Desain PRNG Berbasis Fungsi *Hash* Menggunakan GIMLI

Galuh Dian Pradana[1)], Nurul Qomariasih[2)]
(1) *Rekayasa Kriptografi, Politeknik Siber dan Sandi Negara, galuhdianpradana@gmail.com*
(2) *Rekayasa Kriptografi, Politeknik Siber dan Sandi Negara, nurul.qomariasih@poltekssn.ac.id*

Abstrak

Rangkaian bit acak merupakan komponen penting dalam kriptografi yang digunakan dalam berbagai hal, salah satunya digunakan sebagai kunci masukan dalam algoritme kriptografi. Rangkaian tersebut dapat diperoleh dari RNG atau PRNG. PRNG merupakan suatu mekanisme yang digunakan untuk menghasilkan bit-bit acak yang digunakan untuk membangkitkan kunci sesi, rangkaian kunci stream cipher, dan bilangan acak yang digunakan pada algoritme kunci publik yang harus memenuhi sifat randomness dan unpredictability. Berdasarkan NIST SP 800-90 A revisi 1 suatu PRNG dapat dibangun dari block cipher atau fungsi hash (HASH_DRBG). Berdasarkan permasalahan tersebut, maka peneliti mengajukan desain PRNG berbasis fungsi hash menggunakan fungsi pembangkit GIMLI_HASH. GIMLI_HASH merupakan fungsi hash yang menghasilkan keluaran 256 bit dan dibangun dari fungsi permutasi GIMLI. Keluaran PRNG diuji dengan tool uji keacakan sts 2.1.2 NIST SP 800-22 revisi 1a. Pengujian dilakukan menggunakan level signifikan $\alpha = 0.01$ *dengan pendekatan proporsi yang dilakukan dengan dua scenario menggunakan pengujian berupa seed sama yang diperoleh dari algoritme Mersenne Twister. Pengujian pertama digunakan additional_input berupa nilai null, tetapi pada pengujian kedua digunakan additional_input berupa nilai acak dengan panjang 64 bit. Dari hasil pengujian terhadap keluaran PRNG, disimpulkan bahwa keluaran PRNG lulus pada seluruh uji yang direkomendasikan pada NIST SP 800-22 revisi 1a.*

Kata kunci: GIMLI, GIMLI_*HASH*, PRNG, NIST SP 800-90A revisi 1, NIST SP 800-22 revisi 1a.

1. PENDAHULUAN

Perangkat dengan sumber daya terbatas memiliki batasan dalam konsumsi energi, memori, dan kemampuan komputasi. Oleh karena itu, untuk tetap memberikan layanan keamanan pada perangkat sumber daya terbatas dikembangkan *lightweight cryptography*. Skema *lightweight cryptography* yang banyak dikembangkan berbasis *block cipher*, dan fungsi *hash* dengan kunci. Skema tersebut merupakan skema yang membutuhkan kunci berupa rangkaian bit acak. Rangkaian bit merupakan komponen penting dalam kriptografi, sehingga adanya kelemahan pada barisan bit, dapat menyebabkan kelemahan pada sistem pengamanan [1]. Oleh karena itu, dibutuhkan pembangkit yang menghasilkan rangkaian bit acak (*random*) yang digunakan untuk skema kriptografi.

