukan *string* dengan panjang sembarang dan mengonversinya menjadi *string* keluaran yang panjangnya tetap (umumnya berukuran jauh lebih kecil daripada ukuran *string* semula). *String* yang dihasilkan oleh fungsi *hash* ini disebut nilai *hash*. Fungsi *hash* dirancang sedemikian rupa agar menghasilkan nilai *hash* yang seacak mungkin, sehingga:

1. Dua *string* masukan yang hampir sama menghasilkan nilai *hash* yang sangat berbeda.
2. Untuk domain masukan tertentu, fungsi hanya menghasilkan sedikit benturan *hash* (*hash collision*).
3. Fungsi berlaku satu arah, yaitu jika diketahui nilai *hash* dari fungsi, maka secara komputasi tidak mungkin dapat ditemukan masukan yang bersesuaian.

Karakteristik fungsi *hash* ini, terutama poin ketiga, dimanfaatkan untuk membuat suatu teknik pencegahan *spamming* dan *Denial of Service* (DoS) yang dinamakan Hashcash. Ide hashcash adalah untuk memvalidasi seorang *client* yang melakukan permintaan suatu layanan ke *server* (misalkan mengirim *email*). Caranya dengan meminta *client* melakukan proses komputasi untuk menemukan suatu masukan yang menghasilkan nilai *hash* tertentu sebagai bukti pekerjaan (*proof of work*) dalam meminta layanan tersebut (dalam kasus pengiriman *email*, pekerjaan yang dimaksud adalah mengetik *email*). Selanjutnya, *server* akan mengecek apakah masukan tersebut benar menghasilkan nilai *hash* yang ditentukan. Jika ya, maka layanan akan diberikan.

Berdasarkan karakteristik fungsi *hash* poin ketiga, proses komputasi untuk menemukan masukan tersebut hanya mungkin dilakukan secara *brute force*, sehingga memerlukan *resource* pemrosesan yang cukup besar. Implikasinya, jika proses ini dilakukan secara massal, maka akan memerlukan *resource* yang sangat-sangat besar. Hal ini diharapkan dapat mencegah pengiriman *email* secara massal ke berbagai tujuan (*spamming*) dan mencegah serangan yang menyebabkan suatu server mengalami *Denial of Service* (DoS).

Makalah ini akan membahas seputar Hashcash, yaitu mengenai konsep dasar dan cara kerja Hashcash. Selain itu, makalah juga akan mengkaji penerapan Hashcash sebagai salah satu teknik untuk mencegah spamming, termasuk permasalahan yang ada dalam penerapan tersebut, dan peluang pengembangan Hashcash untuk tujuan lainnya. Pada akhir makalah akan dikemukakan salah satu contoh aplikasi implementasi Hashcash.

Kata Kunci *: Hash, Hashcash, spam, email.*

1. Pendahuluan

Seiring semakin aktif dan berkembangnya aktivitas ber-internet di berbagai belahan dunia, gangguan yang munculpun semakin bervariasi. Salah satu masalah klasik yang dihadapi para pengguna internet adalah *spamming*. Pelaku *spamming* mengirimkan *email* secara massal ke satu atau berbagai tujuan (alamat *email*) dengan maksud komersial (melakukan promosi), politis (menyebarkan propaganda), ataupun hanya sekadar mengganggu pemilik alamat *email*. *Spamming* ini menyebabkan kerugian baik dari sisi waktu si korban *spamming*, maupun kerugian pemakaian bandwidth internet untuk hal-hal yang tidak bermanfaat.

Pada kasus *spam*, *cost* komputasi yang dibebankan kepada pelaku seiring kenaikan jumlah *email* yang dikirim hampir setara nol.

Pelaku ini dapar mengambil keuntungan bahkan dengan tingkat kesuksesan 0,0001 % saja (hanya 1 dari 1 juta *email* yang dikirim). Kerugian yang ditimbulkan dari hal ini adalah, pengguna internet menjadi terganggu dengan iklan yang bukan menjadi *interest* mereka. Juga, dalam skala global terjadi pemborosan *bandwidth* dan sumber daya *CPU*. Penerima *spam* juga membuang waktunya dan mungkin membayar bandwidth pada Penyedia Jasa Internet (*Internet Service Provider* / ISP) untuk *spam* tersebut.

Dari pihak ISP sendiri juga mengalami kerugian karena menyebabkan pengaduan dari para pengguna jasa ISP tersebut. Pengaduan ini dapat menimbulkan ekstra waktu dan uang yang dikeluarkan ISP. ISP pun harus melakukan pemulihan terhadap *mail server* yang terkadang mengalami *crash* karena *spamming* yang intensif.

Berbagai teknik telah diterapkan dalam memerangi *spamming*. Masing-masing teknik mempunyai pendekatan yang berbeda-beda. Salah satu teknik yang diajukan memanfaatkan karakteristik fungsi *hash* untuk membuat suatu mekanisme validasi dengan meminta pengirim email melakukan proses komputasi tertentu sebagai bukti pekerjaan (*proof of work*) mengetik *email*. Teknik ini dinamakan Hashcash.

Secara teori, para pelaku *spam* (*spammer*), yang kesuskesannya tergantung pada kemampuan untuk mengirim sejumlah besar *email* dengan *cost* sangat kecil untuk setiap pesan yang dikirim, tidak akan mau melakukan proses komputasi sebagai bukti pekerjaan karena proses tersebut menyebabkan mereka mengeluarkan *cost* yang semakin besar untuk setiap tambahan *email* yang dikirimkan. Penerima *email* selanjutnya dapat memverifikasi apakah pengirim melakukan proses pembuktian pekerjaan dan hasilnya dapat digunakan untuk membantu menyaring *email* yang masuk.

