Implementasi CubeHash dalam Digital Signature dan Perbandingannya dengan Fungsi Hash Lain

William Eka Putra - 13508071
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
striker_system@hotmail.com

Abstrak*— Fungsi hash merupakan salah satu fungsi yang banyak dipakai di dalam kegiatan sehari-hari karena sifatnya yang mentransformasi pesan sepanjang apapun menjadi sebuah digest yang berukuran kecil dan dapat digunakan untuk aplikasi lainnya, contohnya adalah Digital Signature, penyimpanan password, dan masih banyak lagi. CubeHash merupakan salah satu varian fungsi hash yang diajukan oleh Daniel J. Bernstein pada saat kompetisi NIST hash function. Sedangkan Digital Signature adalah salah satu cara pembuktian keaslian atau autentikasi sebuah dokumen yang dikirim. Seperti layaknya sebuah tanda tangan, Digital Signature dapat berbeda-beda dan unik untuk setiap orang. Dengan mengaplikasikan fungsi hash dalam Digital Signature, penulis mencoba untuk membuat sebuah peningkatan kemangkusan dalam pembubuhan Digital Signature.

***Index Terms*—CubeHash, *Digital Signature*, autentikasi.**

I. PENDAHULUAN

Keaslian dalam suatu dokumen sekarang ini merupakan salah satu hal yang cukup penting dalam kehidupan sehari-hari, satu saja kata diganti bisa mengubah hasil secara signifikan, contohnya dalam hukum atau perjanjian kontrak. Oleh karena itu autentikasi sangat diperlukan untuk setiap dokumen, *Digital Signature* merupakan salah satu solusinya, ditambah dengan fungsi hash yang mempersingkat jumlah karakter membuat *Digital Signature* marak digunakan dalam dunia kerja.

Fungsi hash merupakan prosedur deterministik yang mengambil blok data tertentu dan mengembalikan bit string yang memiliki ukuran tetap yang cukup sensitif terhadap perubahan data meskipun hanya sedikit sehingga penggantian atau pemalsuan data akan mengubah nilai hash secara total. Setelah dilakukan hash, data tersebut akan berubah menjadi sebuah pesan yang terenkripsi atau biasa kita sebut sebagai *message digest*. Penggunaan fungsi hash ini tidak hanya terbatas dalam autentikasi pesan atau dokumen saja tapi juga dapat digunakan sebagai penyingkat dokumen yang cukup panjang misalnya adalah data biometrik seperti sidik jari atau DNA, selain itu fungsi hash ini juga dapat digunakan sebagai sebuah fungsi sederhana, misalnya untuk memeriksa redundansi atau replikasi data, atau melakukan pengecekan data apakah telah lengkap atau rusak, metode ini biasa disebut sebagai metode *checksum*.

Fungsi hash yang cukup terkenal dan digunakan oleh banyak orang adalah *Secure Hash Algorithm* atau disingkat sebagai SHA. SHA merupakan fungsi yang masih terus berkembang sampai sekarang ini dengan cara mengadakan kompetisi yang diselenggarakan oleh National Insitute of Standards and Technology (NIST). Generasi terbaru pada saat ini adalah SHA-3 yang merupakan suksesor dari SHA-1 dan SHA-2. CubeHash merupakan salah satu varian dari SHA-3 yang diajukan oleh Daniel J. Bernstein saat kompetisi NIST.

II. CUBEHASH

CubeHash merupakan salah satu fungsi hash yang diikutsertakan dalam kompetisi fungsi hash NIST, diajukan oleh Daniel J. Bernstein, CubeHash ini lolos dalam tahap pertama NIST tapi saying sekali tidak terpilih sebagai lima besar finalis kompetisi tersebut.

Blok-blok pesan dilakukan XOR dengan initial bits 128 byte. Pada CubeHash yang memiliki pola r/b-h dihasilkan *digest* sepanjang h-bit dan panjang *message* sesuai dengan panjang pesan yang ada. Parameter r dan b memberikan informasi mengenai keamanan dan performa dari fungsi hash itu sendiri yang pada akhirnya bergantung kepada hasilnya atau *message digest*.

Parameter yang dimiliki oleh CubeHash dan dengan definisi dari isinya adalah sebagai berikut:

- parameter *r*, yaitu jumlah ronde (*round*) yang digunakan, didefinisikan dalam {1,2,3,...}
- parameter *b*, yaitu jumlah blok pesan dalam satuan byte, didefinisikan dalam {1,2,3,...,128}
- parameter *h*, yaitu jumlah keluaran hash dalam satuan bits, didefinisikan dalam {8,16,24,...,512}

Parameter *r* yang lazim digunakan adalah 16, sedangkan untuk *b* dan *h* yang biasa digunakan masing-masing adalah 32 dan 512. Sedangkan untuk ukuran pesan, tidak ditentukan selama didefinisikan dalam *string* dengan ukuran antara 0 bit sampai dengan 2^{128}-1 bit.