Pembangkit rangkaian bit acak dibedakan menjadi dua tipe, yaitu *Random Number Generator* (RNG) dan *Pseudorandom Number Generator* (PRNG) [2]. PRNG merupakan suatu mekanisme untuk menghasilkan bit-bit acak yang digunakan untuk membangkitkan kunci sesi, rangkaian kunci *stream cipher*, dan bilangan acak pada algoritme kunci publik yang harus memenuhi sifat *randomness* dan *unpredictability* [3]. Pemenuhan sifat *randomness* dan *unpredictability* pada PRNG dapat diuji dengan pengujian keacakan, salah satunya yaitu uji keacakan yang direkomendasikan pada NIST SP 800-22 revisi 1a. NIST SP 800-22 revisi 1a merupakan standar pengujian yang dikeluarkan *National Institute of Standards Technology* Amerika Serikat untuk menguji barisan bit keluaran RNG dan PRNG. NIST SP 800-22 revisi 1a terdiri dari 15 uji, yaitu *Frequency (Monobits) Test, Frequency Test within a Block, Runs Test, Test for the Longest Run of Ones in a Block, Binary Matrix Rank Test, Discrete Fourier Transform (Spectral) Test, Non-Overlapping Template Matching Test, Overlapping Template Matching Test, Maurer's "Universal Statistical" Test, Linear Complexity Test, Serial Test, Approximate Entropy Test, Cumulative Sums (Cusum) Test, Random Excursions Test, Random Excursions Variant Test* [2]. PRNG dinyatakan lulus uji apabila lulus pada semua uji tersebut. PRNG merupakan skema pembangkit bit acak yang dapat dibangun menggunakan fungsi *hash* atau *block cipher* [4]. PRNG memiliki beberapa fungsi yaitu fungsi pembangkit, fungsi *reseed,* fungsi *instantiate*, dan fungsi *uninstantiate*. Setiap fungsi menghasilkan nilai sama yaitu *initial state* dan nilai *constant* yang dibangkitkan dari nilai *seed* masukan PRNG. PRNG yang dibangun menggunakan fungsi *hash* memiliki empat masukan yaitu *seed*, *additional_input*, *constant*, dan *reseed_counter*. Fungsi *hash* dalam PRNG digunakan sebagai fungsi pembangkit untuk menghasilkan rangkaian bit acak yang diperlukan [4].

Pada tahun 2017, Bernstein *et al.* membuat skema permutasi yang diberi nama permutasi GIMLI. Permutasi GIMLI terdiri dari 24 *round*, setiap *round* terdiri dari tiga operasi yaitu *non-linear layer*, *linear mixing layer*, dan *constant addition.* Permutasi GIMLI dapat digunakan untuk membangun suatu fungsi *hash* (GIMLI_*HASH*) yang terbagi menjadi dua fungsi yaitu fungsi penyerapan (*absorb*) dan fungsi memeras (*squeeze*). Pada penelitian ini dilakukan desain PRNG berbasis fungsi *hash* menggunakan permutasi GIMLI. Permutasi GIMLI

digunakan untuk membangun GIMLI_*HASH* yang berfungsi sebagai pembangkit pada skema PRNG dengan masukan *seed* yang berasal dari algoritme Mersenne Twister.

2. LANDASAN TEORI

2.1 *Random Number Generator*

2.1.1 *Random Number Generator*

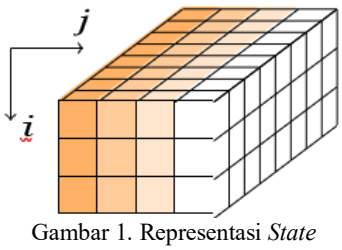
Random Number Generator (RNG) merupakan perangkat atau algoritme yang menghasilkan keluaran acak secara statistik dan tidak bias (*unbiased*) [6]. RNG menggunakan masukan yang bersifat *non-deterministic*, kemudian diproses menggunakan fungsi pemrosesan untuk menghasilkan keluaran yang acak [2]. Keluaran RNG dapat digunakan secara langsung untuk kunci masukan pada algoritme kriptografi atau sebagai masukan pada PRNG [2]. Keluaran RNG yang digunakan dalam algoritme kriptografi harus memenuhi kriteria keacakan berdasarkan suatu tes keacakan [2].