Pada penerapannya, Hashcash bukan hanya dapat digunakan untuk mencegah spamming, namun juga untuk mencegah serangan yang mengakibatkan suatu server mengalami Denial of Service (DoS).

2. Konsep Dasar Hashcash

Hashcash bekerja dengan menantang *CPU* untuk melakukan suatu *cost-function*. *Cost-function* ini bertugas menghitung sebuah token yang dapat digunakan sebagai bukti pekerjaan (*proof-of-work*). Varian interaktif and non-interaktif dari *cost-functions* dapat dibangun untuk digunakan pada situasi di mana *server* dapat mengeluarkan sebuah tantangan/*challenge* (*connection oriented interactive protocol*), maupun saat *server* tidak dapat mengeluarkan sebuah tantangan (saat tipe komunikasi yang terjadi adalah *store–and–forward*, atau *packet oriented*).

Hashcash adalah biaya yang harus dikeluarkan CPU dengan mengalkulasi sejumlah n bit di mana terjadi benturan hash parsial (*partial hash collisions*) pada suatu teks yang dipilih.

Ide penggunaan hash parsial adalah karena dapat dibuat sangat mahal untuk dihitung (dengan memilih jumlah bit yang berbenturan), namun dapat diverifikasi dengan mudah dan cepat. Hal ini menjadi dasar dari sebuah sistem *ecash*. Sistem seperti ini dapat digunakan untuk mengendalikan penyalahgunaan sumber daya internet yang tidak terukur (*un-metered*). Salah satunya adalah untuk mencegah *spamming*.

Hashcash dapat diukur dari panjang bit yang mengalami benturan hash parsial. Kenaikan satu bit lebih panjang menyebabkan *cost* rata-rata yang dibutuhkan untuk melakukan komputasi menjadi dua kali lipat dari semula. Dengan memilih jumlah benturan hash yang sesuai, kita dapat membuat sistem pengukuran yang tidak menimbulkan masalah untuk pengirim *email* biasa, namun mengakibatkan *cost* yang mahal untuk pengiriman *email* sejumlah besar.

Sebagai contoh, sebuah *workstation* standard mampu untuk melakukan komputasi 180,000 *hash* per detik. Sebuah benturan hash 20 bit dapat dihasilkan dalam waktu rata-rata 6 detik untuk *rate* ini. Jika *workstation* tersebut dikenakan 20 bit hash untuk setiap *email* yang dikirimkan. Maka, selama sehari penuh *workstation* tersebut hanya dapat mengirim *email* sebanyak 3750 buah.

Jumlah tersebut seharusnya lebih dari cukup untuk pengguna (pengirim *email*) biasa. Tapi jumlah ini dapat membatasi *spammer* dalam melakukan aksinya apalagi dengan tingkat kesuksesan yang rendah. Hal ini menyebabkan *spammer* hanya mempunyai 2 pilihan: berhenti mengirim *spam*, atau mencari cara untuk menambah tingkat kesuksesan, dalam artian melakukan *spamming* dengan lebih efisien

(tepat-sasaran). Kedua hal ini berimbuh hasil baik bagi pengguna internet.

Inspirasi dari hashcash adalah ide bahwa beberapa fakta matematika sangat sulit untuk ditemukan, namun dapat dengan mudah diverifikasi. Contoh yang diketahui secara umum adalah melakukan pemfaktoran untuk bilangan besar (khususnya bilangan yang hanya mempunyai sedikit faktor). Mudah bagi sebuah *CPU* untuk melakukan proses perkalian beberapa bilangan secara bersamaan untuk menemukan hasil perkaliannya, tapi jauh lebih sulit untuk menemukan faktor dari hasil perkalian itu.

Kriptografi kunci-publik RSA dibangun atas dasar properti dari faktorisasi tersebut. Memberikan jawaban dari tantangan faktorisasi yang diberikan adalah suatu pembuktian bahwa responden sudah melakukan suatu ukuran tingkat pekerjaan tertentu.

Faktorisasi bekerja dengan baik untuk sebuah mekanisme tantangan yang interaktif. Misalkan saja ada sebuah sumber daya *online* yang diinginkan untuk dibayar secara simbolis oleh si penantang. Maka pihak yang memberi tantangan akan menantang si penantang untuk memfaktorkan suatu bilangan tertentu. Hanya peanantang yang dapat membuktikan bahwa mereka *interest* terhadap sumber daya tersebut dengan melakukan pemrosesan *CPU* untuk menjawab tantangan itu.

Namun bagaimanapun juga, beberapa sumber daya tidak dapat dinegosiasikan secara interaktif seperti kasus di atas. Satu sumber daya yang dinilai berharga adalah *inbox email* seseorang. Pesan yang tak diinginkan dapat mengambil ruang *disk* dan bandwidth kepunyaan orang tersebut. Dan yang lebih merugikan adalah pesan tersebut dapat menyita waktu. Tidaklah menjadi masalah ada seorang asing yang mengirim pesan ke *inbox* seseorang. Namun orang asing tersebut harus mempunyai suatu pembuktian bahwa pesan yang mereka kirim adalah serius dan memang penting.

Untuk mendapatkan suatu “pembayaran” yang non-interaktif, hashcash memberikan kesempatan bagi penerima pesan untuk mengeluarkan sebuah tantangan yang ditujukan pada orang yang ingin mengirim *email* kepada si penerima pesan tersebut. Pengirim *email* harus menyertakan sebuah cap (*stamp*) nilai hash yang valid pada *header* pesan yang dikirimkan; lebih rincinya, nilai hashcash tersebut diperoleh salah satunya dengan memasukkan alamat tujuan.