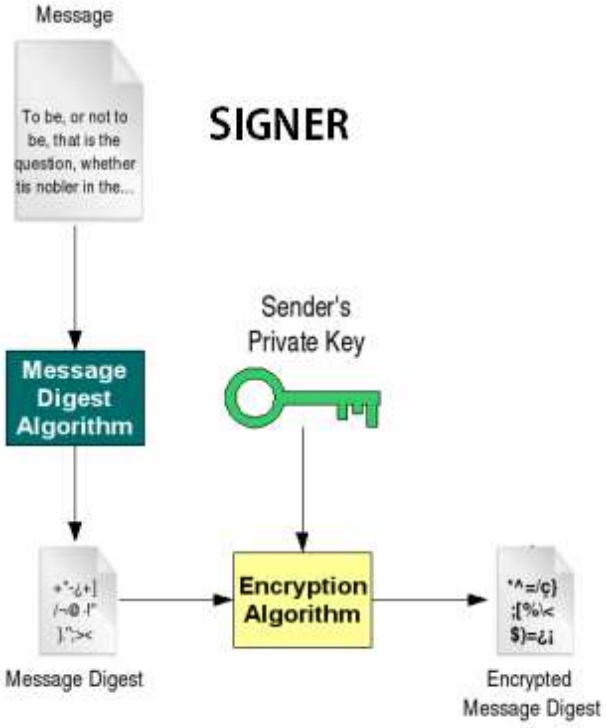
III. DIGITAL SIGNATURE

Sejak zaman dahulu, tanda-tangan sudah digunakan untuk autentikasi dokumen cetak. Tanda-tangan mempunyai karakteristik sebagai berikut:

- Tanda-tangan adalah bukti yang otentik.
- Tanda tangan tidak dapat dilupakan.
- Tanda-tangan tidak dapat dipindah untuk digunakan ulang.
- Dokumen yang telah ditandatangani tidak dapat diubah.
- Tanda-tangan tidak dapat disangkal (*repudiation*).

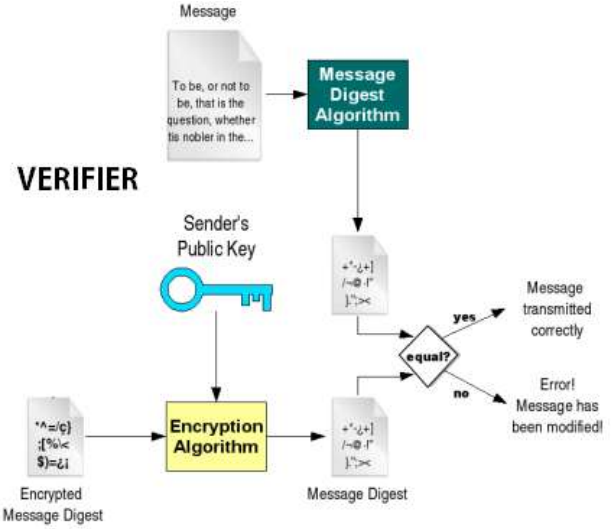
Fungsi tanda tangan pada dokumen kertas juga diterapkan untuk otentikasi pada data digital (pesan, dokumen elektronik). Tanda-tangan untuk data digital dinamakan tanda-tangan digital atau *Digital Signature*. *Digital Signature* bukanlah tulisan tanda-tangan yang di-digitisasi (di-*scan*).

Dalam *Digital Signature* biasanya digunakan dua tahap penenkripsian pesan, yang pertama di pihak pengirim atau *sender* atau *signer*, pesan diubah menjadi *message digest*, kemudian menggunakan kunci privat yang bersifat rahasia *message digest* diubah menjadi *message digest* yang telah terenkripsi. Secara skematis ditunjukkan sebagai berikut:



Gambar 1. Skema Pengiriman Digital Signature

Setelah message digest yang telah dienkripsi dan pesan telah dibuat, dokumen itu dikirimkan ke orang yang dituju yaitu *receiver* atau *verifier* dan telah memiliki kunci public yang bersifat tidak rahasia. *Verifier* melakukan perubahan pesan menjadi *message digest* dan mendekripsi *encrypted message digest* yang telah dikirimkan. Apabila hasil pencocokan kedua *message digest* sama maka dokumen tersebut adalah asli, bila tidak berarti dokumen tersebut telah diganti atau rusak. Secara skematis ditunjukkan sebagai berikut:



Gambar 2. Skema Penerimaan Digital Signature

Metode yang telah penulis pelajari adalah menggunakan fungsi hash untuk message digest dan menggunakan RSA untuk enkripsi dekripsi pesan. Dalam makalah ini RSA tetap digunakan dan fungsi hash yang dipakai adalah yang hendak penulis implementasikan yaitu CubeHash.

