Studi dan Implementasi Algoritma Keccak

Moch. Yusup Soleh / 13507051[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]if17051@students.if.itb.ac.id

***Abstrak*—Makalah yang dibuat membahas tentang studi dan implementasi algoritma keccak, yang merupakan algoritma fungsi hash satu arah yang menjadi salah satu dari lima finalis NIST hash function competition. State pada keccak merupakan array 3 dimensi yang dibagi kedalam dua bagian, yaitu bitrate dan capacity. Keccak berbasis konstruksi Sponge, yang terdiri dari dua fase, yaitu absorbing dan squeezing. Dalam fase absorbing setiap blok masukan akan di-XOR-kan dengan blok bitrate dari state sebelumnya untuk kemudian dilewatkan kedalam fungsi permutasi keccak-f. Keccak sudah diterapkan dalam software maupun hardware dan kesemuanya menghasilkan performansi yang baik.**

***Kata Kunci*—Fungsi Hash, Spon, SHA-3, Keccak-f, State, Absorbing, Squeezing.**

I. PENDAHULUAN

Pada tahun 2005 algoritma hash standar SHA-1 dan SHA-2 terbukti sangat rentang terhadap serangan. Meskipun demikian, dalam prakteknya banyak komputasi yang harus dilakukan dalam melakukan serangan, sehingga ancaman terhadap serangan masih diabaikan. Namun, seiring perkembangan teknologi dan cryptanalysis, kebutuhan akan algoritma fungsi hash baru yang lebih aman semakin tinggi. Untuk merealisasikan hal ini diadakan NIST hash function competition, yang merupakan sebuah kompetisi terbuka yang diadakan oleh NIST (National Institute of Standards and Technology) Amerika Serikat, untuk mendapatkan kandidat algoritma fungsi hash SHA-3 yang akan menggantikan algoritma fungsi SHA pendahulunya yaitu SHA-1 dan SHA-2, pada tahun 2007 sampai 2012.

Dari sejumlah rancangan algoritma fungsi hash yang terdaftar menjadi calon SHA-3 pada kompetisi ini, sampai saat ini terdapat lima finalis kompetisi yang merupakan calon kuat yang akan menjadi pemenang. Salah satu diantaranya adalah algoritma Keccak, yang dirancang oleh Guido Bertoni, Joan Daemen, Michaël Peeters dan Gilles Van Assche. Terpilihnya Keccak sebagai salah satu kandidat dari SHA-3 membuktikan bahwa algoritma ini merupakan algoritma fungsi hash yang kuat dan aman. Oleh karena itu, sangatlah penting saat ini untuk mengetahui cara kerja dari algoritma ini serta implementasinya dalam lingkungan hardware dan software.

II. ALGORITMA FUNGSI HASH

Fungsi Hash adalah fungsi yang menerima masukan string yang panjangnya sembarang dan mengkonversinya menjadi string keluaran yang panjangnnya tetap (*fixed*) (umumnya berukuran lebih kecil daripada ukuran string semula)[6]. Masukan dari fungsi hash adalah sebuah string dengan panjang tertentu, dan keluarannya berupa *message digest* dengan ukuran tetap. Jika string pesan (*message*) adalah M dan hasil dari fungsi hash F pesan tersebut merupakan sebuah *message digest* d, maka pernyataan tersebut dapat dinyatakan dalam persamaan berikut.

$$d = F(M) \quad (1)$$

Dari persamaan (1) diatas, dapat disimpulkan bahwa fungsi hash mengkompresi sembarang pesan yang mempunyai panjang tertentu (sembarang), menjadi suatu ringkasan pesan (*message digest*) dengan panjang yang tetap. Nama-nama lain dari fungsi hash adalah, fungsi kompresi, cetak-jari, *cryptographic checksum*, *Message Integrity Check* (MIC) dan *Manipulation Detection code* (MDC).

Fungsi hash ini banyak kegunaannya, namun inti dari penggunaan fungsi hash adalah sebagai *signature* dari suatu teks atau data masukan. Kegunaan dari signature ini banyak, diantaranya untuk memverifikasi kesamaan salinan suatu arsip dengan arsip aslinya. Contoh lainnya adalah untuk membuat tanda tangan digital atau *digital signature*.

A. Fungsi Hash Satu Arah

Fungsi hash satu arah adlaah fungsi hash yang bekerja dalam satu arah [6]. Pesan yang diubah menjadi *message digest* tidak dapat diubah kembali kedalam pesan semula.