Cara hashcash melakukan tantangan adalah dengan meminta penantang (pengirim *email*) untuk menghasilkan *string* (cap/*stamp*) yang mana saat string tersebut di-*hash* dengan menggunakan Secure Hash Algorithm (SHA-1), akan mengeluarkan nilai hash yang berawalan sejumlah nilai nol (‘0’). Jumlah nilai nol tersebut mengindikasikan nilai bit dari cap hashcash. Karena atribut keseragaman dan kekuatan kriptografi dari SHA-1, satu-satunya cara untuk menemukan sebuah cap hashcash dengan nilai bit b yang diberikan adalah dengan menjalankan SHA-1 sebanyak rata-rata 2^b kali.

Namun, memverifikasi sebuah cap hanya membutuhkan satu kali komputasi SHA-1. Untuk penggunaan pada *email*, sebuah nilai bit sebesar 20-bit adalah nilai yang direkomendasikan. Pengirim membutuhkan waktu untuk melakukan sejuta kali percobaan untuk menemukan cap yang valid, yang mana pada kebanyakan *CPU* saat ini memakan waktu kurang dari satu detik pemrosesan.

Seperti telah dijelaskan, hashcash adalah metode untuk menambahkan cap berupa teks pada *header* sebuah *email* untuk membuktikan bahwa si pengirim telah menghabiskan sejumlah sumber daya *CPU* untuk mengalkulasi cap tersebut sebelum melakukan pengiriman *email*. Dengan kata lain, dengan menghabiskan sejumlah waktu tertentu untuk menghasilkan cap, maka si pengirim kemungkinan bukanlah *spammer*. Si penerima, dengan *cost* komputasi yang rendah, kemudaian dapat memverifikasi bahwa cap valid.

Contoh baris *header* dari hashcash adalah sebagai berikut:

X-Hashcash:

```
1:20:060408:adam@cypherspace.org::1Q
TjaYd7niiQA/sc:ePa
```

Secara teknis, sistem hashcash diterapkan melalui langkah berikut:

- Komputer penerima *email* akan mengalkulasi nilai hash 160 bit SHA-1 dari seluruh string `"1:20:060408:adam@cypherspace.org::1QTjaYd7niiQA/sc:ePa"`. Ini

memakan waktu sekitar 2 mikro detik pada mesin berkecepatan 1 GHz – jauh lebih singkat daripada waktu yang dibutuhkan untuk menerima sisa isi *email*. Jika 20 bit pertama nilai hash yang dihasilkan adalah '0', maka cap tersebut valid.

- Komputer penerima mengecek tanggal pada *header* "060408" (8 April 2006). Jika tanggal tersebut berkisar 2 hari dari hari penerimaan *email*, maka cap tersebut valid (untuk mengompensasi pemuluran waktu dan waktu *routing*).

- Komputer penerima mengecek apakah alamat *email* pada *header* tersebut adalah salah satu alamat *email* yang valid dari penerima (atau salah satu milis yang diikuti penerima).

- Jika semua pengecekan lainnya valid, maka komputer penerima menyimpan cap tersebut pada *database.* Jika cap yang sama tidak ada dalam *database*, maka cap tersebut valid.

Pengirim diharuskan untuk menghasilkan sebuah baris *header* yang berhasil melewati semua pengetesan yang disebutkan di atas. Komputer pengirim pertama-tama membentuk string hashcash inisial (tanggal, alamat *email*, dan sebuah bilangan acak pada akhir string). Lalu computer pengirim akan meng-inkremen bilangan random tersebut berulang kali dan menjalankan SHA-1 sampai menemukan nilai hash yang berawalan nol ('0') sejumlah yang ditentukan. Untuk memperoleh nilai nol pada 20 bit pertama nilai hash membutuhkan sekitar 2^20 iterasi atau sekitar 1 detik pada mesin berkecepatan 1 GHz. Pengirim biasa bahkan tidak akan menyadari bahwa komputernya bekerja satu detik untuk menghasilkan string hashcash. Sebaliknya, bagi para *spammers* satu detik tersebut lebih baik digunakan untuk mengirim ratusan *spam*, daripada mengalkulasi nilai hashcash untuk hanya satu *spam*.

Waktu yang dibutuhkan untuk melakukan komputasi agar menghasilkan benturan hash adalah eksponensial terhadap jumlah bit nol yang ingin dicari. Jadi, kita dapat menambah jumlah bit nol yang harus dicari sampai nilai komputasi akan terlalu mahal bagi para *spammer* untuk mencari *header* yang valid. Sementara untuk mengonfirmasi *header* yang valid selalu membutuhkan waktu yang sama berapapun bit nol yang ditambahkan.

3. Cara Kerja Hashcash

Kerja hashcash adalah berdasarkan prinsip *cost-function*. Berikut akan dijelaskan mengenai *cost-function* dan bagaimana penerapannya pada mekanisem hashcash.

3.1 *Cost Function*

Sebuah *cost-function* haruslah efisien untuk diverifikasi, namun, dengan parameter tertentu, mahal untuk dihitung. Kita akan menggunakan notasi berikut untuk mendefinisikan sebuah *cost-function*.

Dalam konteks *cost-functions*, digunakan istilah *client* untuk merujuk pada pengguna yang harus melakukan komputasi pada sebuah token (dilambangkan τ) menggunakan suatu *cost-function* MINT() yang mana akan digunakan untuk membuat token ikut berpartisipasi dalam sebuah protokol dengan suatu *server*. Kita menggunakan istilah *mint* untuk *cost-function*.