IV. IMPLEMENTASI CUBEHASH

Dalam bab ini hanya akan dibahas cara pengimplementasian secara algoritmik, karena penimplementasian secara program akan dijelaskan di bab selanjutnya bersama dengan program *Digital Signature* dalam satu program.

Pengimplementasian algoritma CubeHash memiliki lima langkah utama, yaitu:

1. Inisialisasi *state* berukuran 128-byte (1024 bit) sebagai fungsi dari (*h*, *b*, *r*)
2. Mengubah pesan masukan menjadi *padded message*, yang terdiri dari satu atau lebih blok sebesar b-byte.
3. Untuk setiap blok b-byte pada *padded message*, block dilakukan XOR dengan b-byte pertama dari *state*, dan kemudian *state* diubah melalui sejumlah r *round* yang identik.
4. Finalisasi *state*
5. Mengembalikan h/8 byte pertama dari *state* dipakai sebagai *output*.

Pembuatan kode program menggunakan langkah-langkah di bawah sebagai acuan (kode program yang mengimplementasikan CubeHash terlampir).

A. INISIALISASI

Proses pertama adalah inisialisasi, yaitu:

1. 128-byte state tersebut dilihat sebagai urutan (sekuens) dari 32 buah variabel sebesar 4-byte yang didefiniskan sebagai x_{00000}, x_{00001}, x_{00010}, x_{00011}, x_{00100}, … ,x_{11111}, setiap variabel diinterpretasikan dalam bentuk *little-endian* sebagai 32-bit *integer*.
2. Semua variabel dari state tersebut masing-masing diinisialisasi dengan nilai:
   - x_{00000} diinisialisasi dengan nilai h/8.
   - x_{00001} diinisialisasi dengan nilai b.
   - x_{00010} diinisialisasi dengan nilai r.
   - sisanya diinisialisasi dengan nilai 0.
3. *State* lalu ditransformasikan terbalik sebanyak 10*r* ronde yang identik.

B. PADDING

Proses padding ini merupakan langkah kedua dan ketiga, prosesnya adalah:

1. Menggabungkan 1 bit ke dalam pesan.
2. Kemudian menggabungkan lagi sesedikit mungkin angka 0 bit untuk mencapai panjang kelipatan 8b bit.

C. FINALISASI DAN TRANSFORMASI

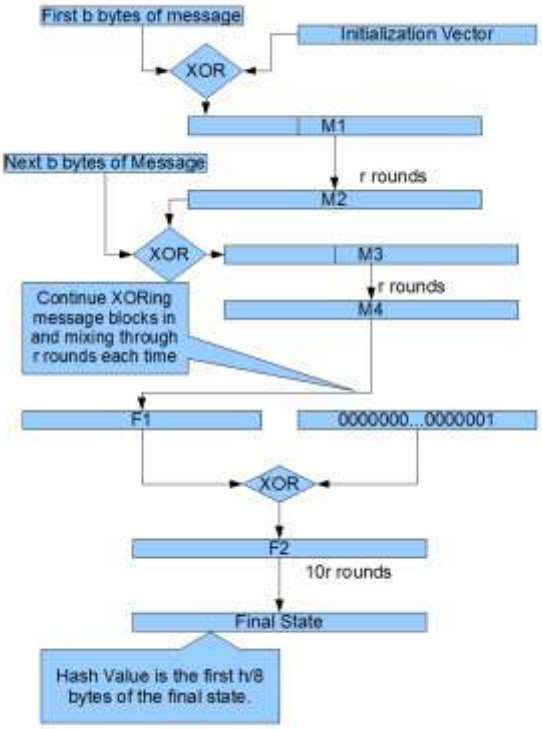
Finalisasi dan transformasi merupakan dua langkah terakhir dalam pengimplementasian CubeHash, proses dan cara kerjanya adalah:

- State terakhir x_{11111} di XOR dengan 1
- Tambahkan x_{0jklm} ke x_{1jklm} modulo 232, untuk setiap (j, k, l, m).
- Putar x_{0jklm} ke atas sebesar 7 bit, untuk setiap (j, k, l, m).
- Tukar x_{00klm} dengan x_{01klm}, untuk setiap (k, l, m).
- XORx_{1jklm} ke x_{0jklm}, untuk setiap (j, k, l, m).
- Tukar x_{1jk0m} dengan x_{1jk1m}, untuk setiap (j, k, m).
- Tambahkan x_{0jklm} ke x_{1jklm} modulo 232, untuk setiap (j, k, l, m).
- Putar x_{0jklm} ke atas sebesar 11 bit, untuk setiap (j, k, l, m).
- Tukar x_{0j0lm} dengan x_{0j1lm}, untuk setiap (j, l, m).
- XOR x_{1jklm} ke x_{0jklm}, untuk setiap (j, k, l, m).
- Tukar x_{1jkl0} dengan x_{1jkl1}, untuk setiap (j, k, l).

