LUX HASH FUNCTION

Brian Al Bahr – NIM: 13506093

Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung

Jl. Ganesha 10, Bandung

E-mail: if16093@students.if.itb.ac.id, bhbrayeun@yahoo.co.id

Abstrak

Definisi dari fungsi *hash* adalah sebuah fungsi matematis yang melakukan konversi terhadap *string* masukan yang panjang menjadi menjadi *string* lain dengan panjang yang selalu sama, tidak bergantung pada panjang *string* masukannya, dan biasanya dengan ukuran yang jauh lebih kecil daripada *string* masukannya. Nilai kembalian dari fungsi *hash* biasanya disebut *hash values*, *hash codes*, atau *hash sums*.

Fungsi *hash* pada umumnya digunakan untuk mempercepat *table look-up*, atau untuk membandingkan data (misalnya mencari data tertentu dalam sebuah basis data, mendeteksi data yang terduplikasi dalam sebuah *file* berukuran besar, dan sebagainya).

LUX *hash function* merupakan salah satu varian *fungsi hash* yang berbeda dengan struktur MD, karena LUX berbasis aliran (*stream*). Struktur fungsi *hash* berbasis aliran dapat menangani *length extension attack, herding attack, multi-collision attack*, dan *meet-in-the-middle attack*. Kelebihan dan kekurangan, serta pembahasan secara lebih detail akan dibahas dalam makalah ini.

Kata kunci: *hash function*, LUX

1. PENDAHULUAN

LUX dirancang oleh Ivica Nikolić, Alex Biryukov, dan Dmitry Khovratovich, melalui kompetisi NIST's SHA-3. LUX menggunakan struktur berbasis aliran, yang berbeda dengan struktur *Message Digest* (*MD*). Selain LUX, juga terdapat beberapa fungsi *hash* berbasis aliran seperti Radiogatun, Panama, Grindahl, dan Enrupt. Struktur fungsi *hash* yang berbasis aliran dapat bertahan terhadap *length extension attack, herding attack, multi-collision attack,* dan *meet-in-the-middle attack.*

Selain kelebihan di atas, fungsi *hash* berbasis aliran juga memiliki kelemahan, yaitu diperlukan ukuran *buffer* yang sangat besar jika dibandingkan dengan struktur *MD*. Selain itu, fungsi-fungsi *hash* tersebut tidak dapat diparalelkan.

2. ISI

2.1 Spesifikasi LUX *hash function*

Secara umum, perancangan LUX *hash function* didasarkan pada empat properti berikut:

- Fungsi *hash* berbasis aliran
- Proses internal yang besar, mencapai tiga kali lipat *Message Digest*
- Pesan diproses dalam potongan-potongan kecil, dengan ukuran 32 bit atau 64 bit
- Fungsi pada setiap putaran menggunakan transformasi seperti pada Rijndael

Sedangkan proses internal yang terjadi dalam LUX *hash function* adalah sebagai berikut:
Status internal dalam fungsi ini terbagi menjadi dua bagian, yaitu *buffer* dan *core*.

- *Buffer – matrix of bytes* berukuran *m* x 16
- *Core – matrix of bytes* berukuran *m* x 8

Output	m	Core	Buffer	Total
256	4	4 x 8	4 x 16	96
512	8	8 x 8	8 x 16	192

Tabel 1 Spesifikasi LUX

Kemudian dilakukan *feedforward* antara *buffer* dan *core* pada setiap putaran.

2.1.1 *State Update Function* (Φ)

Secara garis besar, fungsi ini melakukan *update* dari status *S* awal (*buffer* + *core*) dan blok pesan M_t dan menghasilkan *state* baru:

$$S_{\text{baru}} \leftarrow \Phi(S_{\text{lama}}, M_t)$$

Proses ini disebut satu ronde atau satu putaran. Proses dalam satu ronde ini dapat dibagi menjadi empat subproses:

- Tambahkan blok ke pesan ke *buffer* dan *core*
  $B_0 \leftarrow B_0 \oplus M_t$
  $C_0 \leftarrow C_0 \oplus M_t$

- *Update buffer* dan *core* secara terpisah
  $B \leftarrow F(B)$
  $C \leftarrow G(C)$

- Tambahkan *core* ke *buffer*
  for $i = 0$ **to** 7 **do**
  $B_{i+4} \leftarrow B_{i+4} \oplus C_i$