Server akan mengecek nilai dari token menggunakan fungsi evaluasi VALUE(), dan hanya memroses lebih lanjut dengan protokol jika token mempunyai nilai yang diperlukan.

Pada fungsi terdapat parameter jumlah pekerjaan rata-rata ω yang pengguna akan lakukan untuk menghasilkan sebuah token.

Dengan *interactive cost-functions*, *server* akan mengeluarkan tantangan/*challenge* C kepada *client* – *server* menggunakan fungsi CHAL() untuk menghitung tantangan tersebut. (Fungsi *challenge* juga membutuhkan parameter berupa faktor kerja).

$C \leftarrow \mathrm{CHAL}(s, \omega)$ server challenge function
$\tau \leftarrow \mathrm{MINT}(C)$ mint token based on challenge
$v \leftarrow \mathrm{VALUE}(\tau)$ token evaluation function

Dengan *non-interactive cost-functions client* memilih tantangan/*challenge*-nya sendiri atau nilai awal acak pada fungsi MINT(), dan tidak terdapat fungsi CHAL().

$\tau \leftarrow \mathrm{MINT}(s, \omega)$ mint token
$v \leftarrow \mathrm{VALUE}(\tau)$ token evaluation function

Pada *cost-function* ini ada beberapa karakteristik/atribut yang penting untuk digarisbawahi. Di antaranya:

- ***Publicly auditable.*** *Cost-function* dapat diverifikasi secara efisien oleh sembarang pihak ketiga tanpa akses ke *trapdoor* atau informasi rahasia apapun.

- ***Fixed cost.*** *Cost-function* memerlukan jumlah sumber daya yang tetap untuk melakukan proses komputasi. Algoritma tercepat untuk menghasilkan token yang berbiaya tetap adalah algoritma yang deterministik.

- ***Probabilistic cost.*** *Cost-function* bercirikan *cost* yang diperlukan *client* untuk menghasilkan sebuah token mempunyai atribut waktu yang dapat diperkirakan, namun waktu actualnya acak jika *client* dapat menghitung secara efisien *cost-function* tersebut dengan memulai dari nilai awal yang acak. Terkadang *client* mendapatkan keberuntungan dan mulai mendekati solusi.

- ***Trapdoor-free.*** *Cost-function* yang *trapdoor-free* jika dengan mengeluarkan token, *server* tidak mendapat keuntungan apa-apa.

  Contoh dari *trapdoor-free cost-function* adalah hashcash *cost-function*. Sementara *Juels and Brainard's client-puzzle cost-function* adalah contoh dari *cost-function* di mana *server* mendapatkan keuntungan dengan mengeluarkan token.

Sebuah kerugian dari solusi *cost-function* yang ditawarkan adalah bahwa penantang dapat membuat token secara murah dari nilai arbiter.

Hal ini tergambar pada layanan publik di mana *server* kemungkinan mempunyai konflik kepentingan (*conflict of interests*), misalkan pada *web hit metering.* Pada kasus ini *server* mempunyai suatu kepentingan untuk menaikkan nilai *hit* pada halamannya karena *server* tersebut dibayar per *hit* oleh pengiklan.

Hashcash adalah suatu *cost function* yang non-interaktif, *publicly auditable*, *trapdoor-free* dengan *unbounded probabilistic cost*.

4. Penerapan Hashcash pada *Email* untuk Mencegah *Spamming*

4.1 Format Hashcash (versi 1)

Pada penerapan hashcash, meminta suatu nilai hash khusus dari SHA-1 saja tidak cukup. Kita juga menginginkan cap yang dihasilkan agar spesifik terhadap sumber daya yang diinginkan -- misal, sebuah cap untuk alamat mertz@gnosis.cx seharusnya berbeda dengan cap untuk alamat someuser@yahoo.com. Jika tidak, maka *spammer* dapat menghasilkan hanya satu cap bernilai bit tinggi untuk digunakan di mana-mana.

Begitupun, sekali digunakan, cap ini tidak dapat dibagi-pakai (*shared*) di antara para *spammer* yang ingin mengirim *email* kepada penerima tertentu. Untuk itulah, hashcash juga memakai dua langkah tambahan (atau setidaknya merekomendasikan langkah ini sebagai bagian dari protokol):

- Pertama, cap juga mengandung informasi tanggal. Seorang pengguna mungkin saja ingin agar cap yang sudah melewati periode waktu tertentu dikategorikan sebagai cap yang tidak valid.

- Kedua, sebuah hashcash *client* (penerima) seharusnya mengimplementasi suatu *double spend database*.

Double spend database maksudnya adalah sistem di mana setiap cap hanya boleh digunakan satu kali saja; Jika cap tersebut diterima untuk kedua kalinya, maka akan dikategorikan tidak valid.

Untuk rincinya, sebuah cap hashcash (versi 1) akan terlihat seperti kode berikut:

```
1:bits:date:resource:ext:salt:suffix
```

Cap ini terdiri dari 7 bagian:

1. Nomor versi dari format cap hashcash (versi nol lebih sederhana, namun memiliki batasan-batasan tertentu).

2. Jumlah bit nol ('0') yang diklaim. jika cap tidak memenuhi ketentuan jumlah ini, maka cap tersebut tidak valid.

3. Tanggal dan waktu saat cap dihasilkan. Cap yang melebihi atau kurang dari batas waktu yang ditentukan akan menjadi tidak valid.

4. Sumber daya di mana cap tersebut dihasilkan. Dapat berupa alamat *email*, atau sebuah URI atau nama lainnya yang menandakan sumber daya.