Tahap finalisasi adalah poin pertama yaitu melakukan XOR x_{11111} dengan 1, setelah itu sepuluh poin di bawahnya dapat disebut sebagai transformasi 10 ronde yang identic.

Sebagai tambahan, pada saat melakukan proses padding tidak diperlukan tempat penyimpanan (*storage*) yang terpisah untuk menampung panjang pesan atau blok yang akan diproses, dan variabel lainnya yang digunakan. Pada saat melakukan implementasi hanya dimungkinkan untuk menggunakan satu buah integer antara 0 dan 8*b* untuk mencatat jumlah bit yang telah diproses dalam blok yang sedang dikerjakan sekarang (*current block*).

Proses transformasi dilakukan dalam 10 ronde yang identik dan dapat dituliskan secara skematis seperti gambar berikut:



Gambar 3. Skema CubeHash

V. IMPLEMENTASI PROGRAM

Program yang dibuat adalah program *Digital Signature* yang telah ditambahkan fiturnya yaitu pada saat melakukan hash akan ditampilkan beberapa jenis fungsi hash dan waktu eksekusinya sehingga dapat dianalisis performanya masing-masing. Berikut adalah tampilan dari program yang penulis buat.

Gambar 4. Antarmuka Program Digital Signature

Implementasi dilakukan di dalam lingkungan yang sama sehingga kesalahan waktu pengukuran akibat perbedaan lingkungan tidak akan terjadi, yaitu dengan spesifikasi:

Bahasa : C#
Kakas : Visual Studio 2010 Ultimate
Processor : Intel® Core™2Quad 2.50 GHz
RAM : 4 GB

VI. PERBANDINGAN DAN ANALISIS EMPAT FUNGSI

Dalam bab ini akan dibahas mengenai empat fungsi yang telah disebutkan di atas dan akan di bandingkan dan dianalisis. Program ini memiliki waktu proses yang agak lama karena pembacaan dilakukan per bit dan digunakan empat jenis *message digest algorithm* yang berbeda-beda. Pengecekan performa dilakukan dengan file yang berbeda-beda jenis dan ukurannya dan tiap algoritma akan di uji dengan menggunakan waktu eksekusi.

Gambar 5. Contoh Hasil Eksekusi Performa

Empat buah algoritma yang dibandingkan performanya dan dianalisis hasilnya adalah:

- **CubeHash** (implementasi penulis)
- **MyHash** yang mengaplikasikan SHA-1 (implementasi penulis)
- **MD5** (tersedia di dalam Visual Studio 2010)
- **SHA-1** (tersedia di dalam Visual Studio 2010)

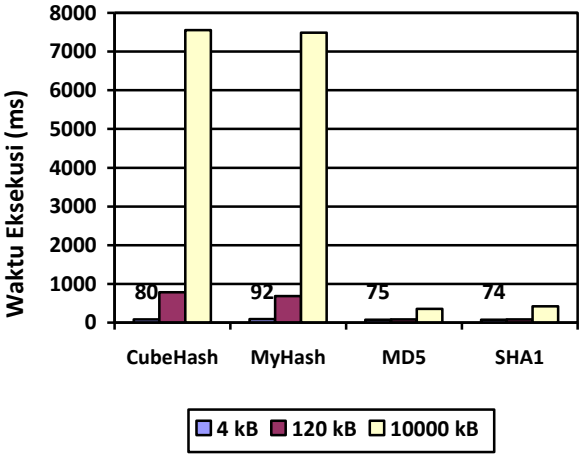
Berikut akan dilampirkan hasil eksperimen penulis dengan berbagai jenis file dan ukuran.

Tabel I. Perbandingan Performa Empat Fungsi Hash

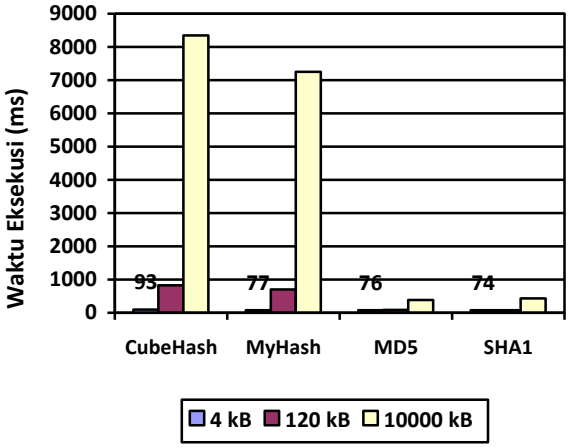
Jenis	Ukuran (kB)	Performa (ms)			
		Cube	MyHash	MD5	SHA-1
Teks (.txt)	4	80	92	75	74
	120	781	686	84	82
	10000	7554	7487	357	425
Gambar (.png)	4	93	77	76	74
	120	826	698	89	79
	10000	8345	7249	384	430
Zip File (.7z)	4	72	85	73	81
	120	813	736	78	72
	10000	8005	7348	405	419

Data uji adalah tiga jenis file yang berbeda dan masing-masing terdiri dari tiga jenis file yang berbeda cukup jauh ukurannya. File yang digunakan adalah file teks yang berisikan tulisan alfanumerik, file gambar yang memiliki ekstensi png dan file kompresi yang memiliki ekstensi 7z.