- Lakukan *feedforward* kolom *buffer* ke *core*
  $C_7 \leftarrow C_7 \oplus B_{15}$

Pada langkah kedua, kita mendefinisikan fungsi *F* dan *G* untuk melakukan *update* terhadap *buffer* dan *core*. Fungsi *F* merupakan sebuah fungsi siklis sederhana yang merotasikan matriks satu kolom ke kanan. Adapun algoritma untuk fungsi *F* adalah:

Input $B = B_0 \| B_1 \| \ldots \| B_{15}$

for $i = 0$ **to** 15 **do**
  $B'_{(i+1) \bmod 16} \leftarrow B_i$

Output B'

Sedangkan fungsi *G* adalah fungsi untuk memanipulasi bagian *core*, yang merupakan satu ronde Rijndael. Transformasinya sendiri terdiri atas empat bagian, yaitu *SubBytes*, *ShoftRows*, *Mix-Columns*, dan *AddConstant*.

Input C

$C \leftarrow SubBytes(C)$
$C \leftarrow ShiftRows(C)$
$C \leftarrow MixColumns(C)$
$C \leftarrow AddConstant(C)$

Output C

SubBytes

$$S(X) = Y$$

di mana $X = X_1 \| X_2$, X_1 adalah 4 bit pertama dan X_2 adalah 4 bit terakhir dari *X*. *S-box* yang digunakan dalam *LUX hash function* adalah:

X1 \ *X2*	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Tabel 2 *S-box* pada LUX

ShiftRows

Fungsi ini merotasikan setiap baris pada matriks ke kiri.

$$C_{i,j} \leftarrow C_{i,(j+v_j) \bmod 8}$$

MixColumns

Operasi ini dilakukan terhadap setiap kolom matriks secara terpisah. Untuk LUX-224 dan LUX-256 transformasinya adalah sebagai berikut:

$$C_i^{\text{baru}} \leftarrow A \cdot C_i^{\text{lama}}; A = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

Sedangkan untuk LUX-384 dan LUX-512, matriks yang digunakan adalah:

$$A = \begin{pmatrix} 01 & 04 & 01 & 01 & 02 & 0c & 06 & 08 \\ 08 & 01 & 04 & 01 & 01 & 02 & 0c & 06 \\ 06 & 08 & 01 & 04 & 01 & 01 & 02 & 0c \\ 0c & 06 & 08 & 01 & 04 & 01 & 01 & 02 \\ 02 & 0c & 06 & 08 & 01 & 04 & 01 & 01 \\ 01 & 02 & 0c & 06 & 08 & 01 & 04 & 01 \\ 01 & 01 & 02 & 0c & 06 & 08 & 01 & 04 \\ 04 & 01 & 01 & 02 & 0c & 06 & 08 & 01 \end{pmatrix}$$

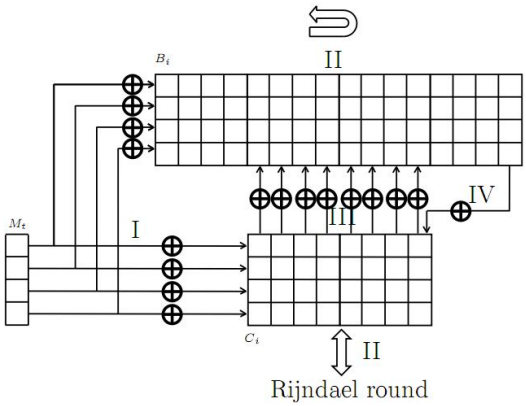
AddConstant

Dalam fungsi *hash*, seringkali digunakan konstanta yang digunakan untuk mencegah serangan.

$$C_0 \leftarrow C_0 \oplus 0x2ad01c64$$

2.1.2 Inisialisasi dan *Padding*

Semua elemen pada matriks *buffer* dan *core* diinisialisasi dengan nilai nol. Kemudian, pesan dibagi-bagi menjadi sejumlah blok dengan ukuran masing-masing *m bytes* (8*m* bits). Jika ukuran blok terakhir kurang dari *m*, maka ditambahkan bit-bit 1 hingga panjangnya sesuai. Jika blok terakhir tepat berukuran 8*m* bits, maka ditambahkan sebuah blok yang berisi sebuah bit 1, diikuti dengan bit-bit 0.