5. Ekstensi yang dibutuhkan pada aplikasi tertentu. Data tambahan lainnya dapat ditempatkan di sini, namun pada penggunaannya kebanyakan bagian ini kosong.

6. Sebuah nilai acak (*random salt*) yang membedakan cap dengan cap lainnya yang dihasilkan pada sumber daya dan waktu yang sama. Misalnya, dua orang berbeda mungkin saja mengirim *email* ke alamat yang sama pada waktu yang bersamaan. Mereka harusnya tidak ditolak karena penggunaan *double spend database*. Namun jika masing-masing dari mereka menggunakan nilai acak ini, maka cap mereka akan berbeda.

7. Sufiks yang merupakan bilangan sebenarnya yang dihasilkan algoritma. Diberikan 6 bagian pertama, si pengirim harus mencoba segala kemungkinan nilai-nilai sufiks ini secara sekuensial untuk menghasilkan cap yang jika di-*hash* akan menghasilkan nilai *hash* yang memiliki bit nol pada awal nilai sejumlah yang ditentukan pada bagian 2.

4.2 Cara Kerja Hashcash pada *Email*

Dalam dunia yang ideal, semua pengirim akan menyertakan token hashcash pada pesan mereka. Penerima akan mengecek semua validitas yang diperlukan saat menerima pesan. Tapi dalam kenyataannya, hashcash tidak secara luas dipergunakan dengan mekanisme demikian. Bagaimanapun, memulai menggunakan hashcash (baik untuk pengirim maupun penerima) tidak akan merusak kakas bantu apapun pada system *email* yang sudah ada. Dengan kata lain, tidak ada yang harus ditakutkan untuk menggunakan mekanisme hashcash pada *email*.

Untuk mencap pesan keluar, tinggal menambahkan secara sederhana baris *header* pada *email*: satu *header* `X-Hashcash` untuk setiap resipien/penerima `To:` atau `Cc:` dari *email*. Misalkan, seseorang ingin mengirim pesan ke `mertz@gnosis.cx`, maka ia akan menyertakan *header* seperti ini:

```
X-Hashcash:
1:20:040927:mertz@gnosis.cx::odVZhQM
P:7ca28
```

Biasanya, MUA (*mail user agents*), *filter*, atau MTA (*mail transport agents*) akan melakukan pekerjaan ini ketimbang meminta pengguna untuk melakukannya secara manual. Melakukan hal tersebut secara manual, bagaimanapun, tidak akan terlalu sulit, setidaknya secara eksperimental. Cara mengecek sebuah cap dimulai dengan mencari nilai hash dari cap tersebut, misal dengan cara:

```
$ echo -n
1:20:040927:mertz@gnosis.cx::odVZhQM
P:7ca28 | sha
00000b50b85a61e7ba8ac4d5fed317c73770
6ae5
```

Pada contoh tersebut dapat dilihat bit nol yang berada di depan nilai hash (tiap digit heksa serupa dengan 4 bit). Tentu saja, dapat pula diperkuat dengan mengecek apakah sumber daya yang dimaksud adalah sumber daya yang dikenal (missal, salah satu dari alamat kontak), lalu mengecek bahwa cap belum pernah digunakan sebelumnya, dan tanggal pada cap valid. Selanjutnya, cap yang valid seharusnya mempunyai jumlah bit nol sebanyak yang sudah ditentukan (20 bit adalah semi-standard yang berlaku, meskipun akan berubah suatu saat mengikuti hukum Moore).

Untuk menghasilkan cap dengan 20-bit nol hanya memerlukan waktu komputasi sekitar satu detik. Bukan sebuah nilai yang besar untuk pengguna biasa yang hanya mengirim beberapa lusin *email* dalam sehari. Namun beberapa tambahan waktu komputasi akan menghambat para *spammer* yang ingin mengirim jutaan pesan sehari. Faktanya, dalam sehari hanya terdapat 86.400 detik. Bahkan jika para *spammer* menggunakan mesin *zombie* yang mereka infeksikan dengan *trojan*, meminta cap hashcash secara individual setidaknya akan memperlambat arus keluar dari mesin komputer *zombie* tersebut. Sementara, untuk mengecek cap hanya

membutuhkan waktu dalam hitungan mikro detik.

Sebaliknya, menambahkan proses menghasilkan hashcash dan proses pengecekan pada MUA – tidak seperti penanganan anti-spam lainnya – tidak mempunyai efek negative kepada siapapun. Untuk penerima yang tidak menggunakan aturan ini, cap hanya berupa tambahan *header* yang dapat diabaikan dengan mudah. Untuk pengirim yang tidak menambahkan cap hashcash, penerima yang mengecek `X-Hashcash:` tidak mempunyai apapun untuk dicek. Jika pengirim tidak menambahkan cap, maka sebaiknya pesan tersebut tidak dilihat.

Sebuah MUA atau sistem penyaring *spam* yang baik mungkin akan me-*whitelist email* yang cap hashcash-nya valid. Sebuah pendekatan berbasis hashcash untuk proses *whitelisting* adalah suatu peningkatan pada system dengan mekanisme interaktif seperti TMDA – pesan tantangan tidak hilang pada saat dikirim-balik, dan pengirim tidak lupa untuk merespon tantangan. Respon tantangan terletak pada pesan aslinya (*original message*) sebagai cap hashcash.

5. Keuntungan dan Kerugian Hashcash

Sistem hashcash memiliki keuntungan untuk usulan *micropayment* yang mana diterapkan untuk melegitimasi *email* yang tidak melibatkan uang sungguhan. Pengirim maupun penerima tidak perlu membayar, sehingga permasalahan administratif pada sistem ini dapat dihindari.