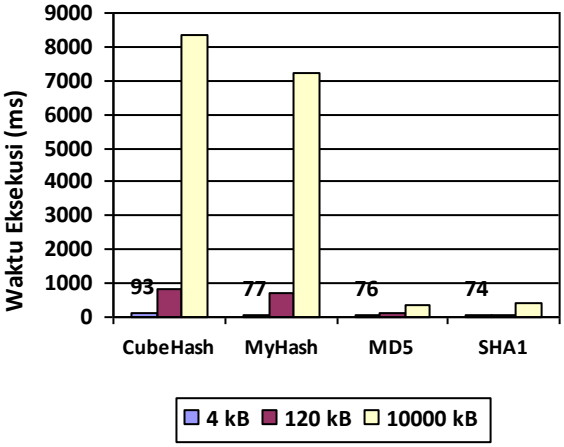
Berikut akan dibuat grafik dari data-data di atas agar mudah untuk menganalisis lebih lanjut dari apa yang bisa kita dapat dari percobaan ini.



Gambar 6. Grafik Waktu Ekseskusi untuk File Teks



Gambar 7. Grafik Waktu Ekseskusi untuk File Gambar



Gambar 8. Grafik Waktu Ekseskusi untuk File Kompresi (Zip)

Melihat tiga buah grafik di atas hasil percobaan, dapat dianalisis bahwa terjadi lonjakan waktu eksekusi untuk file yang lebih besar bagi CubeHash dan MyHash SHA-1 buatan penulis. Sedangkan MD5 dan SHA-1 yang terdapat di dalam kakas pengerjaan (Visual Studio 2010) tetap mengalami kenaikan tetapi tidak sesignifikan dua algoritma buatan penulis.

Hal ini mungkin terjadi karena pengimplementasian oleh penulis yang tidak efisien. Dengan mengesampingkan kedua algoritma yang tersedia dalam kakas, penggunaan algoritma CubeHash terbukti tidak terlalu efisien dalam menghasilkan *message digest* ditinjau dari performa yang dibandingkan dengan algoritma SHA-1 buatan penulis juga.

Waktu eksekusi untuk file berukuran kecil hampir tidak terlihat bedanya antara kedua algoritma hasil implementasi penulis, namun seiring dengan bertambahnya ukuran file yang dihash, semakin terlihat perbedaannya antara CubeHash dan SHA-1, yaitu CubeHash memakan waktu lebih lama untuk memproses file dalam ukuran besar.

Akan tetapi melihat dari kedua algoritma hasil implementasi penulis, tidak terlihat perbedaan yang cukup berarti dalam hal waktu eksekusi. Waktu yang dibutuhkan oleh keduanya seiring dengan meningkatnya ukuran file pun kurang lebih sama. Hanya saja CubeHash tertinggal namun tidak pernah tertinggal lebih dari satu detik. Sehingga dapat dikatakan performa CubeHash dan SHA-1 berada di dalam tingkat yang sama dengan mengabaikan faktor lain atau kesalahan implementasi yang dilakukan oleh penulis.

VII. Percobaan dan Analisis Mengenai CubeHash

A. SENSITIVITAS CUBEHASH

Dari hasil percobaan yang dilakukan, dapat dilihat bahwa CubeHash algoritma hasil implementasi penulis memiliki performansi yang kurang baik, apalagi jika dibandingkan dengan algoritma SHA-1 yang disediakan oleh kakas pembuatan yaitu Visual Studio 2010. Namun masalah keamanan CubeHash memiliki tingkat keamanan yang cukup tinggi.

CubeHash memiliki sensitivitas terhadap perubahan karakter walaupun hanya menambah spasi, atau mengubah huruf besar menjadi huruf kecil dan sebaliknya.