Sifat-sifat dari fungsi hash satu arah adalah[6]:

1. Fungsi hash (*F*) dapat diterapkan pada blok data ukuran apa saja.
2. *F* menghasilkan nilai (*d*) dengan panjang tetap

(*fixed-length output*).

3. *F(x)* mudah dihitung untuk setiap nilai *x* yang diberikan.
4. Untuk setiap *d* yang dihasilkan, tidak mungkin dikembalikan nilai *x* sedemikian sehingga *F(x) = d*. itulah sebabnya fungsi *F* dikatakan fungsi hash satu arah (*one-way hash function*).
5. Untuk setiap *x* yang diberikan, tidak mungkin mencari $y \neq x$ sedemikian sehingga *F(y) = F(x)*.
6. Tidak mungkin mencari pasangan *x* dan *y* sedemikian sehingga *F(x) = F(y)*.

III. STUDI ALGORITMA KECCAK

Keccak merupakan algoritma fungsi hash satu arah yang berbasis pada konstruksi spon dengan menggunakan fungsi permutasi *keccak-f* dengan rentang (panjang) permutasi *b*, ukuran setiap *lane* dimana:

$$b = 25 \times 2^l \quad \ldots\ldots(2)$$

Dengan

$$0 \leq l \leq 6 \quad \ldots\ldots(3)$$

Algoritma keccak memiliki prinsip yang sama dengan algoritma *cipher block*, dimana proses dilakukan terhadap blok-blok, setiap hasil proses bergantung dari masukan dan hasil proses sebelumnya, serta setap proses dikenakan pada sebuah fungsi utama yang terdiri dari sejumlah *round* fungsi yang diiterasi beberapa kali. Namun terdapat perbedaan antara algoritma hash satu arah Keccak, dengan algoritma cipher block, yaitu:

1. Keccak tidak memiliki *key-schedule*.
2. Menggunakan konstanta *round* yang bersifat tetap dari pada *round key*.

Keccak menggunakan *inner state* selama proses *hashing* berlangsung. Dan fungsi spon yang digunakan terdiri dari *padding, absorbing,* dan *squeezing*. Setiap state memiliki panjang sesuai dengan panjang permutasi, yaitu *b*.

Algoritma keccak menerima tiga parameter masukan, yaitu *bitrate* (*r*)*, capacity* (*c*)*,* dan *difersity* (*d*). Secara umum proses dari keccak ini adalah:

1. Preproses pesan masukan (P), yaitu menerapkan *padding* pada pesan masukan. Panjang pesan masukan hasil *padding* harus merupakan kelipatan *r*, dengan *r* = bitrate.
2. Pemecahan pesan masukan menjadi P_0, P_1, P_2,…P_i, dimana i = jumlah kelipatan panjang *bitrate* untuk panjang pesan masukan.
3. *Absorbing* pada semua pecahan pesan masukan.
4. *Squeezing* sebanyak j, dimana j = kelipatan panjang keluaran *r/w* untuk memenuhi panjang output yang diinginkan, r = *bitrate* dan *w* = panjang *lane* dari state. Dengan:

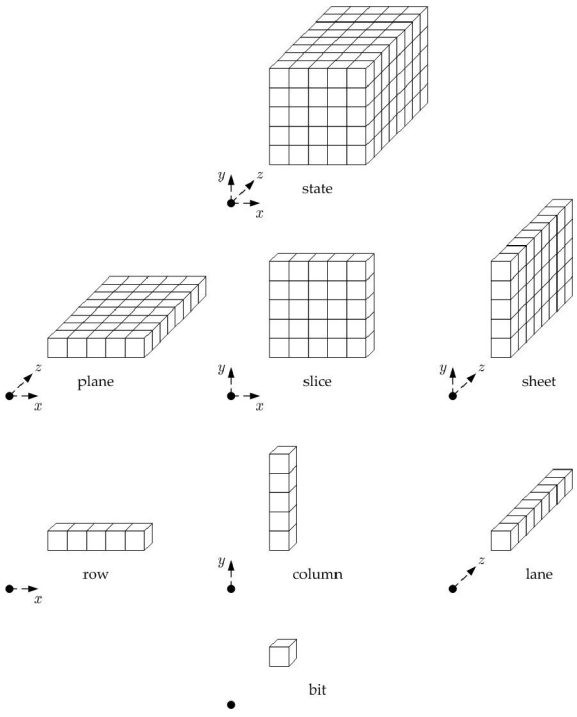
$$w = 2^l \quad \ldots\ldots(4)$$

5. Keluaran merupakan konkatenasi dari keluaran *Squeezing* pada rentang bitrate tertentu.

A. State

State pada keccak merupakan serangkaian bit yang dipandang sebagai suatu array tiga dimensi dari bit-bit tersebut. Setiap sumbu pada array tersebut direpresentasikan oleh *x, y,* dan *z*. *x* x *y* merupakan slice dari state, dan *z* merupakan sumbu *lane* state.

