Studi dan Implementasi Fungsi Hash Whirlpool

Firdi Mulia - 13507045[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]if17045@students.if.itb.ac.id

Abstrak*— Whirlpool adalah sebuah fungsi hash yang didesain oleh Vincent Rijment (co-creator dari Advanced Encryption Standard) dan Paulo S. L. M. Barreto yang pertama kali diperkenalkan tahun 2000. Fungsi hash ini telah direkomendasikan oleh NESSIE (New European Schemes for Signatures, Integrity and Encryption) dan diadopsi oleh International Organization for Standardization (ISO) dan International Electrotechnical Commission (IEC) sebagai bagian dari ISO/IEC 10118-3. Selama ini belum pernah ada serangan yang dilaporkan berhasil pada fungsi ini serta juga belum pernah ditemukan kemungkinan terjadinya collision pada fungsi hash ini. Tetapi baru- baru ini, Shirai dan Shibutani menemukan sebuah kelemahan dalam matrix difusi dari whirlpool. Walaupun kelemahan ini tidak banyak berpengaruh. Fungsi hash whirlpool merupakan fungsi hash satu arah yang menghasilkan 512 bit message digest dan bisa menerima pesan yang panjanganya kurang dari 2^{256} bit. Whirlpool terdiri dari pemanggilan fungsi yang diulang-ulang pada 512 bit block cipher yang menggunakan kunci 512 bit. Fungsi pembulatan dan jadwalnya didesain dengan strategi Wide Trail. Implementasi whirlpool pada 8 bit dan 64 bit prosesor menguntungkan khususnya dari struktur fungsi, yang tidak terikat pada platform manapun.

***Istilah Indeks*— Hash, Matriks Difusi, *Whirlpool*, *Wide Trail*.**

I. PENDAHULUAN

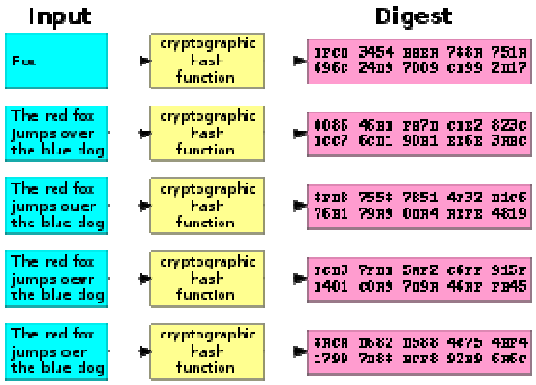
A. Fungsi Hash

Sebuah fungsi hash dalam kriptografi adalah fungsi deterministic yang mengambil sejumlah blok data dan mengembalikan string dengan panjang bit tetap, yang disebut dengan nilai hash, sehingga perubahan baik yang disengaja maupun tidak disengaja kepada data akan mengubah nilai hash. Data yang perlu dikodekan disebut pesan, dan nilai hash kadang-kadang disebut *message digest*.

Fungsi hash yang ideal mempunyai empat properti :

- Mudah untuk mengkomputasi nilai hash untuk pesan apapun
- Dari nilai hash, tidak bisa dikonversi kembali ke pesan
- Tidak mungkin untuk memodifikasi pesan tanpa mengubah nilai hash
- Tidak mungkin untuk memiliki dua pesan yang berbeda tetapi dengan nilai hash yang sama.

Fungsi hash memiliki banyak penerapan dalam keamanan informasi, khususnya dalam tanda tangan digital, Message Authentication Codes (MAC), dan bentuk otentifikasi lainnya.



Gambar 1. Skema dari fungsi hash

Fungsi hash juga bisa digunakan sebagai fungsi hash yang biasa, untuk menentukan indeks data dalam *hash table*, untuk pengolahan sidik jari, untuk mendeteksi duplikasi data atau secara unuik mengidentifikasi file, dan untuk mendeteksi data yang *corrupt*.

B. Fungsi Hash Whirlpool

Whirlpool merupakan fungsi hash yang didesain oleh Vicent Rijmen (co-crator dari Advanced Encription Standard) dan Paulo S. L. M. Barreto pada tahun 2000. Fungsi hash ini telah direkomendasikan oleh NESSIE (New European Schemes for Signatures, Integrity and Encryption) dan diadopsi oleh International Organization for Standardization (ISO) dan International Electrotechnical Commission (IEC) sebagai bagian dari ISO/IEC 10118-3.

Whirlpool didesain pada block cipher square. Whirlpool juga menggunakan konstruksi Miyaguchi-Preneel berdasarkan AES. Fungsi hash ini menerima pesan dengan panjang kurang dari 2^{256} bit dan menghasilkan 512 bit message digest. Penemu fungsi hash ini menyatakan bahwa whirlpool tidak (dan tidak akan) dipatenkan dan boleh digunakan secara gratis untuk tujuan apapun.

Penamaan algoritma ini adalah dari galaxi whirlpool di Canes Venatici. Dua program pertama yang mulai

menggunakan whirlpool adalah FreeOTFE dan TrueCrypt.

Gambar 2. Galaxy Whirlpool

II. DESKRIPSI PRIMITIF WHIRLPOOL

Primitif whirlpool adalah sebuah fungsi hash Merkle yang berdasarkan pada sebuah blok cipher, W, yang beroperasi pada sebuah 512 *hash state* menggunakan sebuah state kunci berkait, yang keduanya diturunkan dari data masukan. Pada subpokok bahasan berikutnya akan didefinisikan komponen pemetaan dan konstanta yang membangun whirlpool, kemudian dispesifikasikan fungsi hash yang lengkap untuk komponen-komponen ini.