2.1.2 *Pseudorandom Number Generator*

Pseudorandom Number Generator (PRNG) merupakan suatu mekanisme yang digunakan untuk menghasilkan bit-bit acak yang digunakan untuk membangkitkan kunci sesi, rangkaian kunci *stream cipher*, dan bilangan acak yang digunakan pada algoritme kunci publik yang harus memenuhi sifat *randomness* dan *unpredictability* [3]. PRNG merupakan algoritme *deterministik* dengan masukan sebuah *seed* acak sepanjang k, menghasilkan keluaran rangkaian biner dengan panjang l yang lebih panjang dari pada k yang terlihat acak [6]. Keluaran PRNG yang digunakan untuk algoritme kriptografi harus bersifat acak yang dapat dipenuhi dengan tes keacakan serta nilai keluaran tidak berhubungan dengan nilai *seed* yang digunakan [3]

2.2 Permutasi GIMLI

GIMLI merupakan skema permutasi dengan masukan 384 bit yang didesain untuk keamanan yang tinggi pada *block cipher, tweakable block cipher, stream cipher, message-authentication codes, authenticated cipher*, fungsi *hash*, dan lain-lain [5]. GIMLI merepresentasikan masukan yang diproses sepanjang 384 bit *state* menjadi sebuah balok dengan dimensi $3 \times 4 \times 32$ atau matriks 3×4 dari 32-bit *words* seperti ditunjukan pada Gambar 1.



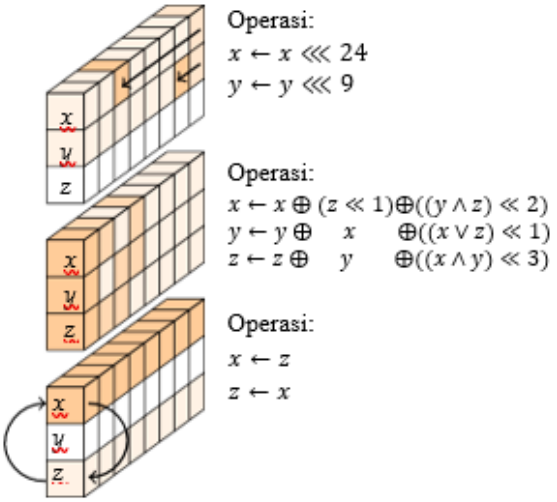
Gambar 1. Representasi *State*

Gambar 1 merepresentasikan masukan 384 bit *state* sebagai berikut :

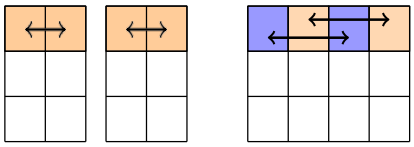
a. Kolom j merupakan rangkaian 96 bit sedemikian sehingga *state* $s_j = \{s_{0,j}; s_{1,j}; s_{2,j}\} \in \mathcal{W}^3, j = 1,2,\ldots$
b. Baris i merupakan rangkaian 128 bit sedemikian sehingga *state* $s_i = \{s_{i,0}; s_{i,1}; s_{i,2}; s_{i,3}\} \in \mathcal{W}^4, i = 1,2,\ldots$

GIMLI merupakan skema permutasi yang terdiri dari 24 *round*, dengan setiap *round* pada GIMLI terdiri dari tiga rangkaian operasi yaitu *non-linear layer*, *linear mixing layer*, dan *constant addition* yang dijelaskan pada Algoritme 1. *Non-linear layer* merupakan operasi yang terdiri dari tiga suboperasi yaitu rotasi *words* pertama dan kedua, 3-input *nonlinear T-function*, dan pertukaran *words* pertama dengan *words* ketiga seperti pada Gambar 2. *Linear layer* merupakan operasi yang terdiri dari dua operasi pertukaran, *Small-Swap* dan *Big-Swap* seperti dijelaskan pada Gambar 3. *Small-Swap* dilakukan setiap empat *round* sekali dimulai dari *round* pertama, dan *Big-Swap* dilakukan setiap empat *round* sekali dimulai dari *round* ketiga. *Round constant* merupakan operasi yang dilakukan setiap empat *round* sekali, yaitu melakukan operasi XOR antara *state word* pertama ($s_{0,0}$) dan $0x9e377900 \oplus r$ dengan $\in \{24, 20, 16, \ldots, 4\}$.