Misalnya sebuah kalimat yaitu: "Misteri Gambar Burung Nazca di Aztec" akan dihash, tanda kutip tidak disertakan, maka dihasilkan *digest* sebagai berikut:
6FE2896F10D7EB86B9FDB8FAB3ABC6DFE1C29F37

Penulis mencoba menambahkan spasi di akhir kalimat tersebut menjadi: "Misteri Gambar Burung Nazca di Aztec " dan dihash tanpa menyertakan tanda kutip, maka dihasilkan *digest* sebagai berikut:
8C3E81F3CE87695186DAB285AF6B20CE96923851

Kemudian dilakukan penggantian huruf yaitu huruf "A" pada "Aztec" akan dirubah menjadi "a" sehingga kalimatnya: "Misteri Gambar Burung Nazca di aztec" menghasilkan *digest* sebagai berikut:
AF6F06716467D81903934F4D074B0AE6DEE1CF70

Sebaliknya, huruf "i" pada kata "Misteri" diganti menjadi huruf capital sehingga kalimat menjadi: "MIsteri Gambar Burung Nazca di Aztec" akan menghasilkan *digest* sebagai berikut:
2FE10A73E0C670D29C7ED9846BB79A4FAB8454D5

Menghilangkan huruf "A" pada kata "Aztec" sehingga mengakibatkan kalimat menjadi: "Misteri Gambar Burung Nazca di ztec" menghasilkan:
22FF6F9BF1A0059E3B4AC44C31E77DEDAB9BFCE4

Terakhir akan dicoba menambahkan enter pada akhir kalimat, dan menghasilkan *digest* sebagai berikut:
07C76F1AAFA57367F4EBD75410EE0826656CC765

Dari berbagai percobaan yang penulis lakukan di atas yaitu memodifikasi kalimat dengan cara mengganti karakter, mengurangi karakter, dan menambah karakter, meskipun hanya satu karakter atau spasi dan enter, mengakibatkan perubahan *digest* yang cukup signifikan.

Tidak terlihat adanya pola yang mirip atau pengulangan yang sama sehingga dapat dikatakan CubeHash ini memiliki sensitivitas yang cukup tinggi terhadap modifikasi karakter pesan.

B. VARIASI CUBEHASH

CubeHash merupakan algoritma atau fungsi yang dapat memiliki beberapa varian yaitu dengan cara memodifikasi variabel r dan b pada tahap inisialisasi.

Salah satu varian yang diajukan dalam kompetisi fungsi hash NIST adalah CubeHash8/1 yaitu CubeHash dengan nilai r = 8 dan b = 1. Tidak hanya satu varian itu saja, sang pembuat Daniel J. Bernstein juga mengajukan beberapa variasi CubeHash dalam kompetisi NIST, contoh variasi CubeHash tersebut adalah:

- CubeHash16/32-224 untuk SHA-3-224
- CubeHash16/32-256 untuk SHA-3-256
- CubeHash16/32-384 untuk SHA-3-384-normal
- CubeHash16/32-512 untuk SHA-3-512-normal
- CubeHash16/1-384 untuk SHA-3-384-formal
- CubeHash16/1-512 untuk SHA-3-512-formal

Dikatakan bahwa algoritma CubeHash tercepat adalah CubeHash16/32 yang performanya dapat mengatasi SHA-256 dan SHA-512. Dengan performa yang cukup cepat, namun tingkat keamanannya bukanlah rendah, namun dapat dikatakan cukup baik.

Dengan mengetahui bahwa *b* adalah parameter yang menunjukkan jumlah blok yang dipergunakan dan *r* adalah jumlah ronde yang dilalui, maka dapat dianalisis bahwa tingkat keamanan algoritma CubeHash meningkat

seiring dengan meningkatnya jumlah ronde yang digunakan (semakin banyak perputaran akan semakin aman) yaitu nilai *r* yang semakin naik dan nilai *b* yang semakin turun, yaitu menggunakan blok sesedikit mungkin agar celah yang dihasilkan untuk para kriptanalisis juga semakin sedikit.

Jadi dapat dimisalkan bahwa CubeHash8/1-512 lebih aman (lebih kuat pengamanannya) daripada CubeHash 1/1-512, dan CubeHash 1/1-512 lebih aman daripada CubeHash 1/2-512, sehingga dapat dikatakan CubeHash yang paling lemah adalah CubeHash 1/128-*h*. Akan tetapi keamanan berbanding terbalik dengan waktu eksekusi atau performa. Makin aman sebuah algoritma CubeHash akan mengakibatkan semakin lama waktu pemrosesannya, hal ini dikarenakan banyaknya ronde yang harus dilewati untuk menghasilkan sebuah *digest*.

Berdasarkan hasil percobaan penulis, CubeHash16/32 memiliki rata-rata waktu eksekusi yang paling cepat dibanding varian lainnya, dan sesuai perkataan sang pembuat yaitu Daniel J. Bernstein, mengatakan bahwa CubeHash 16/32 merupakan yang tercepat di antara varian lainnya dan memiliki kecepatan proses kurang lebih 16 kali lebih cepat dibanding dengan CubeHash8/1.