A. Input dan Output

Status hash whirlpool secara internal dilihat sebagai sebuah matriks $M_{8x8}[GF(2^8)]$. Karena itu, 512 bit blok data (yang secara eksternal direpresentasikan sebagai larik byte dengan secara sekuensial mengelompokkan bit-bit dalam kelompok 8-bit) harus dipetakan ked an dari format matriks ini. Hal ini dapat dilakukan dengan fungsi μ : $GF(2^8)^{64} \rightarrow M_{8x8}[GF(2^8)]$ dan kebalikannya :

$$\mu(a) = b \leftrightarrow b_{ij} = a_{8i+j}, 0 \leq i,j \leq 7$$

B. Layer non-linear γ

Fungsi γ : $M_{8x8}[GF(2^8)] \rightarrow M_{8x8}[GF(2^8)]$ terdiri dari aplikasi paralel dari kotak substitusi non-linear : $GF(2^8) \rightarrow GF(2^8)$, x → S[x] untuk semua byte dari argument secara individual :

$$\gamma(a) = b \leftrightarrow b_{ij} = S[a_{ij}], 0 \leq i,j \leq 7$$

C. Permutasi siklik π

Permutasi π : $M_{8x8}[GF(2^8)] \rightarrow M_{8x8}[GF(2^8)]$ secara siklik berganti tiap kolom dari argument tersebut secara independen, jadi kolom j digeser ke bawah sejauh j:

$$\pi(a) = b \leftrightarrow b_{ij} = a_{(i-j) mod\ 8,j}, 0 \leq i,j \leq 7$$

D. Layer difusi linear θ

Layer difusi θ : $M_{8x8}[GF(2^8)] \rightarrow M_{8x8}[GF(2^8)]$ adalah pemetaan linear biasa yang berdasarkan pada [16,8,9] kode MDS dengan matriks generator Gc = [I C] dimana C = cir(01_x,01_x, 04_x, 01_x, 08_x, 05_x, 02_x, 09_x), yaitu :

$$C = \begin{bmatrix} 01_x & 01_x & 04_x & 01_x & 08_x & 05_x & 02_x & 09_x \\ 09_x & 01_x & 01_x & 04_x & 01_x & 08_x & 05_x & 02_x \\ 02_x & 09_x & 01_x & 01_x & 04_x & 01_x & 08_x & 05_x \\ 05_x & 02_x & 09_x & 01_x & 01_x & 04_x & 01_x & 08_x \\ 08_x & 05_x & 02_x & 09_x & 01_x & 01_x & 04_x & 01_x \\ 01_x & 08_x & 05_x & 02_x & 09_x & 01_x & 01_x & 04_x \\ 04_x & 01_x & 08_x & 05_x & 02_x & 09_x & 01_x & 01_x \\ 01_x & 04_x & 01_x & 08_x & 05_x & 02_x & 09_x & 01_x \end{bmatrix}$$

Tujuan dari π adalah untuk mendispersikan byte dari setiap baris diantara semua baris.

E. Kunci tambahan σ[k]

KUnci tambahan σ[k] : $M_{8x8}[GF(2^8)] \rightarrow M_{8x8}[GF(2^8)]$ terdiri dari penambahan bitwise (xor) dari sebuah matriks kunci k ε $M_{8x8}[GF(2^8)]$:

$$\sigma[k](a) = b \leftrightarrow b_{ij} = a_{ij} \oplus k_{ij}, 0 \leq i,j \leq 7$$

Pemetaan ini juga digunakan untuk mengenalkan konstanta putaran dalam jadwal kunci.

F. Fungsi pembulatan konstan cr

Konstanta pembulatan pada putaran ke-r, r > 0, adalah matriks c^r ε $M_{8x8}[GF(2^8)]$ yang didefinisikan sebagai

$$c_{0j}^r \equiv S[8(r-1)+j], 0 \leqslant j \leqslant 7,$$
$$c_{ij}^r = 0, \qquad 1 \leqslant i \leqslant 7, 0 \leqslant j \leqslant 7.$$

G. Fungsi pembulatan ρ[k]

Fungsi putaran ke-r memiliki pemetaan komposit ρ[k] : $M_{8x8}[GF(2^8)] \rightarrow M_{8x8}[GF(2^8)]$, yang diparameterisasikan dengan kunci matriks k^r ε $M_{8x8}[GF(2^8)]$ dan diberikan dengan :

$$\rho[k] \equiv \sigma[k] \circ \theta \circ \pi \circ \gamma$$

H. Jadwal kunci

Jadwal kunci mengembangkan kunci cipher 512 bit K ε $M_{8x8}[GF(2^8)]$ ke dalam sekuens kunci putaran $K^0, \ldots, K^R$:

$$K^0 = K,$$
$$K^r = \rho[c^r](K^{r-1}),\ r > 0,$$

I. Blok cipher internal W

Blok cipher 512-bit W[K] : $M_{8x8}[GF(2^8)] \rightarrow M_{8x8}[GF(2^8)]$, yang diparameterisasikan dengan 512 bit kunci cipher K, yang didefinisikan sebagai

$$W[K] = \left(\bigcirc_{1}^{r=R} \rho[K^r] \right) \circ \sigma[K^0]$$

Dengan kunci putaran $K^0, \ldots, K^R$ yang diturunkan dari K oleh jadwal kunci. Jumlah putaran default R adalah 10.

J. Padding dan Penguatan MD

Sebelum diarahkan ke operasi hash, sebuah pesan M dari panjang L < 2^{256} dipadding dengan sebuah bit 1, kemudian beberapa bit 0 seperlunya untuk mendapatkan sebuah stirng yang panjangnya adalah sebuah bilangan ganjil dengan kelipatan 256, dan langkah terakhir adalah dengan 256 bit representasi biner dari L, yang menghasilkan pesan yang terpadding m, dipartisi dalam t blok m1, …, mt,. Blok-blok ini dipandang sebagai larik byte dengan secara sekuensial mengelompokkan bit dalam kelompok 8-bit.

K. Fungsi kompresi

Whirlpool mengiterasi skema hash Miyaguchi-Preneel dalam pesan yang terpadding t menggunakan blok cipher 512 bit :

$$\eta_i = \mu(m_i),$$
$$H_0 = \mu(IV),$$
$$H_i = W[H_{i-1}](\eta_i) \oplus H_{i-1} \oplus \eta_i,\ 1 \leqslant i \leqslant t$$

Dimana IV (Initialisation vector) adalah string yang terdiri dari 512 bit 0.