GIMLI merupakan permutasi yang dapat digunakan untuk membangun suatu fungsi *hash* (GIMLI-*HASH*). Fungsi *hash* ini terbagi menjadi dua fungsi yaitu fungsi penyerapan (*absorb*) dan fungsi memeras (*squeeze*) yang dijelaskan pada Algoritme 2 dan Algoritme 3. GIMLI-*HASH* menginisialisasi nol untuk 48 *bytes state* dengan masukan pesan sepanjang 16 *bytes* dan menghasilkan keluaran (*digest*) sepanjang 256 bit seperti pada Algoritme 4. Fungsi ***touint32*** digunakan untuk mengonversi nilai sepanjang 4 *bytes* menjadi 32 bit *unsigned integer* dalam *little-endian*, dan ***tobytes*** digunakan untuk mengonversi nilai 32 bit *unsigned integer* menjadi 4 *bytes*.



Gambar 2. *Non Linear Layer*



Small-Swap *Big-Swap*

Gambar 3. *Non Linear Layer*

Algoritme 1 Permutasi GIMLI

INPUT: $s = (s_{i,j}) \in \mathcal{W}^{3\times4}$
OUTPUT: $GIMLI(s) = (s_{i,j}) \in \mathcal{W}^{3\times4}$
For r dari 24 sampai 1, ***do***
  For j dari 0 sampai 3, ***do***
    $x \leftarrow s_{0,j} \lll 24$ $\triangleright\ SP-box$
    $y \leftarrow s_{1,j} \lll 9$
    $z \leftarrow s_{2,j}$
    $s_{2,j} \leftarrow x \oplus (z \ll 1) \oplus ((y \wedge z) \ll 2)$
    $s_{1,j} \leftarrow y \oplus x \oplus ((x \vee z) \ll 1)$
    $s_{0,j} \leftarrow z \oplus y \oplus ((x \wedge y) \ll 3)$
  End for
  $\triangleright\ linear\ layer$
  If $r \bmod 4 = 0$, ***then*** $\triangleright\ Small-Swap$
    $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} \leftarrow s_{0,1}, s_{0,0}, s_{0,3}, s_{0,2}$
  Else if $r \bmod 4 = 2$, ***then*** $\triangleright\ Big-Swap$
    $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3} \leftarrow s_{0,2}, s_{0,3}, s_{0,0}, s_{0,1}$
  End if

  If $r \bmod 4 = 0$, ***then*** $\triangleright\ Add\ constant$
    $s_{0,0} = s_{0,0} \oplus 0x9e377900 \oplus r$
  End if
End for
Return $(s_{i,j})$

Algoritme 2 *Absorb*

INPUT: $s = (s_{i,j}) \in \mathcal{W}^{3\times4}, m \in \mathbb{F}_{256}^{16}$
OUTPUT: $Absorb(s,m) = (s_{i,j}) \in \mathcal{W}^{3\times4}$
For j dari 0 sampai 3, ***do***
  $s_{0,i} \leftarrow s_{0,i} \oplus \boldsymbol{touint32}(m_{4i}, \ldots, m_{4(i+1)})$
End for
$s \leftarrow GIMLI(s)$
Return s

Algoritme 3 *Squeeze*

INPUT: $s = (s_{i,j}) \in \mathcal{W}^{3\times4}$
OUTPUT: $Squeeze(s) = h \in \mathbb{F}_{256}^{16}$
$h \leftarrow \boldsymbol{tobytes}(s_{0,0}) \| \boldsymbol{tobytes}(s_{0,1}) \| \boldsymbol{tobytes}(s_{0,2}) \| \boldsymbol{tobytes}(s_{0,3})$