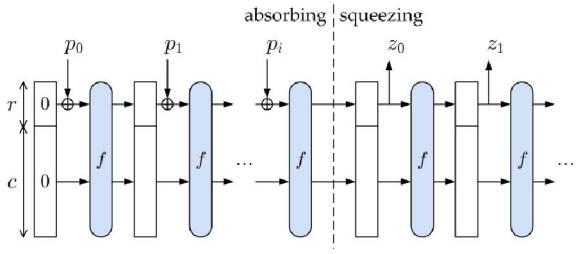
Proses yang dilakukan pada state keccak berbasis pada tiap slice state tersebut. Jumlah bit untuk tiap slice pada state adalah tetap yaitu 5 x 5 atau 25-bit. Sedangkan ukuran tiap *lane* untuk state adalah 1, 2, 4, 8, 16, 32 atau 64. Gambar 1 menjelaskan mengenai state dan elemen-elemennya dengan lebih detil.



Gambar 1. Elemen-elemen State [11]

B. Spon

Fungsi Spon pada keccak berbasis pada konstruksi spon. Konstruksi spon merupakan konstruksi iterasi sederhana untuk membangun sebuah fungsi spon dengan variabel panjang input dan panjang output yang berubah-ubah bergantung pada panjang transformasi (atau permutasi) tetap *f* yang beroperasi dalam sejumlah tetap *b* dalam bit.



Gambar 2. Konstruksi Spon [9]

Dari Gambar 2, secara umum, dalam konstruksi spon terdapat dua fase, yaitu fase *absorbing* dan fase *squeezing*.

1. Fase *absorbing*, merupakan fase dimana proses dilakukan terhadap semua pecahan dari input masukan (P_0, P_1, P_2,…P_i) di-xor-kan dengan bagian bitrate dari state kemudian dilewatkan kedalam fungsi *f*. *Absorbing* dilakukan secara iteratif sesuai dengan jumlah pecahan yang didapat.
2. Fase *squeezing*, merupakan fase untuk mendapatkan hasil keluaran. Dalam fase ini dilakukan konkatenasi terhadap sejumlah bit tertentu dari hasil fungsi *f* sedemikian sehingga jumlah bit konkatenasi tersebut sama dengan jumlah bit konkatenasi yang diinginkan.

Fungsi spon keccak merupakan penerapan dari konstruksi spon dengan terlebih dahulu melakukan proses inisialisasi. Secara umum proses inisialisasi dibagi menjadi dua tahap, yaitu:

1. Men-set setiap bit pada state dengan nol untuk inisial state.
2. Menerapkan *padding* pada pesan masukan sedemikian sehinga panjang pesan masukan merupakan kelipatan dari panjang bitrate awal yang ditentukan. Proses ini dilakukan dengan menambahkan 1 dan sejumlah 0 sesedikit mungkin sampai panjang pesan memenuhi kelipatan panjang bitrate state yang ditentukan. Persamaan (5) dan (6) diterapkan dalam proses ini [7].

$$P = pad(M, 8) \,||\, enc(d, 8) \,||\, enc(r/8, 8) \quad \text{.....(5)}$$

$$P = pad(P, r) \quad \text{.....(6)}$$

   - *pad(M, n)*, fungsi dimana pesan *M* ditambah 1 kemudian ditambah 0 sedemikian rupa sehingga jumlah *M* merupakan kelipatan terkecil dari *n*.
   - *enc(x, n)*, fungsi yang menghasilkan string dengan panjang *n*-bit yang diambil dari Least Significant Bit (LSB) ke Most Significant Bit (MSB), pada *x*.