Sebaliknya, selama hashcash membutuhkan sumber daya komputasi yang signifikan untuk dialokasikan pada setiap *email* yang dikirim, maka akan sulit untuk digunakan pada *low-end* atau *battery-powered hardware* tanpa bantuan *server* eksternal.

Hashcash juga sangat sederhana untuk diimplementasikan pada MUA (*mail user agents*) dan system penyaring *spam*. *Server* terpusat tidak dibutuhkan. Hashcash dapat secara bertahap digunakan – *header* Hashcash tambahan dapat diabaikan pada system *email* yang tidak memakai mekanisme hashcash.

Satu problem yang cukup vital dari hashcash adalah, bahwa tidak jelas apakah ada suatu parameter yang efektif semisal parameter yang mengatakan pengirim ini orang baik dan pengirim itu orang jahat. Estimasi paling mungkin sampai pada konklusi bahwa hanya satu dari dua hal berikut yang dapat diperoleh: *email* yang baik akan ditolak karena kekurangan kekuatan pemrosesan pada sisi pengirim; atau *email* yang buruk akan tetap masuk.

Alasan dari hal ini adalah *botnet* atau *cluster farm* yang mana para *spammer* dapat meningkatkan kekuatan pemrosesan mereka secara drastis, atau topologi *email* terpusat seperti milis, pada mana beberapa *server* adalah untuk mengirim sejumlah besar *email* yang legal.

Kebanyakan dari isu ini dipetakan semisal *botnet* akan kadaluwarsa lebih cepat karena pengguna mengetahui *load CPU* yang tinggi dan mengambil langkah penanggulangan, dan *server* milis dapat didaftarkan ke dalam *whitelist* dari *hosts subscribers* dank arena itu akan melewati tantangan hashcash. Tapi mereka merepresentasikan suatu hambatan yang cukup besar untuk penerapan hashcash.

Masalah lainnya adalah bahwa komputer terus berkembang menjadi lebih cepat mengikuti hukum Moore. Sehingga kesulitan perhitungan yang diperlukan harus ditingkatkan seiring dengan waktu. Bagaimanapun, Negara berkembang dapat diperkirakan menggunakan *hardware* yang lebih tua, yang berarti mereka akan kesulitan untuk ikut berpartisipasi dalam system *email* menggunakan hashcash. Ini juga terlihat pada individu yang berpendapatan lebih rendah pada Negara berkembang yang tidak dapat membeli perangkat keras baru.

Masalah teoritis yang timbul karena peningkatan secara terus menerus dari kesulitan perhitungan adalah batas jumlah bilangan dari bit yang terdapat pada pesan SHA-1. Jika tren dalam kekuatan pemrosesan saat ini terus berlanjut, maka 160 bit yang tersedia pada SHA-1 akan habis terpakai semua dalam kurun waktu sekitar 200 tahun atau lebih. Hal ini karena komputer pada masa tersebut akan mampu untuk melakukan pencarian secara *brute-force* terhadap semua kemungkinan jumlah bit nol nilai hash SHA1 dalam waktu sedetik.

Namun sepertinya masalah ini dapat secara mudah diselesaikan dengan berpindah menggunakan algoritma yang menyediakan nilai hash yang lebih lebar seperti SHA-512.

6. Pemanfaatan Hashcash untuk Tujuan Lainnya

Hashcash terbukti sangat bermanfaat untuk tantangan non-interaktif. Tapi tidak ada alasan mengapa hashcash tidak dapat digunakan dalam konteks interaktif sebaik pada non-interaktif. Selama kakas bantu terus ditambah untuk mendukung hashcash -- terutama untuk aplikasi multiguna seperti Mozilla *suite* – maka akan menjadi lebih mudah untuk menggunakan hashcash secara netral untuk keadaan interaktif maupun non-interaktif.

Sebagai contoh, jika Thunderbird *mailer* memiliki API *calls* untuk komputasi hashcash, maka secara kasar akan mudah untuk sepupunya, Firefox *web browser*, untuk merespon pada tantangan interaktif dengan menggunakan API yang sama untuk menghasilkan cap hashcash.

6.1 Memroteksi Wiki

Satu penerapan hashcash di luar konteks *email* adalah untuk menghindari *defacement* yang seringkali dialami system Wiki. Karena Wiki secara umum terbuka bagi siapapun untuk berkontribusi, cacat pada komunitas Wiki adalah suatu program yang me-*crawling* Wiki dan menambahkan *link* komersial yang tidak relevan pada konten-konten Wiki.

Serangan tersebut kadang memaksa system Wiki meminta pengguna Wiki untuk memakai suatu *account* dalam melakukan penyuntingan konten-konten Wiki. Account ini diberikan berdasarkan pada tantangan *email* otomatis yang harus dikirim-balik untuk membuktikan bahwa penyunting mendapatkan suatu kunci acak. Namun, kebutuhan *account* tersebut tampaknya malah bertentangan dengan filosofi dari sistem Wiki.

Menambahkan suatu tantangan hashcash tidak mencegah *defacement* otomatis terhadap situs Wiki, namun dapat membuat program *crawling* terlarang berjalan dengan lambat. Jika me-*deface* satu situs memerlukan beberapa detik lebih lama dibandingkan hanya kurang dari sedetik, maka para pelaku akan berpikir dua kali untuk melakukan *defacement* tersebut. Faktanya, untuk pemanfaatan hashcash semacam ini, sebaiknya menggunakan lebih dari 20 bit nol nilai hashcash.