C. KRIPTANALISIS PADA CUBEHASH

Seperti yang telah dikatakan di atas, tingkat keamanan CubeHash bergantung kepada dua parameter masukannya yaitu r dan b. CubeHash r/b akan menjadi mudah untuk diserang jika nilai b terlampau besar. Ambillah contoh CubeHash16/64, berarti algoritma ini membebaskan para kriptanalis untuk memakai 64 byte terakhir dari masukan untuk state tertentu. Apabila nilai b semakin besar, maka kesempatan untuk menyerang dan sedikit mendapatkan tabrakan atau *collision*. Namun hal ini dapat diatasi dengan cara menurunkan nilai b, berdasarkan studi literature tambahan, untuk CubeHash16/32, dimiliki pipe sebesar 768 bit, dan kemungkinan serangan menurun secara signifikan.

Masalah lain yang dapat dimanfaatkan untuk melakukan kriptanalisis adalah nilai r, apabila nilai r semakin kecil, maka tingkat kekompleksan algoritma CubeHash sangatlah rendah, misalnya CubeHash yang memiliki nilai r = 1 hanyalah memiliki satu kali transformasi dan akan mudah untuk di dekripsi oleh kriptanalis, sebaliknya apabila nilai r semakin meningkat, maka transformasi akan semakin kompleks dan tentu saja akan menjadi sulit dipecahkan. Akan tetapi ada sebuah harga yang harus dibayarkan apabila nilai r ditingkatkan, yaitu performa atau waktu eksekusi, semakin kompleks dan aman sebuah algoritma CubeHash maka akan semakin lama waktu yang dibutuhkan untuk menjalankan sebuah proses ini.

Setelah melakukan percobaan dan studi literatur lebih lanjut, dikatakan sepuluh ronde atau r = 10 sudah cukup untuk memproduksi algoritma yang kebal terhadap kriptanalisis kepada CubeHash. Serangan yang biasa digunakan adalah metode serangan linear atau Linear *Attack*, *Collision Attack* dan *Differential Attack*.

Linear Attack ini mencari *linear differential path* untuk melakukan serangan terhadap blok yang ada. Berikut adalah tabel kemungkinan terbaik ditemukannya *linear differential path* untuk setiap serangan:

Tabel II. Kemungkinan Terbaik ditemukannya Path bergantung pada nilai r dan b

r	b	max nb. it.	probability
1	64	3	2^3
	32	5	2^{32}
	16	5	
	8	5	
	4	5	
	2	7	2^{221}
	1	15	2^{1225}
2	64	3	2^{32}
	32	3	
	16	3	
	8	3	
	4	3	
	2	4	2^{221}
	1	8	2^{1225}
4	64	3	2^{189}
	32	3	
	16	3	
	8	3	
	4	3	
	2	4	2^{964}
	1	9	2^{2614}
8	64	3	2^{650}
	32	3	2^{830}
	16	3	
	8	3	2^{1009}
	4	3	
	2	5	2^{2614}
	1	5	

Melihat tabel di atas, analisis mengenai hubungan berhasilnya serangan terhadap nilai r dan b yang telah dikemukakan adalah benar. Terlihat bahwa untuk setiap nilai r yang semakin besar, kemungkinan path yang ditemukan sangatlah banyak dan membingungkan para kriptanalis untuk menentukan path mana yang harus digunakan, hal ini berlaku juga untuk setiap nilai b yang semakin kecil. Jadi dapat dikatakan bahwa CubeHash yang aman adalah yang memiliki nilai r yang tinggi sedangkan untuk performa adalah CubeHash yang memiliki nilai b yang cukup besar.

VIII. Pengaplikasian CubeHash dalam Digital Signature

CubeHash yang diaplikasikan dalam *Digital Signature* harus disesuaikan nilai r dan b sesuai dengan ukuran file dan preferensi pengguna, apakah pengguna mengutamakan performa, yaitu waktu proses yang cepat ataukah pengguna lebih mengutamakan keamanan yang harus dibayar dengan waktu proses yang lebih lama.

IX. Kesimpulan

Dari percobaan yang dilakukan dan implementasi program yang dibuat oleh penulis serta beberapa studi literature tambahan, dapat disimpulkan:

- CubeHash dapat digunakan untuk meningkatkan kemangkusan pembuatan *Digital Signature*.
- CubeHash dapat digunakan lebih efisien dibandingkan SHA-1 karena dapat dimodifikasi parameter r dan b dimana yang menentukan tingkat keamanan dan performa sesuai kebutuhan.
- Performa CubeHash dalam program implementasi masih di bawah program yang terdapat di dalam kakas pembuatan, hal ini dikarenakan kurang efisiennya pengimplementasian oleh penulis.
- CubeHash ini termasuk dalam algoritma yang cukup sensitive terhadap perubahan satu karakter meskipun hanya mengubah sedikit dan tidak nampak, misalnya penambahan spasi di akhir kalimat.
- Parameter r dalam CubeHash menentukan tingkat kekuatan keamanan, semakin besar semakin kuat, namun semakin lama pula waktu yang dibutuhkan untuk melakukan proses.
- Parameter b dalam CubeHash apabila semakin kecil (mendekati 1) maka akan mengurangi kemampuan serangan, namun apabila nilai b semakin besar, maka serangan akan semakin mudah dilakukan.
- Untuk mendapatkan CubeHash yang paling aman dan performa yang baik, harus disesuaikan tergantung kebutuhan pengguna dengan memperhatikan ukuran file yang hendak dijadikan digest, banyaknya file yang hendak diproses, jenis file, dan preferensi pengguna itu sendiri.