Gambar 3 menjelaskan algoritma dari fungsi spon yang digunakan pada keccak.

```
KECCAK[r,c,d](M) {
  Initialization and padding
  S[x,y] = 0,                                  forall (x,y) in (0..4,0..4)
  P = M || 0x01 || byte(d) || byte(r/8) || 0x01 || 0x00 || .. || 0x00

  Absorbing phase
  forall block Pi in P
    S[x,y] = S[x,y] xor Pi[x+5*y],             forall (x,y) such that x+5*y < r/w
    S = KECCAK-f[r+c](S)

  Squeezing phase
  Z = empty string
  while output is requested
    Z = Z || S[x,y],                           forall (x,y) such that x+5*y < r/w
    S = KECCAK-f[r+c](S)

  return Z
}
```

Gambar 3. *Psuedo code* algoritma fungsi spon [7]

C. Fungsi Permutasi Keccak-f

Persamaan umum untuk fungsi keccak-f adalah *keccak-f[b]* atau *keccak-f[r+c]*, dimana b sesuai dengan persamaan (2). Karena nilai *l* mempunyai rentang antara 0 sampai 6, maka nilai *b* yang mungkin adalah 25. 50, 100, 200, 400, 800 dan 1600.

Fungsi permutasi *keccak-f* merupakan fungsi utama dalam keccak. Fungsi ini mengambil state sebagai masukan, dan melakukan sejumlah operasi permutasi yang terdiri dari 5 tahapan operasi yaitu: *non-linearity* (chi), *diffusion* (theta), *inter-slice dispersion* (rho), *disturbing horizontal/vertical alignment* (pi), dan *break symetri* (iota).

Operasi permutasi dalam fungsi keccak-f ini sering juga disebut dengan Round. Dalam setiap fungsi keccak-f dilakukan beberapa round. Jumlah round yang rekomendasikan dapat dihitung dengan menggunakan persamaan (7), dengan *l* sebagaimana dalam persamaan (3) [7]. Jadi untuk *keccak-f[1600]*, jumlah round yang direkomendasikan adalah 24.

$$N_r = 12 + 2l \quad \text{.....(7)}$$

Gambar 5 dan Gambar 4 merupakan algoritma dari keccak-f dan fungsi roundnya.

```
Round[b](A,RC) {
  θ step
  C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4],   forall x in 0..4
  D[x] = C[x-1] xor rot(C[x+1],1),                            forall x in 0..4
  A[x,y] = A[x,y] xor D[x],                          forall (x,y) in (0..4,0..4)

  ρ and π steps
  B[y,2*x+3*y] = rot(A[x,y], r[x,y]),                forall (x,y) in (0..4,0..4)

  χ step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]), forall (x,y) in (0..4,0..4)

  ι step
  A[0,0] = A[0,0] xor RC

  return A
}
```

Gambar 4. *Psuedo code* algoritma fungsi round [7]

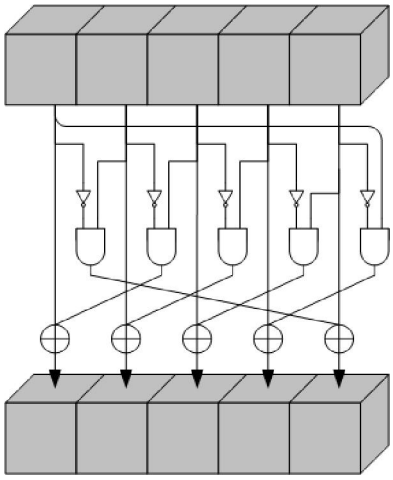
```
KECCAK-f[b](A) {
  forall i in 0..nr-1
    A = Round[b](A, RC[i])
  return A
}
```

Gambar 5. *Psuedo code* algoritma keccak-f [7]

C.1. Operasi non-linearity / χ (chi)

Operasi *non-linearity* merupakan satu-satunya operasi *non-linear mapping* dalam keccak-f. Tanpa operasi ini, fungsi round keccak akan menjadi linier. Operasi ini dapat dipandang sebagai aplikasi 5*w* operasi S-Box untuk 5-bit baris.

Operasi ini sendiri bersifat *invertible* atau dapat dibalikan, invers dari χ itu sendiri bersifat berbeda [4]. Operasi non-linier χ adalah komplemen dari fungsi non-linier γ padar RadioGatun, Panama, dan beberapa algoritma lainnya. Gambar 6 dan Gambar 7 masing-masing menjelaskan proses dan algoritma operasi ini.