Return h

Algoritme 4 Fungsi *Hash* GIMLI (GIMLI_*HASH*)

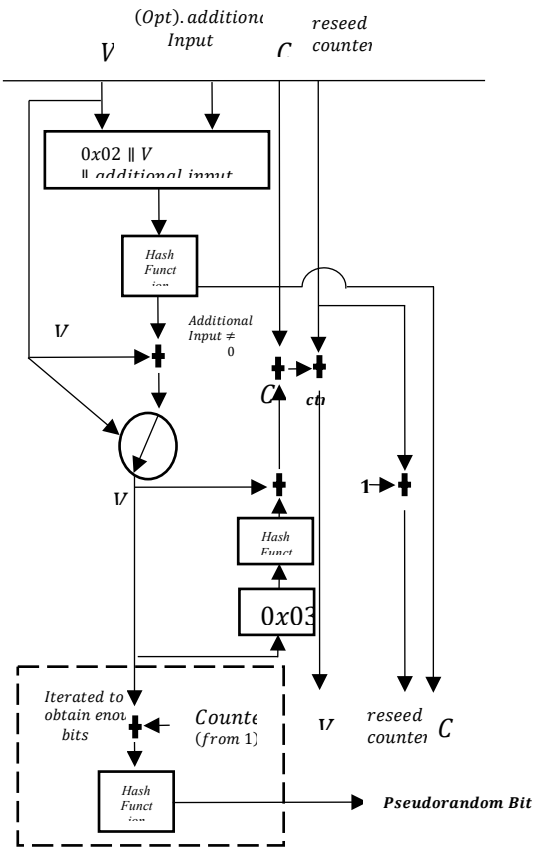
INPUT: $M \in \{0,1\}^*$
OUTPUT: $GIMLI\_HASH(m) = h \in \{0,1\}^{256}$
$s \leftarrow 0$
$m_1, \ldots\ldots, m_t \leftarrow \boldsymbol{pad}(M)$
For i dari 1 sampai t, ***do***
  If $i = t$, then
    $s_{2,3} \leftarrow s_{2,3} \oplus 0x01000000$
End if
  $s \leftarrow Absorb(s, m_i)$
End for
$h \leftarrow Squeeze(s)$
$s = GIMLI(s)$
$h \leftarrow h \| Squeeze(s)$
Return h

2.3 Desain PRNG Berbasis Fungsi *Hash* Berdasarkan NIST SP 800-90A Revisi 1

Bit-bit acak merupakan komponen penting dalam kriptografi yang digunakan dalam berbagai proses, meliputi proses pembangkitan kunci dan enkripsi. Rangkaian bit yang digunakan dalam kriptografi merupakan bit acak yang memenuhi kriteria sebagai berikut [3]:

a. Berdistribusi seragam, barisan bit-bit *string* dikatakan memiliki distribusi seragam apabila frekuensi terjadinya bit satu dan nol adalah sama.
b. Independen yaitu tidak ada subbarisan dalam barisan bit-bit *string* yang dipengaruhi oleh barisan lain.

Pseudorandom Number Generator (PRNG) merupakan skema penghasil bit-bit acak yang dapat digunakan sebagai kunci untuk algoritme kriptografi. Berdasarkan NIST SP 800-90A revisi 1 suatu PRNG dapat dibangun dengan menggunakan fungsi *hash* dan *block cipher* dengan masing-masing skema terdiri dari empat fungsi yaitu fungsi pembangkit, fungsi *reseed,* fungsi *instantiate*, dan fungsi *uninstantiate.* Setiap fungsi dari skema tersebut menghasilkan tiga buah nilai sama yaitu *initial state* dan nilai *constant* yang merupakan nilai yang bergantung pada *initial state* serta *reseed_counter*. Skema *Hash*_DRBG merupakan PRNG berbasis fungsi *hash* yang direkomendasikan dalam NIST SP 800-90A revisi 1. Skema tersebut telah direkomendasikan bersamaan dengan fungsi *hash* yang dapat dipakai sebagai fungsi pembangkitannya, meliputi SHA-1, SHA-224, SHA-256, SHA-384, dan SHA-512. Skema *Hash*_DRBG memperoleh masukan yang berasal dari masukan acak yang *trully random* maupun *pseudorandom* agar menghasilkan keluaran yang acak. Pada penelitian ini dilakukan desain PRNG berdasarkan konstruksi yang direkomendasikan oleh NIST SP 800-90A revisi 1 berbasis fungsi *hash* dengan skema dijelaskan pada Gambar 4.