Kebanyakan kita berpikir bahwa penerapan delay waktu untuk mengonfirmasi penyuntingan dari suatu Wiki akan mempunyai efek yang sama dengan penerapan hashcash. Tapi faktanya, penerapan semacam itu mempunyai kelemahan. Sebuah mesin *crawler* dapat memaralelkan proses *defacement* yang dilakukannya – jika setiap penyuntingan menerapkan lima detik waktu delay, maka sebagai contoh, mesin tersebut dapat secara mudah memakai lima detik tersebut untuk menginisiasikan penyuntingan pada halaman Wiki lainnya. Dengan meminta utilisasi aktif *CPU*, seperti pada hashcash, vandalisasi tersebut tidak dapat lagi berjalan secara paralel.

Tantangan yang diajukan Wiki dapat berupa tantangan interaktif maupun tantangan non-interaktif. Sangat dimungkinkan bagi sebuah situs Wiki untuk mengarahkan pengguna ke halaman tantangan sebelum meneruskan ke halaman sunting yang sebenarnya. Suatu sumber daya acak dapat dimunculkan sebagai tantangan pada halaman ini.

Namun, pendekatan yang lebih baik adalah dengan menghilangkan kebutuhan berinteraksi. Sebagai contoh, pada sebuah system Wiki yang berjalan, pengguna kemungkinan melakukan penyuntingan konten dengan menggunakan URL seperti berikut:

```
http://somewhere.net/wiki?action=edi
t&id=SomeTopic
```

Sistem Wiki yang berada di bawah suatu perlindungan hashcash kemungkinan menggunakan URL yang berbeda, semisal:

```
http://somewhere.net/wiki?stamp=1:24
:040928:SomeTopic:edit:KG4E9PaK2VLjK
M2Z:0000Zbrc
```

Server Wiki akan mengecek cap yang disertakan pada URL tersebut sebelum mengizinkan pengguna melakukan penyuntingan. Tapi, penyuntingan tidak membutuhkan membuat *account* atau menyertakan suatu informasi personal. Lebih jauh lagi, pengecekan *double spending* dan masa berlaku menjamin suatu nilai *interest* dalam penyuntingan. Tidak sulit memang untuk menghasilkan URL di atas dengan menggunakan perintah:

```
hashcash -mCb 24 -x edit
SomeTopic
```

Bagaimanapun, idealnya, sebuah *web browser* akan memilih untuk menghasilkan cap pada proses *background* untuk menjamin delay waktu. Misalkan, URL di atas diciptakan dalam suatu cache saat pengguna membaca konten:

```
http://somewhere.net/wiki?SomeTo
pic
```

Kemungkinan besar, beberapa cap penyuntingan lainnya dapat juga di-cache untuk halaman-halaman yang terhubung dengan halaman Wiki yang sedang diakses.

6.2 Melakukan *Benchmark* terhadap Sumber Daya *CPU*

Pemanfaatan hashcash secara interaktif dapat diterapkan pada kasus pemrosesan terdistribusi. Proyek seperti Great Internet Mersenne Prime Search (GIMPS) dan SETI@home, dan *task* semisal *folding* protein dan *puzzle* kriptografi, terkadang ditugaskan kepada sejumlah besar mesin komputer sukarelawan (*volunteer machine*). Setiap sukarelawan tinggal mengunduh beberapa kode, lalu menjalankan bagian tertentu dari *task* yang lebih besar, dan mengirim balik hasil komputasi sementara ke *server* pusat. Pekerjaan ini adalah contoh dari pemanfaatan pemrosesan *CPU*.

Semua *task* yang terdistribusi terbuka bagi siapapun untuk menjadi sukarelawan. Namun tidak sulit membayangkan suatu *task* yang membutuhkan koordinasi tingkat tinggi mengalami kegagalan menjalankan *task* tersebut dalam waktu yang ditentukan. Akibatnya, hal ini mungkin berpengaruh pada proses komputasi lainnya dan mungkin dapat membahayakan komputasi secara keseluruhan.

Pada kasus seperti ini, *server* tentu akan meminta komputer calon sukarelawan untuk memenuhi kemampuan minimal tertentu sebelum ikut berpartisipasi pada proses komputasi. Selama pengecekan kecepatan menggunakan tipe komputasi tertentu dapat lebih presisi, hashcash tetap memberikan mekanisme *benchmark* terhadap *CPU* secara umum. SHA-1 adalah tipikal komputasi matematis. Dan jika kandidat sukarelawan sudah meng-*install* hashcash, maka menjawab suatu tantangan hashcash dapat berarti prasyarat minimum yang harus dipenuhi sebelumnya.

Trik dari pengecekan kemampuan *CPU* adalah dengan meminta nilai bit yang tinggi dengan waktu kadaluwarsa yang singkat. Hanya *CPU* yang cukup cepatlah yang dapat menjawab tantangan yang dilemparkan. Agar hal ini berjalan, nama sumber daya harus diberikan secara semi-interaktif – jika tidak, suatu kandidat dapat hanya mengirim cap tanggal untuk memberikan pandangan yang salah dari pembuatan secara cepat.

Sebagai contoh, suatu Pentium III yang cepat atau G4 yang lambat dapat menghasilkan suatu cap hashcash 20-bit dalam waktu kurang dari sedetik, namun sebuah Pentium-II atau G3 tidak dapat. Kita dapat meminta tantangan hashcash 32-bit yang harus diselesaikan dalam waktu satu jam kepada komputer kandidat sebagai tes penyaringan awal. Kandidat mengirim sebuah pesan seperti, "Send me a challenge"; *server* yang bersesuaian akan merespon dengan, "The time is 040927124732; your challenge resource is a37tQk." Jika *server* mendapatkan hash yang valid pada pukul 13:47 hari itu, maka kandidat dinyatakan lulus uji.