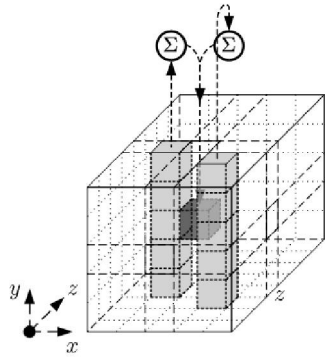
Gambar 6. *Operasi non-linearity/χ [9]*

for $y = 0$ **to** 4 **do**
  for $x = 0$ **to** 4 **do**
    $A[x, y] = a[x, y] \oplus ((\text{NOT}\, a[x + 1, y])\, \text{AND}\, a[x + 2, y])$
  end for
end for

Gambar 7. Algoritma *operasi non-linearity/χ [4]*

C.2. Operasi diffusion / θ (theta)

Operasi *diffusion* bersifat linier. Operasi ini hanya men-xor-kan 11 bit menjadi satu. Oleh karena itu, setiap bit berpengaruh terhadap sebelas bit lainnya. Dalam proses ini terjadi 50 XOR dan 5 rotasi. Gambar 8 dan 9 masing-masing menjelaskan tentang proses dan algoritma dari proses ini.



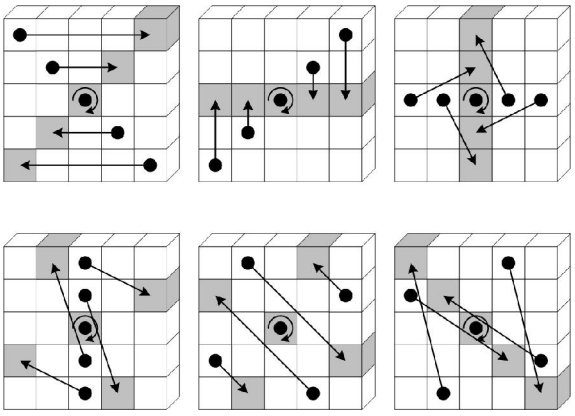
Gambar 8. *Operasi diffusion/θ [9]*

for $x = 0$ **to** 4 **do**
  $C[x] = a[x, 0]$
  for $y = 1$ **to** 4 **do**
    $C[x] = C[x] \oplus a[x, y]$
  end for
end for
for $x = 0$ **to** 4 **do**
  $D[x] = C[x - 1] \oplus \text{ROT}(C[x + 1], 1)$
  for $y = 0$ **to** 4 **do**
    $A[x, y] = a[x, y] \oplus D[x]$
  end for
end for

Gambar 9. Algoritma operasi diffusion/θ[4]

C.3. Operasi disturbing horizontal/vertical alignment / π (pi)

Operasi *disturbing horizontal/vertical alignment* adalah sebuah operasi transposisi terhadap sebuah lane yang menyediakan dispersi dan bertujuan untuk mendapatkan *long-term diffusion* [4]. Inti dari operasi ini adalah mengalikan setiap bit pada *slice* dengan matrix [[0,1],[2,3]]. Gambar 10 dan 11 masing-masing menjelaskan tentang proses dan algoritma dari proses ini.



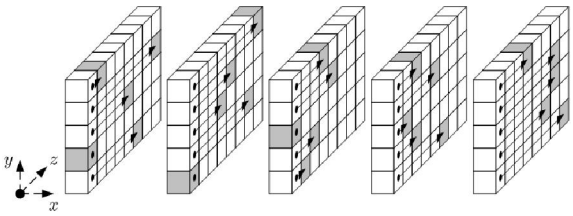
Gambar 10. *Operasi π [9]*

for $x = 0$ to 4 do
  for $y = 0$ to 4 do
    $\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
    $A[X, Y] = a[x, y]$
  end for
end for

Gambar 11. Algoritma *operasi π [4]*

C.4. Operasi inter-slice dispersion / ρ (rho)

Operasi *inter-slice dispersion* terdiri dari operasi translasi dalam *lane*. Tanpa opersai ini, difusi antara *slice* akan menjadi sangat lambat. Operasi ini juga bersifat linier, dengan inverse berupa penggeseran ulang yang bekebalikan dengan penggeseran sebelumnya. Gambar 12 dan 13 masing-masing menjelaskan tentang proses dan algoritma dari proses ini.