L. Komputasi Message Digest

Message Digest Whirlpool untuk M didefinisikan sebagai keluaran Ht untuk fungsi kompresi, kemudian dipetakan balik ke sebuah bit string :

$$\text{WHIRLPOOL}(M) \equiv \mu^{-1}(H_t)$$

III DESAIN

A. Mode Hashing

Kenapa tidak disebut Miyaguchi-Preneel bukannya, katakanlah, Matyas-Meyer-Oseas (MMO) akan dijelaskan pada bagian dibawah ini. Perlu diperhatikan bahwa jadwal menyerupai kunci enkripsi dari kunci cipher bawah pseudo-keydidefinisikan oleh konstanta bulat, sehingga inti dari proses hashing dapat secara formal dipandang sebagai dua baris berinteraksi enkripsi. Perlu dipertimbangkan Kita bisa menulis tombol putaran terakhir sebagai $K^R = W^1$ [c] (H_{i-1}). Hasilnya kemudian diexored ke status cipher enkripsi langkah terakhir. Sekarang ambil rekursi MMO: Hi = W [H_{i-1}] (i) i. Secara formal menerapkan konstruksi ini dengan enkripsi baris enkripsi kunci akan kita dapat $K'^R = W'$ [c] (H_{i-1}) Hi-1. Dengan menggunakan nilai ini sebagai kunci putaran terakhir efektif menciptakan dua MMO baris yang berinteraksi secara formal(dibandingkan dengan garis enkripsi yang berinteraksi), dan hasil dalam Miyaguchi-Preneel skema, yang karenanya muncul sebagai pilihan alami untuk fungsi kompresi.

B. Pilihan dari kotak substitusi

Form yag sejak awal dikirim untuk whirlpool digunakan sebuah S-box yang pseudo-random, yang dipilih untuk memenuhi kondisi beikut :

- Parameter delta tidak boleh melebihi 8 x 2^{-8}
- Paramter lamda tidak boleh melebihi 16 x 2^{-6}
- Order non lienar v harus maksimum, misalnya 7

Batas pada delta dan lambda berkorespondensi dua kali pada nilai yang bisa diakses minimum untuk jumlah ini. Sebagai kondisi tambahan, S-box tidak memiliki poin yang tetap, dan diimpos dalam sebuah upaya untuk mempercepat pencarian. Kondisi ini diinspirasi oleh studi empiris, dimana korelasi yang kuat ditemukan diantara properti kriptografik dan jumlah poin yang tetap dari sebuah kotak substitusi menyarankan meminimalkan jumlah dari poin ini. Representasi rasional dan polinom

Akan tetapi, dengan struktur yang sangat lemah pada S-box ini menghalangi implementasi hardware yang efisien. Terlebih lagi, sebuah kecacatan bisa tidak sadar pada pencarian program yang acak yang menyebabkan nilai dari S-box yang asli mungkin tidak benar, seperti yang dituliskan 15 x 2^{-16} padahal ukuran yang sebenarnya adalah 16 x 2^{-6} karena adanya bias yang negatif, karena itu, saya sekarang akan mendeskripsikan S-box alternatif yang memenuhi kondisi desain, yang memungkinkan untuk implementasi yang lebih efisien dalam hardware, dimana tidak mempengaruhi implementasi software direpresentasikan dalam cara yang lebih masuk akal.

S-box yang baru mempunyai asal dalam konstruksi tiga layer yang terdiri dari dua layer non linear (yang tiap layer memiliki 2 4x4 S-box) yang dipisahkan dengan seubah transformasi linear M : $GF(2^4)^2$ → $GF(2^4)^2$. Bentuk yang paling umum dari transformasi dapat diasumsikan yang diberikan oleh matriks berikut

$$M = \begin{bmatrix} a+1 & a \\ a & a+1 \end{bmatrix},\ a \in \text{GF}(2^4)$$

Yang mereduksi struktur dengan mengatur R(u) Ξ a x u (R yang aktual digenerate secara random). Jadi, menulis S sebagai sebuah pemetaan S : $GF(2^4)^2$ → $GF(2^4)2$, S(u,v) = (u',v'), sehingga bisa diturunkan

$$u' = E(E(u) \oplus r),\ v' = E^{-1}(E^{-1}(v) \oplus r)$$

Dimana r adalah komplemen R(E(u) xor E^{-1}(v)).

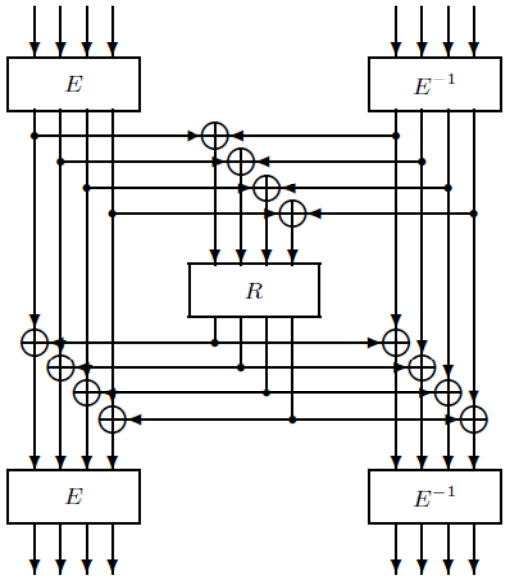
Tabel E (dengan inversi nya yaitu E-1) tidak digenerate secara random; tetapi diturunkan dari sebuah pemetaan eksponensial dengan delta, lamda, dan v masing-masing :

$$E : \text{GF}(2^4) \rightarrow \text{GF}(2^4) : E(u) = \begin{cases} \text{B}_x^u \text{ if } u \neq \text{F}_x, \\ 0_x \text{ otherwise,} \end{cases}$$

Dimana kemunculan dari u = $u_3x^3 + u_2x^2 + u_1x+u_0$ sebagai eksponen dalam B adalah diambil dari nilai numeriknya. Basis Bx dipilih sehingga E tidak memiliki baik poin fixed ataupun poin u sehjingga E(E(u)) = u.

Perlu diperhatikan bahwa E-1 memenuhi properti yang sama.

Table R adalah permutasi yang digenerasi secara pseudo random dengan delta, lamda, dan v yang optimal sehingga S-box yang dibangun dari E,E-1 dan R memenuhi kriteria desain.