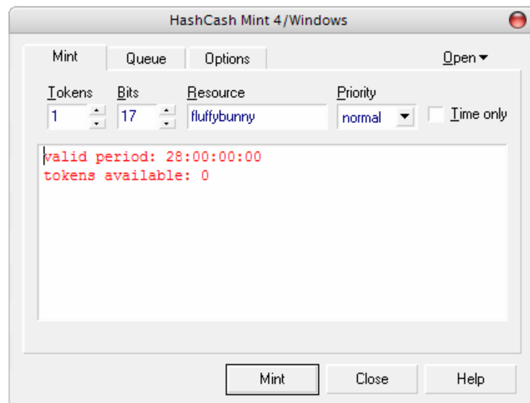
Namun, protokol yang diajukan ini tidak menjamin bahwa pekerjaan komputasi dapat terselesaikan dengan baik. Bahkan komputer yang cepat pun terkadang dapat terdiskoneksi. Dan pengguna dapat merubah pandanfannya mengenai menjalankan perangkat lunak yang terdistribusi. Tapi setidaknya suatu metode kualifikasi yang cukup baik dapat diterapkan.

7. Contoh Aplikasi Implementasi Hashcash

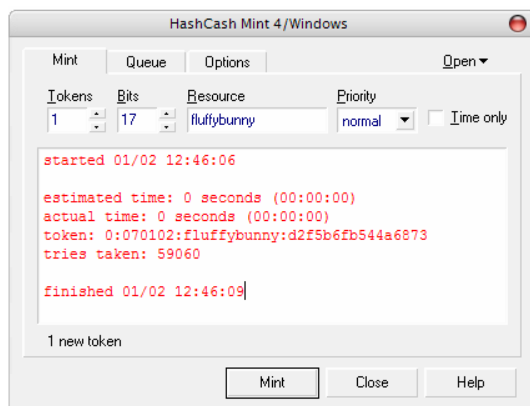
Berdasarkan keterangan [2], hashcash sudah dicoba dijalankan pada **dizum.com** dan **shinn.net** *mail2news gateway* sebagai kontra-penanggulangan terhadap USENET *flooding*.

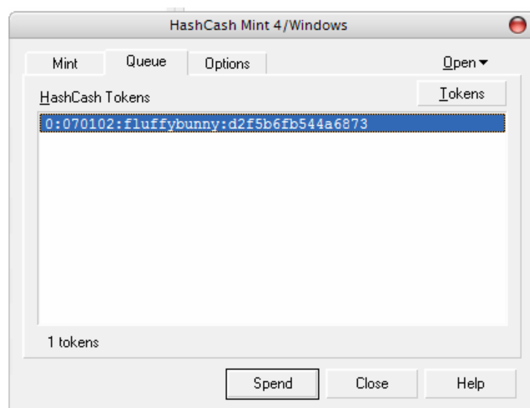
Contoh tampilan aplikasi simulasi hashcash yang dibuat Adam Back adalah sebagai berikut:

(dapat diperoleh pada http://www.cypherspace.org/hashcash/hcw/)

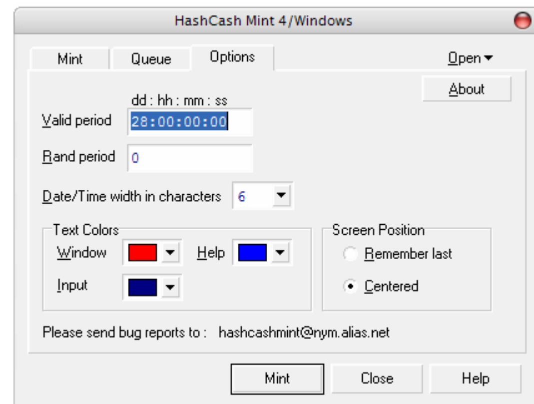


Tampilan hasil *minting* hashcash:





opsi-opsi yang ada pada aplikasi:



8. Simpulan

Hashcash adalah suatu mekanisme untuk mencegah *spamming* dengan melakukan suatu tantangan kepada pengirim *email* untuk melakukan proses komputasi sebagai bukti pekerjaan (*proof-of-work*). Sumber daya yang dibutuhkan untuk melakukan proses komputasi ini menjadi pembatas bagi *spammer* untuk mengirim *email* dalam jumlah massal.

Hashcash bekerja didasari dengan prinsip *cost-function* yang non-interaktif, *publicly auditable*, *trapdoor-free* dengan *unbounded probabilistic cost*.

Pada dunia nyata, hashcash sudah dicoba dijalankan pada **dizum.com** dan **shinn.net** *mail2news gateway* sebagai kontra-penanggulangan terhadap USENET *flooding*.

Berbagai karakteristik hashcash membuatnya dapat diterapkan untuk beberapa domain permasalahan lain yang membutuhkan karakteristik tersebut. Dua contoh penerapan hashcash pada domain lainnya adalah untuk memroteksi situs Wiki dari penyuntingan yang mengandung unsur komersialisasi, dan sebagai *benchmark* sederhana pada perangkat lunak yang bekerja secara terdistribusi.

DAFTAR PUSTAKA

[1] Back, Adam. HashCash. http://www.hashcash.org. Diakses pada tanggal 11 Desember 2006.

[2] Back, Adam. Hashcash.org. http://www.cypherspace.org/hashcash/. Diakses pada tanggal 11 Desember 2006.

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

[4] Back, Adam. (2002). Hashcash - A Denial of Service Counter-Measure.

[5] Mertz, David. Charming Python: Beat spam using hashcash. http://www.ibm.com/developerworks/linux/l-hashcash.html. Diakses pada tanggal 11 Desember 2006.