Gambar 4. Skema *Hash*_DRBG

2.4 *A Statistical Test Suite For Random And Pseudorandom Number Generator For Cryptographic Applications* (NIST SP 800-22 Revisi 1A)

NIST SP 800-22 revisi 1a merupakan sekumpulan tes uji statistik bit-bit keluaran *random number generator* (RNG) dan *pseudorandom number generator* (PRNG). Uji ini terdiri dari 15 uji dengan parameter level signifikan (α) yang dapat dipilih adalah 0,001 – 0,1. Keluaran RNG dan PRNG yang diuji diharapkan memenuhi kriteria sebagai berikut [2]:

a. Keseragaman, yaitu keluaran RNG maupun PRNG yang diuji memiliki frekuensi kejadian munculnya bit 0 dan bit 1 adalah sama.
b. Skalabilitas, yaitu tes keacakan yang diterapkan pada rangkaian juga dapat diterapkan pada subrangkaian uji hasil keluaran keluaran RNG maupun PRNG. Jika hasil pengujian pada rangkaian uji adalah acak, maka hasil pengujian pada subrangkaian uji juga acak.
c. Konsistensi, yaitu RNG maupun PRNG menghasilkan keluaran yang acak meskipun dengan masukan yang berbeda-beda.

2.4.1 Interpretasi Hasil Uji

Hasil pengujian dapat diinterpretasikan dengan dua pendekatan, yaitu pendekatan proporsi dan pendekatan keseragaman distribusi $p - value$.

a. Pendekatan Proporsi
Daerah penerimaan pada pendekatan proporsi ditentukan dengan persamaan berikut:
$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$, dengan $\hat{p} = 1 - \alpha$ dan m adalah ukuran sampel uji. Apabila proporsi dari barisan uji yang diuji berada di luar daerah penerimaan, maka barisan uji dinyatakan tidak lulus sehingga barisan uji tidak acak.
b. Pendekatan Keseragaman Distribusi $p - value$
Distribusi $p - value$ diuji untuk menjamin terpenuhinya sifat keseragaman, interval 0 sampai 1 dibagi ke dalam 10 *sub-interval* dan $p - value$ dari barisan uji dikelompokkan ke dalam 10 *sub-interval* tersebut. Hasil pengelompokkan kemudian diuji keseragamannya menggunakan uji χ^2 dengan persamaan berikut.

$\chi^2 = \sum_{i=1}^{10} \frac{\left(F_i - \frac{m}{10}\right)^2}{\frac{m}{10}}$, dengan F_i merupakan banyak $p - value$ dalam interval i dan s adalah ukuran sampel. $P - value$ hasil observasi ($p - value_\tau$) dari hasil statistik uji dihitung dengan persamaan $igamc\left(\frac{9}{2}, \frac{x^2}{2}\right)$. Jika nilai $p - value_\tau \geq 0{,}0001$, maka rangkaian dinyatakan berdistribusi seragam.