X. Ucapan Terima Kasih

Penulis mengucapkan terima kasih terutama kepada Tuhan Yang Maha Esa karena berkat anugerah yang diberikan-Nya makalah ini dapat diselesaikan. Penulis juga mengucapkan terima kasih kepada Bapak Ir. Rinaldi Munir, M.T. selaku dosen pengajar kuliah IF3058 Kriptografi karena berkat kuliah dan referensi yang diberikan oleh beliau makalah ini dapat disempurnakan.

References

[1] http://www.informatika.org/~rinaldi/Kriptografi/2010-2011/kripto10-11.htm
[2] http://en.wikipedia.org/wiki/CubeHash
[3] http://en.wikipedia.org/wiki/Digital_signature
[4] http://upload.wikimedia.org/wikipedia/commons/1/19/CubeHash
[5] http://cubehash.cr.yp.to/submission2/spec.pdf
[6] http://cubehash.cr.yp.to/submission/estimates.pdf
[7] http://cubehash.cr.yp.to/submission/tweak.pdf
[8] Murphy, Sean dan Fred Piper. 2002. *Cryptography: A Very Short Introduction*. Oxford University Press
[9] Brier, Eric dan Thomas Peyrin. 2009. *Cryptanalysis of CubeHash*. Ingenico: France.

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, 30 April 2011

William Eka Putra - 13508071

LAMPIRAN

Fungsi Transformasi

```
//Transform
void CB_Tranform(CB_hashed state)
{
	//kamus lokal
	uint32[] y = new uint32[16];

	//algoritma
	for (r = 0; r < _rounds; ++r) //untuk jumlah r yang telah ditentukan
	{
		//lakukan identical round (step by step dijelaskan)
		for (i = 0; i < 16; ++i)
		{
			state.blocks[i + 16] += state.blocks[i];
		}
		for (i = 0; i < 16; ++i)
		{
			y[i ^ 8] = state.blocks[i];
		}
		for (i = 0; i < 16; ++i)
		{
			state.blocks[i] = invert(y[i],7);
		}
		for (i = 0; i < 16; ++i)
		{
			state.blocks[i] ^= state.blocks[i + 16];
		}
		for (i = 0; i < 16; ++i)
		{
			y[i ^ 2] = state.blocks[i + 16];
		}
		for (i = 0; i < 16; ++i)
		{
			state.blocks[i + 16] = y[i];
		}
		for (i = 0; i < 16; ++i)
		{
			state.blocks[i + 16] += state.blocks[i];
		}
		for (i = 0; i < 16; ++i)
		{
			y[i ^ 4] = state.blocks[i];
		}
		for (i = 0; i < 16; ++i)
		{
			state.blocks[i] = invert(y[i],11);
		}
		for (i = 0; i < 16; ++i)
		{
			state.blocks[i] ^= state.blocks[i + 16];
		}
		for (i = 0; i < 16; ++i)
		{
			y[i ^ 1] = state.blocks[i + 16];
		}
		for (i = 0; i < 16; ++i)
		{
			state.blocks[i + 16] = y[i];
		}
	}
}
```

Fungsi untuk melakukan padding

```
//Padding
Byte[] CB_Padding(Byte[] input)
{
      //kamus lokal
      int CB_Length = input.Length + 1;
      Byte[] padded_out = new Byte[CB_Length];
      int i = 0;
      int b = (int) _b;

      //algoritma
      if (b > 1)
      {
            CB_Length += (b - (CB_Length % b));
      }

      while (i < input.Length)
      {
            padded_out[i] = input[i];
            i++;
      }
      padded_out[i] = 128;
      i++;

      while (i < CB_Length)
      {
            padded_out[i] = 0;
            i++;
      }

      return padded_out;
}
```

Di bagian main untuk melakukan finalisasi

```
//Main process; Finalize
//10x transform
u = (int)(128 >> (state.pos % 8));
u <<= 8 * ((state.pos / 8) % 4);
state.blocks[state.pos / 32] ^= u;
CB_Transform(state);
state.blocks[31] ^= 1;

for (i = 0;i < 10;++i)
{
      transform(state);
}

for (i = 0; i < state.hashbitlen / 8; ++i)
{
      hashval[i] = (Byte)(state.blocks[i / 4] >> (8 * (i % 4)));
}
```