Gambar 12. *Operasi inter-slice dispersion [9]*

$A[0, 0] = a[0, 0]$
$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
for $t = 0$ to 23 do
  $A[x, y] = \mathrm{ROT}(a[x, y], (t + 1)(t + 2)/2)$
  $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
end for

Gambar 13. Algoritma *opr. inter-slice dispersion [4]*

C.5. Operasi break symetri / ι (iota)

Operasi *break symetri* terdiri dari penambahan konstanta round yang bertujuan untuk membuyarkan kesimetrisan. Jumlah posisi bit aktif dalam konstanta round adalah *l + 1*. Jika *l* bertambah, konstanta round akan menambah ke-asimetrisan lebih banyak.

IV. ANALISIS KEAMANAN DAN IMPLEMENTASI

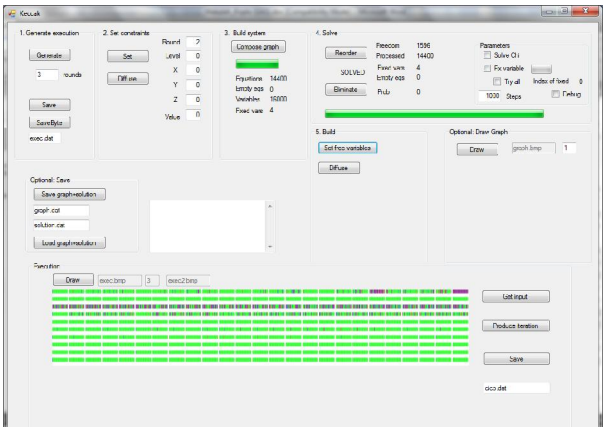
Dalam hal kriptografi, keamanan suatu rancangan enkripsi atau dekripsi sangatlah penting. Oleh karena itu setiap kali muncul metode enkirpsi atau dekripsi yang baru harus diuji terlebih dahulu keamanannya agar bisa menjadi bahan pertimbangan bagi penggunanya.

A. Keamanan

Untuk menaksir keamaan dari sebuah konstruksi dapat digunakan *indifferentiability framework* yang diperkenalkan oleh Maurer, Renner dan Holenstein. Dalam penelitian "*Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology*"[2] Sebelumnya sudah dibuktikan bahwa probabilitas keberhasilan suatu difenesiasi *sponge construction* memanggil permutasi yang bersifat acak dibatasi atasnya oleh $1 - exp(-N^2 2^{-(c+1)})$, dengan N adalah jumlah pemanggilan *f* atau *inverse*-nya. Batasan ini menyederhanakan batasan bawahnya $\sqrt{\pi}\, 2c/2$ untuk kompleksitas yang diharapkan dari *differentiating attack* yang berhasil.

Seperti dijelaskan pada bagian III, keccak menggunakan struktur spon. Penggunaan struktur spon ini aman terhadap *generic attack* karena sesuai dengan *indifferentiability proof*. Selain itu, fungsi permutasi keccak-f juga didesain sedemikian rupa sehingga tidak ada properti yang dapat dieksploitasi.

Salah satu masalah dalam keccak adalah adanya masalah CICO (*Constrained input constrained output*). Dalam [1], penelitian dilakukan untuk menyelesaikan masalah ini, yaitu dengan menggunakan triangulation tool. Pada percobaan ini dilakukan pengesetan *fixed variable* untuk mencari *free variable*. Gambar 14 merupakan hasil dengan solusi untuk jumlah round 3, dan pengesetan 2 *fixed variable*.



Gambar 14. Triangulation tool

B. Implementasi

Pada implementasinya, *keccak* memiliki beberapa keunggulan, diantaranya yaitu *security-speed trade off*. Dalam hal ini, implementasi bisa dilakukan bergantung apa yang dibutuhkan oleh pengguna, yaitu kecepatan atau keamanan.

c-bit terakhir dari *state* tidak pernah secara langsung terpengaruh oleh blok-blok masukan dan tidak pernah dikeluarkan selama fase *squeezing*. Kapasitas *c* sebenarnya menentukan tingkat keamanan yang dapat

dicapai dari sebuah konstruksi.