Gambar 3 : Struktur dari S-Box whirlpool.

E berkorespondensi pemetaan E : GF(24) → GF(24) : E(u) = B dimana u tidak sama denga Fx, dan E(Fx) = 0. R dihasilkan secara pseudo-random pada langkah yang dapat diverifikasikan. Keduanya memiliki nilai optimal untuk delta, lamda, dan v.

Pencarian acak yang dilakukan dapat mencari S-box dengan lambda = 14x 2^{-6}, sedikit lebih baik dibandingkan dengan batas desain. Sebuah deskripsi dari algoritma pencarina dan sebuah daftar dari hasil S-box.

Tabel 1 mini-box E

u	0_x	1_x	2_x	3_x	4_x	5_x	6_x	7_x	8_x	9_x	A_x	B_x	C_x	D_x	E_x	F_x
$E[u]$	1_x	B_x	9_x	C_x	D_x	6_x	F_x	3_x	E_x	8_x	7_x	4_x	A_x	2_x	5_x	0_x

Tabel 2 mini-box R

u	0_x	1_x	2_x	3_x	4_x	5_x	6_x	7_x	8_x	9_x	A_x	B_x	C_x	D_x	E_x	F_x
$R[u]$	7_x	C_x	B_x	D_x	E_x	4_x	9_x	F_x	6_x	3_x	8_x	A_x	2_x	5_x	1_x	0_x

C. Pilihan dari layer difusi

Matriks sirkular C mempunyai banyak elemen 1 (kira-kira 3 per baris) untuk sebuah matriks MDS 8x8; terlebih lagi, setiap elemen mempunyai berat Hamming paling banyak 2, dan derajat polinom paling banyak 3. Batasan-batasan ini secara khusus menguntungkan untuk smart card dan perangkat keras, dan dari semua matriks yang memenuhi kriteria ini merupakan implementasi yang efisien dari platform tersebut.

D. Putaran terakhir

Perbedaan diantara struktur whirlpool dan struktur square dan rijndael adalah fakta bahwa pada whirlpool, operasi θ ada di setiap putaran, sedangkan rijndael tidak ada pada putaran pertama dan terakhir. Pertama, akan dijelaskan kenapa satu aplikasi dari operasi θ dapat ditinggalkan tanpa mengubah level keamanan dari cipher. Kemudian, akan didaftarkan beberapa alas an untuk meninggalkan satu aplikasi dari operasi θ, diikuti dengan alas an kenapa tetap dipertahankan di whirlpool.

Misalnya terdapat cipher kotak dengan dua putaran, dan sebuah tambahan kunci :

$$E = \sigma[K^2] \circ \tau \circ \gamma \circ \theta \circ \sigma[K^1] \circ \tau \circ \gamma \circ \theta \circ \sigma[K^0]$$

Operasi θ dan σ[K] dapat ditukarkan, karena K diganti dengan kunci ekivalen K' = θ(K). Karena itu, persamaan diatas dapat dituliskan sebagai :

$$E = \sigma[K^2] \circ \tau \circ \gamma \circ \theta \circ \sigma[K^1] \circ \tau \circ \gamma \circ \sigma[\theta(K^0)] \circ \theta$$

Pada persamaan di atas, jelas bahwa θ tidak berkontribusi pada keamanan dari cipher, karena bisa dilakukan oleh penyerang, tanpa mengetahui kuncinya. Karena itu, definisi dari persamaan yang pertama bisa diabaikan. Definisi baru menjadi :

$$E' = \sigma[K^2] \circ \tau \circ \gamma \circ \theta \circ \sigma[K^1] \circ \tau \circ \gamma \circ \sigma[K^0]$$

Perlu diperhatikan bahwa di analisa ini, tidak dibuat asumsi apapun tentang penyerangan yang akan dilakukan oleh penyerang. Dari sana dibuktikan bahwa keamanan dari E dan E' adalah ekivalen.

Alasan lain adalah tidak ada hubungannya dengan keamanan. Terlebih lagi, implementasi pada prosesor kecil yang mengeksekusi semua transformasi secara eksplisit akan mengalami performa yang meningkat. Yang ketiga, keuntungan bahwa dekripsi dan enkripsi sama satu sama lain.

Alasan paling kuat untuk menjaga semua putaran identik satu sama lain, adalah performa dari prosesor dengan sebuah *cache memory* yang agak cepat. Jika tidak semua ronde identik, maka jumlah tabel untuk disimpan dalam memori bertambah. Untuk implementasi yang cepat dari square, tabel untuk putaran yang lengkap dapat disimpan dalam cache, tetapi tidak ada tempat yang tersisa untuk tabel dengan putaran yang tidak lengkap. Hasil dari putaran tanpa θ membutuhkan waktu yang lebih lama untuk dieksekusi.

IV TARGET KEAMANAN

Diasumsikan diambil sebuah hasil hash adalah nilai dari substring n-bit dari keluaran penuh whirlpool, maka :

- Beban yang diharapkan dari mengenerasi sebuah collision adalah $2^{n/2}$ eksekusi dari whirlpool.

- Diketahui nilai dari sebuah n-bit, beban kerja yang diharapkan dari menemukan sebuah pesan yang meng-hash dari nilai tersebut adalah perpangkatan dari 2^n eksekusi dari whirlpool.

Terlebih lagi, sangat memungkinkan untuk mendeteksi hubungan diantara kombinasi linear dair bit masukan dan kombinasi dari bit hasil hash. Juga sangat memungkinkan untuk memprediksi bit dari hasil hash yang akan berubah ketika masukan bit tertentu diganti, sehingga whirlpool aman dari serangan yang berbeda-beda.

Pernyataan ini merupakan hasil dari margin keamanan yang diambil dengan menimbang semua serangan yang diketahui. Tetapi penulis juga menyadari bahwa tidak mungkin untuk membuat pernyataan yang tidak spekulatif pada hal yang masih belum diketahui.