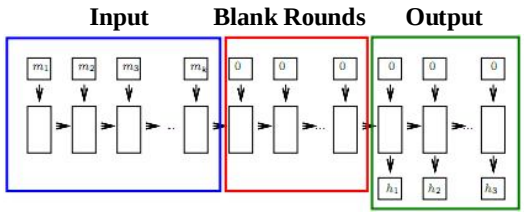


Gambar 1 *Round function* pada LUX

I : pesan ditambahkan ke *buffer* dan *core*
II : *update buffer* dan *core*
III : penambahan *core* ke *buffer*
IV : penambahan 1 kolom *buffer* ke *core*

2.1.3 *Hashing* dan *Output*

Seperti pada fungsi *hash* berbasis aliran lainnya, LUX menggunakan tiga fase yang sama, yaitu fase *input*, fase *blank rounds*, dan fase *output*. Eksekusi ketiga fase ini dilakukan berurutan.



Gambar 2 *Hashing* dan *Output*

Fase *Input*

Setelah inisialisasi dan *padding* terhadap pesan selesai dilakukan, fase *input* dimulai. Pada fase ini, keseluruhan pesan "diserap", blok per blok, tanpa menghasilkan *output* apa pun. Setiap blok pesan, mulai dari blok pertama, dilewatkan sebagai argument untuk fungsi *state update* Φ, yang mentransformasikan *state* internal menjadi *state* baru, dalam setiap rondenya.

Fase input berakhir ketika semua blok pesan telah selesai diproses secara sekuensial. Oleh karena itu, tampak bahwa jumlah ronde pada fase ini sangat bergantung pada ukuran pesan.

Fase *Blank Rounds*

Fase ini digunakan untuk meningkatkan *diffusion* dari blok pesan terakhir. Untuk LUX, fase ini terdiri atas 16 ronde. Pada setiap ronde, blok pesan yang menjadi masukan adalah blok dengan nilai nol.

Fase *Output*

Fase ini merupakan fase terakhir, yang akan menghasilkan nilai *hash* dari keseluruhan pesan. Pada setiap rondenya, blok pesan yang menjadi masukan (sama seperti pada fase sebelumnya) adalah blok dengan nilai nol. Namun, setiap ronde akan menghasilkan *output* berukuran *m* bytes.

Output setiap ronde adalah nilai kolom C_3 pada matriks *core*. Oleh karena itu, LUX-224 menghasilkan *output* pada ronde ke 7, LUX-256 pada ronde ke 6, dan LUX-512 pada ronde ke 8.

2.2 Perancangan

Pada dasarnya, LUX dirancang untuk keamanan dan efisiensi, seperti pada rancangan Panama dan RadioGatun yang juga berbasis aliran, yang

terbukti efisien. Untuk itu, dipilih sebuah model transformasi yang sudah terbukti efisien dan aman, yaitu Rijndael.

2.2.1 Prinsip Umum

Inisialisasi dengan nilai nol telah terbukti sebagai cara yang praktis dan tidak menimbulkan ancaman keamanan apa pun. *Padding* pada pesan saat ini juga telah menjadi prosedur standar untuk merancang fungsi *hash* modern.

LUX mengalokasikan blok tambahan pada akhir pesan karena ukuran blok pesan terbatas pada *m* bytes. Pada LUX-224 dan LUX-256, jika *m* = 4, blok pesan berukuran 32 bit. Untuk memungkinkan melakukan *hash* terhadap pesan hingga mencapai 2^{64}, diperlukan dua blok pesan.
Jumlah ronde dalam fase *blank rounds* dipilih sedemikian rupa sehingga blok pesan terakhir telah melewati semua tahap dari *buffer*. Dengan demikian, *core* mendapat pengaruh ganda dari blok pesan terakhir, juga membatasi kemungkinan penyerang untuk memanipulasi *core* hanya pada blok pesan terakhir.

2.2.2 Perancangan *Core*

Ukuran *core* yang digunakan yaitu *m* x 8, untuk mengurangi rasio antara pesan masukan (*m* bytes) dan ukuran *core* (8*m* bytes). Sehingga, pada LUX, rasionya adalah 1/8, yang membuat tingkat keamanan lebih tinggi dibanding fungsi *hash* lainnya.

2.2.3 Perancangan *Buffer*

Jika pada Panama digunakan *buffer* linier yang sederhana, yang hanya berdasarkan pada putaran, pada *buffer* RadioGatun sedikit lebih kompleks, yaitu dengan ditambahkannya suatu *feedback* dari *core* ke *buffer*. Dengan cara ini, analisis terhadap fungsi menjadi lebih sulit karena *feedback* pada akhir *buffer* bergantung pada *state* sebelumnya pada *core*. LUX mengadopsi ide ini; *bufferi* pada LUX mendapatkan *feedback* dari *core*.