Tingkat keamanan dalam keccak berbanding terbalik dengan kecepatannya. Sehingga, semakin tinggi tingkat keamanan implementasi keccak, maka kecepatannya akan semakin berkurang. Hal ini dikarenakan, hubungan bitrate dan kapasitas yang saling membentuk panjang permutasi, yaitu *bitrate* ditambah dengan *capacity* ini menghasilkan panjang permutasinya, maka suatu keccak untuk panjang permutasi yang sama dapat dikenakan dua pilihan, yaitu:

1. Lebih cepat namun kurang aman. Hal ini dapat dilakukan dengan meningkatkan bitrate-nya, yang otomatis akan mengurangi kapasitas.
2. Lebih aman namun kurang cepat / lebih lamban. Hal ini dapat dilakukan dengan meningkatkan kapasitas, yang secara otomatis mengurangi bitratenya.

Contoh kasus untuk masalah tersebut adalah untuk kasus *keccak-f[1600]*, yang berarti r + c = 1600, dengan asumsi tingkat sekuritas $2^{c/2}$ dan tingkat kecepatan adalah r.

1. Jika r = 1024, dan c = 576, memiliki tingkat sekuritas $2^{c/2}=2^{288}$, dan tingkat kecepatan 1024.
2. Jika r = 512, dan c = 1028, memiliki tingkat sekuritas $2^{c/2}=2^{544}$, dan tingkat kecepatan 512.

Dari hasil diatas didapatkan bahwa hasil pertama jauh lebih cepat dari pada hasil yang kedua, namun hasil yang ke dua jauh lebih aman dari pada hasil yang pertama.

Penghitungan performansi implementasi keccak pada software telah dilakukan pada [4], dengan menggunakan *keccak-f[1600]* pada dua platform yang berbeda yaitu:

1. Platform A:
   - Linux openSUSE 11.0 x86 64
   - CPU: Intel Xeon 5150 (CPU ID: 06F6), 2.66GHz, dual core
   - 1333MHz and 4Mb of level-2 cache
   - Compiler: GCC 4.4.1 using "gcc -O3 -g0 -march=barcelona" untuk 64-bit code dan penambahan "-m32" untuk 32-bit code.
2. Platform B:
   - Vista Ultimate x86 or x64, version 6.0.6001, SP1 build 6001;
   - CPU: Intel Core2 Duo E6600 at 2.4GHz;
   - x86: Microsoft Visual Studio 2008 Version 9.0.21022.8 RTM, 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 untuk 80x86
   - x64: Microsoft Visual Studio 2008 version 9.0.30428.1 SP1Beta1, Microsoft Windows SDK 6.1 Targeting Windows Server 2008 x64 C/C++ Optimizing Compiler Version 15.00.21022.08 untuk x64.

Code yang dibangun menggunakan bahasa pemograman C biasa. Gambar 14 dan Gambar 15 memperlihatkan hasil penghitungan performansi yang dilakukan terhadap ke dua platform tersebut.

Operation	Platform A	Platform B
KECCAK-f[1600] only	1648 c	1845 c
Squeezing with $r = 1024$	12.9 c/b	14.4 c/b
KECCAK-f and XORing 1024 bits	1680 c	1854 c
Absorbing with $r = 1024$	13.1 c/b	14.5 c/b

Gambar 14. Perofmansi dengan kompilasi menggunakan 64-bit instruksi [4]

Operation	Platform A	Platform B
KECCAK-f[1600] only	4408 c	5904 c
Squeezing with $r = 1024$	34.4 c/b	46.1 c/b
KECCAK-f and XORing 1024 bits	4600 c	6165 c
Absorbing with $r = 1024$	35.9 c/b	48.2 c/b

Gambar 15. Perofmansi dengan kompilasi menggunakan 32-bit instruksi [4]

Dengan c = *cycle*; dan c/b = *cycle/byte*.

Dari gambar diatas didapatkan bahwa performansi yang dilakukan dengan kompilasi menggunakan instruksi 32-bit lebih rendah dari pada 64-bit, karena memang instruksi 64-bit bisa menyelesaikan perhitungan dengan lebih cepat karena mampu menyelesaikan 64-bit setiapkalinya. Untuk meningkatkan performansi dari 32-bit dilakukan optimasi dengan menggunakan SIMD atau *single instruction multiple data*. Pada Gambar 16 dan Gambar 17 terlihat bahwa setelah dilakukan optimasi dengan menggunakan SIMD, performansi dari 32-bit dan 64-bit menjadi meningkat.