V. IMPLEMENTASI

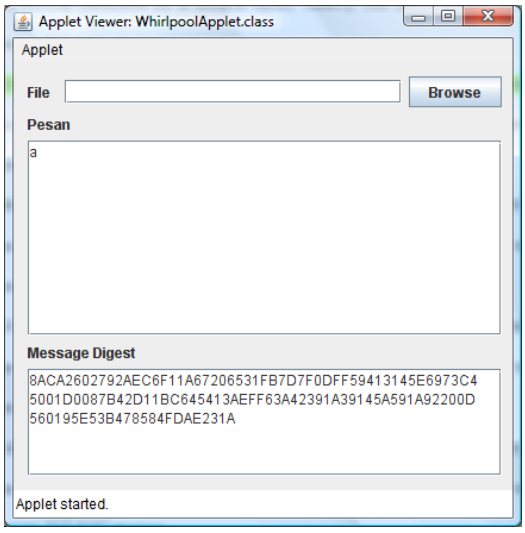
Whirlpool dapat diimplementasikan secara efisien. Pada platform berbeda, dengan optimisasi yang berbeda dan tradeoff yang berbeda.

A. Implementasi pada Java Applet

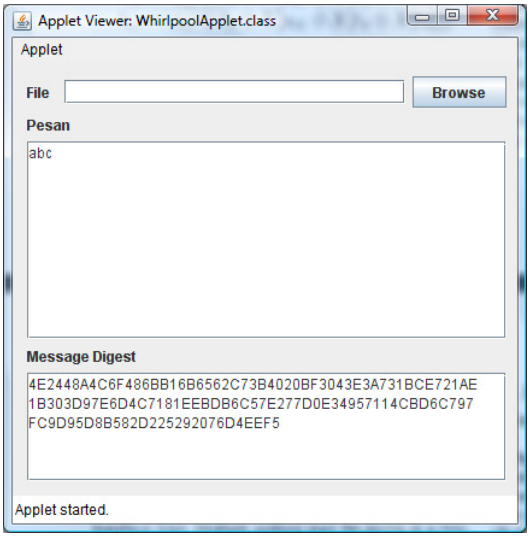
Saya mengimplementasikan fungsi hash Whirlpool ini dalam Java Applet. Di bawah ini merupakan source code nya. Tetapi hanya saya cantumkan potongan source code yang penting saja untuk memberikan gambaran jalannya program.

```
public static final int DIGESTBITS = 512;
public static final int DIGESTBYTES = DIGESTBITS
                                    >>> 3;
protected static final int R = 10;

private static final String sbox =
  "\u1823\uc6E8\u87B8\u014F\u36A6\ud2F5\u796F\u9
152" +

"\u60Bc\u9B8E\uA30c\u7B35\u1dE0\ud7c2\u2E4B\uFE5
7" +

"\u1577\u37E5\u9FF0\u4AdA\u58c9\u290A\uB1A0\u6B8
5" +

"\uBd5d\u10F4\ucB3E\u0567\uE427\u418B\uA77d\u95d
8" +

"\uFBEE\u7c66\udd17\u479E\ucA2d\uBF07\uAd5A\u833
3" +

"\u6302\uAA71\uc819\u49d9\uF2E3\u5B88\u9A26\u32B
0" +

"\uE90F\ud580\uBEcd\u3448\uFF7A\u905F\u2068\u1AA
E" +

"\uB454\u9322\u64F1\u7312\u4008\uc3Ec\udBA1\u8d3
d" +

"\u9700\ucF2B\u7682\ud61B\uB5AF\u6A50\u45F3\u30E
F" +

"\u3F55\uA2EA\u65BA\u2Fc0\udE1c\uFd4d\u9275\u068
A" +

"\uB2E6\u0E1F\u62d4\uA896\uF9c5\u2559\u8472\u394
c" +

"\u5E78\u388c\ud1A5\uE261\uB321\u9c1E\u43c7\uFc0
4" +

"\u5199\u6d0d\uFAdF\u7E24\u3BAB\ucE11\u8F4E\uB7E
B" +

"\u3c81\u94F7\uB913\u2cd3\uE76E\uc403\u5644\u7FA
9" +

"\u2ABB\uc153\udc0B\u9d6c\u3174\uF646\uAc89\u14E
1" +

"\u163A\u6909\u70B6\ud0Ed\ucc42\u98A4\u285c\uF88
6";

static {
      for (int x = 0; x < 256; x++) {
          char c = sbox.charAt(x/2);
          long v1 = ((x & 1) == 0) ? c >>> 8 : c
                                     & 0xff;
          long v2 = v1 << 1;
          if (v2 >= 0x100L) {
              v2 ^= 0x11dL;
          }
          long v4 = v2 << 1;
          if (v4 >= 0x100L) {
              v4 ^= 0x11dL;
          }
          long v5 = v4 ^ v1;
          long v8 = v4 << 1;
          if (v8 >= 0x100L) {
              v8 ^= 0x11dL;
          }
          long v9 = v8 ^ v1;
          C[0][x] =
              (v1 << 56) | (v1 << 48) | (v4 <<
             40) | (v1 << 32) | (v8 << 24) | (v5
             << 16) | (v2 <<  8) | (v9);

          for (int t = 1; t < 8; t++) {
              C[t][x] = (C[t - 1][x] >>> 8) |
             ((C[t - 1][x] << 56));
          }
     }
protected void processBuffer() {
      for (int i = 0, j = 0; i < 8; i++, j += 8)
     {
          block[i] =(((long)buffer[j]) << 56) ^
              (((long)buffer[j + 1] & 0xffL) <<
             48) ^
              (((long)buffer[j + 2] & 0xffL) <<
             40) ^
              (((long)buffer[j + 3] & 0xffL) <<
             32) ^
              (((long)buffer[j + 4] & 0xffL) <<
             24) ^
              (((long)buffer[j + 5] & 0xffL) <<
             16) ^
              (((long)buffer[j + 6] & 0xffL) <<
             8) ^
              (((long)buffer[j + 7] & 0xffL)
              );
      }
      for (int i = 0; i < 8; i++) {
          state[i] = block[i] ^ (K[i] =
             hash[i]);
      }
      for (int r = 1; r <= R; r++) {
          for (int i = 0; i < 8; i++) {
              L[i] = 0L;
```