Oleh karena itu, ukuran *buffer* pada LUX adalah dua kali lipat ukuran *core*. Jika matriks *core* berukuran *m* x 8, maka ukuran matriks *buffer* adalah *m* x 16. Karena adanya transformasi *buffer*, baik itu rotasi kolom, penambahan kolom *core* atau operasi lainnya, maka *buffer* dapat dibedakan menjadi dua bagian:

- *Middle part* : bagian kolom 5 sampai 12, di mana terjadi rotasi kolom dan *feedback* dari *core*,
- *Passive part* : bagian kolom 1-4 dan 13-16, di mana hanya terjadi rotasi kolom.

Jika jumlah kolom pada *passive part* ditingkatkan, hanya akan sedikit mempengaruhi efisiensi karen *cost* untuk melakukan rotasi kolom adalah nol. Namun, penambahan ini berpengaruh besar pada keamanan dari fungsi ini. Pada awalnya, dipilih 4 kolom di bagian awal *passive part* karena blok pesan yang ditambahkan pada *core* mencapai kondisi *full diffusion* setelah 3 ronde/putaran Rijndael. Jika ditambahkan sebuah kolom, maka *diffusion* akan meningkat, sehingga tingkat keamanan semakin tinggi. Hal yang sama juga dapat dilakukan terhadap kolom di bagian akhir *passive part*.

2.3 Analisis Tingkat Keamanan

Tabel di bawah ini menunjukkan tingkat keamanan LUX *hash function* (LUX-224, LUX-256, LUX-384, dan LUX-512) terhadap berbagai jenis serangan:

Digest	**Kompleksitas Serangan**			
	224	**256**	**384**	**512**
Collision attack	2^{112}	2^{128}	2^{192}	2^{256}
Preimage attack	2^{224}	2^{256}	2^{128}	2^{512}
Second preimage attack	2^{224}	2^{256}	2^{128}	2^{512}
HMAC-PRF distinguisher	2^{112}	2^{128}	2^{192}	2^{256}
Randomized hashing attack	2^{112}	2^{128}	2^{192}	2^{256}

Tabel 3 Tingkat Keamanan LUX

Dari tabel di atas, tampak bahwa kompleksitas untuk melakukan serangan terhadap LUX *hash function* cukup tinggi, sehingga dapat disimpulkan bahwa tingkat keamanan LUX *hash function* tergolong tinggi.

2.4 Implementasi LUX

Pada implementasinya, LUX memberikan performanasi yang cukup baik, dengan kecepatan sebagai berikut (dalam *cycles per byte*):

	LUX-256	LUX-512
32-bit (C)	14.8	28.2
64-bit (asm)	10.2	9.5

Tabel 4 Implementasi LUX

Kecepatan pada *platform* 32-bit dapat ditingkatkan jika implementasi dilakukan dengan *assembly*.

2.5 Kelebihan LUX

LUX *hash function* memiliki beberapa kelebihan, yaitu:

- Kriptanalisis hanya dapat dilakukan pada konstruksinya.
- Trik-trik yang digunakan pada implementasi Rijndael juga dapat diterapkan pada LUX.
- Kecepatan tinggi, merupakan salah satu fungsi *hash* yang berjalan tercepat pada *platform* 32 dan 64 bit.
- Kecepatan stabil pada berbagai jenis prosesor (AMD, Intel)
- Menggunakan semua fungsi berbasis AES.

3. KESIMPULAN

Berikut ini merupakan beberapa kesimpulan yang dapat diambil mengenai LUX *hash function*:

- LUX memiliki daya tahan yang kuat terhadap berbagai jenis serangan; tingkat keamanannya tinggi.
- Performansi yang diberikan LUX bagus, ditinjau dari sisi kecepatan dan efisiensi.

DAFTAR PUSTAKA

[1] Munir, Rinaldi, (2006). Diktat Kuliah IF5054 Kriptografi, Departemen Teknik Informatika Institut Teknologi Bandung.

[2] http://en.wikipedia.org/wiki/Hash_function

[3] https://cryptolux.org/mediawiki/uploads/f/f3/LUX.pdf

[4] http://eprint.iacr.org/2008/520.pdf

[5] http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/Feb2009/documents/LUX.pdf