Operation	Platform A	Platform B
KECCAK-f[1600] only	2520 c	2394 c
Squeezing with $r = 1024$	19.7 c/b	18.7 c/b
KECCAK-f and XORing 1024 bits	2528 c	2412 c
Absorbing with $r = 1024$	19.8 c/b	18.8 c/b

Gambar 16. Perofmansi dengan optimasi dan kompilasi menggunakan 64-bit instruksi [4]

Operation	Platform A	Platform B
KECCAK-f[1600] only	2816 c	2475 c
Squeezing with $r = 1024$	22.0 c/b	19.3 c/b
KECCAK-f and XORing 1024 bits	2832 c	2475 c
Absorbing with $r = 1024$	22.1 c/b	19.3 c/b

Gambar 17. Perofmansi dengan optimasi dan kompilasi menggunakan 32-bit instruksi [4]

Pada implementasi algoritma keccak dalam *hardware*, bergantung pada arsitektur yang digunakan. Hal ini dikarenakan perbedaan arsitektur menyebabkan perbedaan *trade-off* kecepatan dan sekuritas yang sudah dijelaskan sebelumnya.

Terdapat dua arsitektur yang sudah dipelajari dan

diimpelemntasikan algoritma keccak ini, yaitu *high-speed core* dan *low-area coprocessor*. Pada *high-speed core*, *core* terdiri dari 3 komponen utama, yaitu: fungsi round, *state register* dan *input/output buffer*. Pada fase *absorbsing, I/O buffer* memberikan transfer input secara simultan melalui bus dan komputasi keccak-f untuk input sebelumnya. Sedangkan dalam fase *squeezing*, *I/O buffer* memberikan transfer output secara simultan melalui bus dan komputasi keccak-f untuk output setelahnya.

Selain itu, Keccak telah diimplementasikan dalam VHDL untuk diterapkan pada ASIC dan FPGA [4][8]. Namun implementasi keccak dalam bahasa VHDL sangat sulit [8]. Tapi implementasi dalam FPGA yang telah dilakukan bisa mendapatkan performansi yang tinggi dengan cukup mudah.

V. KESIMPULAN

Algoritma keccak merupakan algoritma yang berbasis struktur spon, yang terdiri dari 2 fase yaitu *absorbing* dan *squeezing*. Pada fase *absorbing*, dilakukan permutasi terhadap input masukan, sedangkan pada fase *squeezing* dilakukan permutasi dan penggabungan untuk mendapatkan hasil keluaran.

Fungsi permutasi *keccak-f* merupakan fungsi utama dalam keccak. Fungsi ini mengambil state sebagai masukan, dan melakukan sejumlah operasi permutasi yang terdiri dari 5 tahapan operasi yaitu: *non-linearity* (chi), *diffusion* (theta), *inter-slice dispersion* (rho), *disturbing horizontal/vertical alignment* (pi), dan *break symetri* (iota). Kesemua proses tersebut bertunjuan untuk memberikan fungsi random sehingga tidak mudah diserang oleh *generic attack*.

Implementasi keccak dalam software dan hardware sudah dilakukan dan memberikan hasil yang baik [4]. Pada implementasinya dalam software, penggunaan SIMD (*single instruction multiple data*). Hal ini ditandai dengan peningkatan performansi yang terjadi baik pada kompilasi dengan menggunakan 64-bit maupun dengan 32-bit.

DAFTAR PUSTAKA

[1] Aumasson, Jean-Philippe & Dmitry Khovratovich. (2009). *First Analysis of Keccak*. University of Luxemborg.
[2] Bertoni, Guido, et al. (2009). *Cryptographic Sponge*.
[3] Bertoni, Guido, et al. (2008). *Keccak Specifications*.
[4] Bertoni, Guido, et al. (2010). *Keccak Sponge Function Family Main Document*.
[5] Boura, Christina & Anne Canteaut. (2010). *A Zero-Sum Property for the Keccak-f Permutation with 18 Rounds*.
[6] Munir, Rinaldi. (2005). *Diktat Kuliah IF5054: Kriptografi*. Departemen Teknik Informatika.
[7] Ros, Alexander & Carl Sigurjonsson. (2010). *Introduction to SHA-3 CubeHash vs. Keccak*.
[8] Strombergson, Joachim. (2008). *Implementation of the Keccak Hash Function in FPGA Devices*.
[9] http://keccak.noekeon.org
[10] http://plaintext.crypto.lo.gy/article/495/untwisted-a-cryptol-implementation-of-keccak-part-1
[11] http://sponge.noekeon.org

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, 10 Mei 2011

ttd

Moch. Yusup Soleh / 13507051