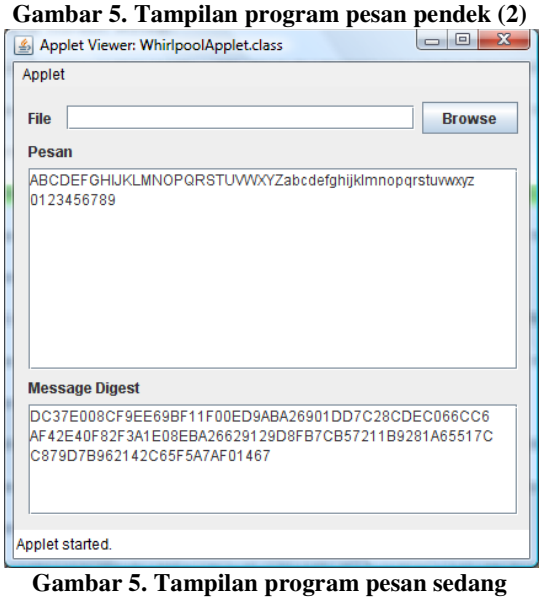
```
                for (int t = 0, s = 56; t < 8;
t++, s -= 8) {
                    L[i] ^= C[t][(int)(K[(i - t) &
                    7] >>> s) & 0xff];
                }
            }
            System.arraycopy(L, 0, K, 0, 8);
            K[0] ^= rc[r];
            for (int i = 0; i < 8; i++) {
                L[i] = K[i];
                for (int t = 0, s = 56; t < 8;
                    t++, s -= 8) {
                    L[i] ^= C[t][(int)(state[(i -
                    t) & 7] >>> s) & 0xff];
                }
            }
            System.arraycopy(L, 0, state, 0, 8);
        }
        for (int i = 0; i < 8; i++) {
            hash[i] ^= state[i] ^ block[i];
        }
    }

    public static void makeNESSIETestVectors() {
        Whirlpool w = new Whirlpool();
        byte[] digest = new byte[64];
        byte[] data = new byte[128];
        Arrays.fill(data, (byte)0);
        for (int i = 0; i < 1024; i++) {
            w.NESSIEinit();
            w.NESSIEadd(data, i);
            w.NESSIEfinalize(digest);
            String s = Integer.toString(i);
            s = "     ".substring(s.length()) + s;
        }
        data = new byte[512/8];
        Arrays.fill(data, (byte)0);
        for (int i = 0; i < 512; i++) {
            // set bit i:
            data[i/8] |= 0x80 >>> (i % 8);
            w.NESSIEinit();
            w.NESSIEadd(data, 512);
            w.NESSIEfinalize(digest);
            // reset bit i:
            data[i/8] = 0;
        }
        for (int i = 0; i < digest.length; i++) {
            digest[i] = 0;
        }
        for (int i = 0; i < LONG_ITERATION; i++) {
            w.NESSIEinit();
            w.NESSIEadd(digest, 512);
            w.NESSIEfinalize(digest);
        }
    }
```



Gambar 4. Tampilan program pesan pendek (1)



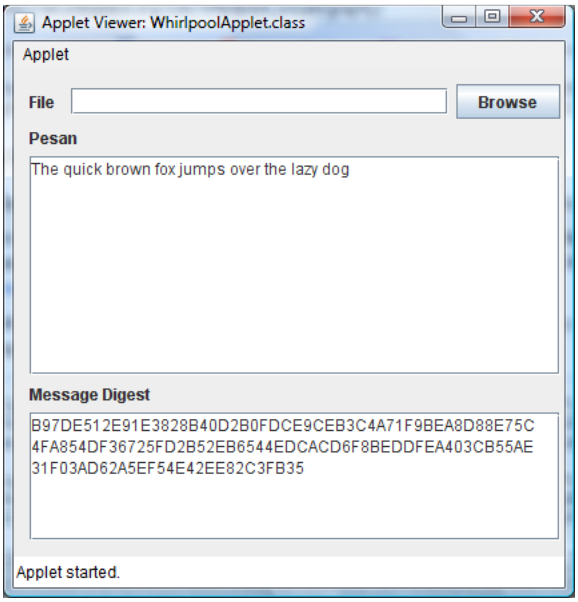
Gambar 5. Tampilan program pesan pendek (2)



Gambar 5. Tampilan program pesan sedang

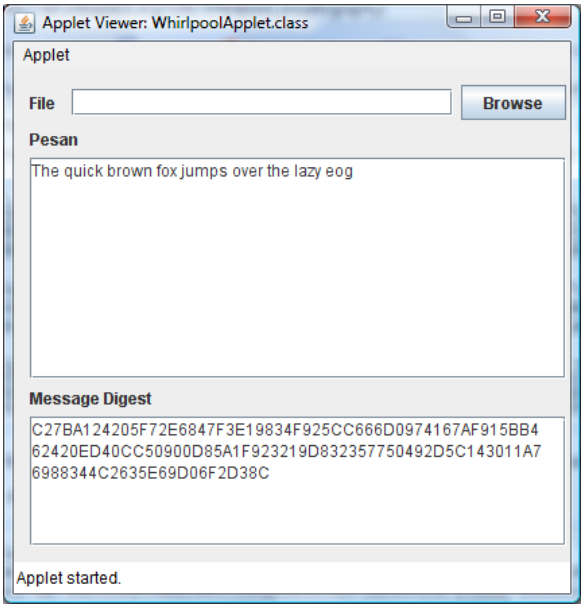
Untuk membuktikan kesensifitasan fungsi hash whirlpool,

saya mencoba menguji di program ini :



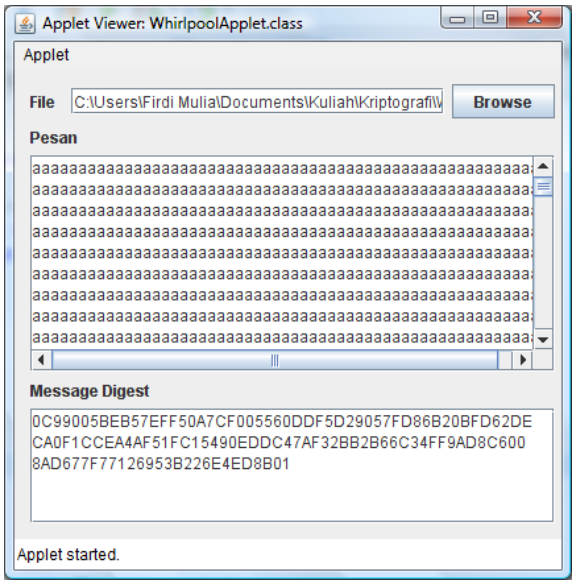
Gambar 6. Tampilan program pesan awal

Setelah itu, pesan tersebut diubah huruf d dari kata "dog" menjadi huruf e. Dan hasilnya adalah :



Gambar 7. Tampilan program setelahnya

Terlihat ternyata fungsi hash Whirlpool memang sensitif dengan perubahan sedikit pesan saja.



Gambar 8. Tampilan program pesan panjang (huruf a sebanyak 10^6)

B. Prosesor 64-bit

Pada prosesor ini, penulis menyarankan sebuah tabel lookup untuk mengimplementasikan ρ. Misalnya Ck adalah baris ke k dari matriks circular C; menggunakan 8 tabel Tk[x] kongruen S[x] . Ck, $0 \le k \le 7$, yaitu

$$
\begin{aligned}
T_0[x] &= S[x] \cdot [01_x\ 01_x\ 04_x\ 01_x\ 08_x\ 05_x\ 02_x\ 09_x] \\
T_1[x] &= S[x] \cdot [09_x\ 01_x\ 01_x\ 04_x\ 01_x\ 08_x\ 05_x\ 02_x] \\
T_2[x] &= S[x] \cdot [02_x\ 09_x\ 01_x\ 01_x\ 04_x\ 01_x\ 08_x\ 05_x] \\
T_3[x] &= S[x] \cdot [05_x\ 02_x\ 09_x\ 01_x\ 01_x\ 04_x\ 01_x\ 08_x] \\
T_4[x] &= S[x] \cdot [08_x\ 05_x\ 02_x\ 09_x\ 01_x\ 01_x\ 04_x\ 01_x] \\
T_5[x] &= S[x] \cdot [01_x\ 08_x\ 05_x\ 02_x\ 09_x\ 01_x\ 01_x\ 04_x] \\
T_6[x] &= S[x] \cdot [04_x\ 01_x\ 08_x\ 05_x\ 02_x\ 09_x\ 01_x\ 01_x] \\
T_7[x] &= S[x] \cdot [01_x\ 04_x\ 01_x\ 08_x\ 05_x\ 02_x\ 09_x\ 01_x]
\end{aligned}
$$

Kemudian sebuah baris bi dari b = (θ ∘ π ∘ γ)(a) dapat dikalkulasikan dengan delapan tabel lookup dan tujuh operasi xor seperti :

$$b_i = \bigoplus_{k=0}^{7} T_k[a_{(i-k) \bmod 8,k}]$$

Penambahan kunci kemudian melengkapi evaluasi dari ρ dengan sebuah penambahan xor. Tabel T membutuhkan 2^8 x 8 byte untuk setiap storage. Sebuah implementasi dapat menggunakan fakta bahwa masukan yang berkorespondensi dari tabel T yang berbeda adalah permutasi siklik dari satu sama lain dan menyimpan beberapa memori untuk mengenalkan permutasi ekstra pada waktu runtime. Biasanya ini menurunkan performansi dari implementasi.

C. Prosesor 32 bit

Matriks sirkular C dari suku 2m menunjukkan struktur berikut :

$$C = \begin{bmatrix} U & V \\ V & U \end{bmatrix}$$

Dimana U dan V adalah matriks dengan suku m. Sebuah implementasi 32 bit bisa mendapatkan keuntungan dari struktur ini dengan melambangkan elemen c ε GF$(2^8)^8$ sebagai pasangan c = [c0 c1] dari elemen ci ε GF$(2^8)^4$:

$$b = \theta(a) \Leftrightarrow \begin{cases} \hat{b}_0 = \hat{a}_0 U \oplus \hat{a}_1 V \\ \hat{b}_1 = \hat{a}_0 V \oplus \hat{a}_1 U \end{cases}$$

Yang akan melipatgandakan dengan kompleksitas yang diturunkan untuk prosesor 64 bit tergantung pada jumlah tabel lookup dan xor, tetapi menggunakan tabel yang lebih kecil (setiap tabel menempati 2^8 x 4 byte).

D. Prosesor 8-bit

Pada prosesor 8-bit dengan jumlah RAM yang terbatas, misalnya prosesor smart card, pendekatan yang sebelumnya tidak memungkinkan. Pada prosesor ini substitusi dilakukan byte per byte, dikombinasikan dengan transformasi σ[k]. Untuk θ, perlu untuk mengimplementasikan perkalian matriks. Pseudo-code dibawah menghitung satu baris dari b = θ(a), menggunakan sebuah tabel X yang mengimplementasikan perkalian dengan polinom g(x) = x dalam GF(2^8) dan lima variabel t0 sampai t4, dengan cost 46 xor dan 24 tabel lookup.

$$\begin{aligned}
&t_0 \leftarrow a_{i1} \oplus a_{i3} \oplus a_{i5} \oplus a_{i7};\\
&t_1 \leftarrow a_{i3} \oplus a_{i6};\\
&t_2 \leftarrow a_{i5} \oplus a_{i0};\\
&t_3 \leftarrow a_{i7} \oplus a_{i2};\\
&t_4 \leftarrow a_{i1} \oplus a_{i4};\\
&b_{i0} \leftarrow a_{i0} \oplus t_0 \oplus X[a_{i2} \oplus X[t_1 \oplus X[t_4]]];\\
&b_{i2} \leftarrow a_{i2} \oplus t_0 \oplus X[a_{i4} \oplus X[t_2 \oplus X[t_1]]];\\
&b_{i4} \leftarrow a_{i4} \oplus t_0 \oplus X[a_{i6} \oplus X[t_3 \oplus X[t_2]]];\\
&b_{i6} \leftarrow a_{i6} \oplus t_0 \oplus X[a_{i0} \oplus X[t_4 \oplus X[t_3]]];\\
&t_0 \leftarrow a_{i0} \oplus a_{i2} \oplus a_{i4} \oplus a_{i6};\\
&t_1 \leftarrow a_{i4} \oplus a_{i7};\\
&t_2 \leftarrow a_{i6} \oplus a_{i1};\\
&t_3 \leftarrow a_{i0} \oplus a_{i3};\\
&t_4 \leftarrow a_{i2} \oplus a_{i5};\\
&b_{i1} \leftarrow a_{i1} \oplus t_0 \oplus X[a_{i3} \oplus X[t_1 \oplus X[t_4]]];\\
&b_{i3} \leftarrow a_{i3} \oplus t_0 \oplus X[a_{i5} \oplus X[t_2 \oplus X[t_1]]];\\
&b_{i5} \leftarrow a_{i5} \oplus t_0 \oplus X[a_{i7} \oplus X[t_3 \oplus X[t_2]]];\\
&b_{i7} \leftarrow a_{i7} \oplus t_0 \oplus X[a_{i1} \oplus X[t_4 \oplus X[t_3]]];
\end{aligned}$$

VI. ANALISIS

A. Perkiraan efisiensi

Dengan menggunakan implementasi Java pada 2.5 GHz Intel Core 2 Duo, saya mengamati kalau whirlpool beroperasi sekitar 219 putaran per byte yang terhash. Fungsi kompresi berjalan sekitar 168 per byte hash. Hasil tersebut tidaklah sebagus yang diperkirakan.

Banyak factor yang mempengaruhi performansi. Pertama, sebuah prosesor 32 bit digunakan untuk menguji sebuah implementasi 64 bit; mungkin hasil yang lebih baik akan diperoleh kalau dijalankan di prosesor 64 bit. Kedua, sepertinya kemampuan paralel dari prosesor tidak sepenuhnya digunakan.

B. Teknik untuk mencegah kelemahan implementasi pada perangkat lunak

Fungsi hash ini tidak menggunakan kunci rahasia. Sebagai prinsipnya, fungsi tersebut tidak rentan terhadap teknik pencarian kunci yang dideskripsikan oleh Kocher. Akan tetapi, fungsi hash yang kadang-kadang digunakan sebagai pembangunan blok untuk primitif kriptografik, seperti MAC, yang menggunakan kunci rahasia. Pada kasus itu, perhatian perlu diberikan ke implementasi dari putaran transformasi seperti penjadwalan kunci dari primitif.

Contoh pertama adalah timing attack yang dapat diaplikasikan jika waktu pengeksekusian dari primitif tergantung pada nila ikunci dan plainteks. Ini biasanya disebabkan oleh kehadiran jalur eksekusi kondisional. Sebagai contoh, perkalian dengan sebuah nilai konstan pada bidang yang terbatas kadang-kadang diimplementasikan sebagai sebuah perkalian yang diikuti oleh sebuah reduksi, yang kemudian reduksi diimplementasikan sebagai xor. Kerentanan ini dicegah dengan mengimplementasikan perkalian dengan sebuah konstan dengan tabel lookup.

Kelas kedua dari serangan adalah serangan yang berdasarkan pada pengamatan yang hati-hati terhadap kekuatan pola konsumsi dari peralatan yang terenkripsi. Perlindungan terhadap tipe serangan ini hanya bisa diraih dengan ukuran kombinasi dari tingkatan hardware dan software.

C. Kriptanalisis Terhadap Fungsi Whirlpool

Aplikasi dari teknik kriptanalisis diferensial ke fungsi hash berdasarkan blok cipher. Walaupun ada beberapa perbedaan penting diantara serangan pada blok cipher dan serangan pada fungsi hash, pada dasarnya adalah teknik yang sama. Kedua serangan memerlukan sebuah karakteristik diferensial ditemukan yang mempunyai peluang besar.

Jumlah cabang dari transformasi θ adalah B = 9. Karena teorema propagasi pola square, untuk semua nilai masukan yang berbeda dari W, terdapat jumlah S-box dengan nilai masukan yang berbeda pada putaran berurutan setidaknya B^2 = 81. Sebagai konsekuensinya, tidak ada karakteristik diferensial yang ditemukan setelah

4 putaran dari W yang memiliki peluang lebih besar dibandingkan 2^{-405}. Ini menyebabkan serangan diferensial klasik sangat kecil kemungkinannya untuk berhasil dari fungsi hash.

VII. KESIMPULAN

1. Whirlpool lebih efisien dibanding dengan kebanyakan fungsi hash modern karena mendukung eksekusi paralel.
2. Whirlpool tidak membutuhkan ruang penyimpanan yang besar (baik untuk kode maupun untuk tabel).
3. Bisa diimplementasikan pada lingkungan yang memiliki banyak batasan seperti *smart card*.
4. Whirlpool tidak memerlukan instruksi khusus yang harus diimplementasikan dalam prosesor terlebih dahulu.
5. Mempunyai *message digest* yang panjang sehingga meningkatkan proteksi terhadap serangan.
6. Fungsi hash ini belum dideteksi adanya *collision*.

REFERENSI

Cryptographic Hash Algorithm. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html. Tanggal akses : 14 Mei 2010.

Whirlpool (Cryptography). http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html. Tanggal akses : 14 Mei 2010.

P. Barreto, V. Rijmen, "The Whirlpool Hashing Function", 2003, http://saluc.engr.uconn.edu/refs/algorithms/**hash**alg/barreto00**whirlpool**.pdf. Tanggal akses : 15 Mei 2010

T. Shirai, "*On the diffusion matrix employed in the Whirlpool hashing function*" http://www.cosic.esat.kuleuven.be/nessie/reports/phase2/whirlpool-20030311.pdf. Tanggal akses : 14 Mei 2010.

D. Kotturi, Y. Seong-Moo, *"High-Speed Parallel Architecture of the Whirlpool Hash Function"*, http://www.sersc.org/journals/IJAST/vol7/3.pdf, Tanggal akses : 16 Mei 2010.

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, 16 Mei 2010

Firdi Mulia
13507045