2.4.2 Uji Keacakan NIST

a. *Frequency (Monobits) Test*
Frequency (Monobit) Test merupakan uji yang bertujuan untuk mengetahui apakah jumlah bit 0 dan bit 1 dalam suatu barisan kurang lebih sama seperti yang diharapkan untuk barisan yang acak, yaitu jumlah bit 0 dan 1 sama.
b. *Frequency Test within a Block*
Frequency Test within a Block merupakan uji yang bertujuan untuk mengetahui apakah frekuensi bit 1 dalam M-bit blok uji mendekati nilai M dibagi 2. Rangkaian bit dinyatakan lulus uji apabila jumlah bit 1 dalam blok uji mendekati nilai M dibagi 2.
c. *Runs Test*
Runs Test merupakan uji yang bertujuan untuk mengetahui apakah jumlah *run* atau barisan bit yang sama, bisa bernilai bit 1 semua atau 0 semua dalam suatu barisan dengan panjang bervariasi yang diharapkan seperti pada rangkaian acak. Hal ini untuk menentukan apakah osilasi antara kemunculan bit 0 dan 1 terlalu cepat atau terlalu lambat. Untuk melakukan *runs test* harus dilakukan *frequency(monobit) test* pada barisan yang akan diuji terlebih dahulu.
d. *Test for the Longest Run of Ones in a Block*
Test for the Longest Run of Ones in a Block merupakan uji yang bertujuan untuk mengetahui apakah panjang *run* 1 terpanjang dalam barisan yang diuji konsisten dengan panjang *run* 1 terpanjang yang diharapkan dalam barisan acak.
e. *Binary Matrix Rank Test*

Binary Matrix Rank Test merupakan uji yang memeriksa *rank* dari pemisahan *submatriks* seluruh barisan. Uji ini digunakan untuk memeriksa kebebasan linier antar *substring* dengan panjang tetap pada barisan awal.

f. *Discrete Fourier Transform (Spectral) Test*
Discrete Fourier Transform (Spectral) Test merupakan uji yang bertujuan untuk mendeteksi ciri periodik atau pola berulang yang letaknya saling berdekatan satu sama lain dalam barisan uji.

g. *Non-Overlapping Template Matching Test*
Non-Overlapping Template Matching Test merupakan uji yang berfokus pada jumlah kejadian target *string* yang telah dispesifikasi sebelumnya. Uji ini bertujuan untuk mendeteksi *generator* yang menghasilkan terlalu banyak kejadian bit 0 atau bit 1.

h. *Overlapping Template Matching Test*
Overlapping Template Matching Test merupakan uji yang hampir sama dengan *non overlaping template matching test,* keduanya menggunakan m-bit *window* untuk mencari pola $m-$bit tertentu. Perbedaan uji ini dengan uji pada poin g yaitu ketika pola ditemukan, *window* bergeser hanya satu bit sebelum pola berakhir.

i. *Overlapping Template Matching Test*
Maurer's "Universal Statistical" Test merupakan uji yang bertujuan untuk mendeteksi apakah barisan dapat dikompres secara signifikan tanpa kehilangan informasi. Barisan yang dapat dikompres secara signifikan dianggap tidak acak.

j. *Overlapping Template Matching Test*
Overlapping Template Matching Test merupakan uji bertujuan untuk mengetahui apakah barisan cukup kompleks untuk dianggap acak. Barisan acak dikarakteristikkan dengan panjang LFSR. LFSR yang pendek dinyatakan tidak acak.

k. *Serial Test*
Serial Test merupakan uji yang digunakan untuk mengetahui apakah jumlah kejadian dari 2^m m-bit pola *overlapping* mendekati seperti yang diharapkan untuk barisan acak.

l. *Approximate Entropy Test*
Approximate Entropy Test merupakan uji yang bertujuan untuk membandingkan frekuensi blok *overlapping* pada dua panjang yang berdekatan yaitu blok m dan blok $m+1$ terhadap hasil yang diharapkan untuk barisan acak.

m. *Cumulative Sums (Cusum) Test*
Cumulative Sums (Cusum) Test merupakan uji yang berfokus pada kemunculan maksimal bit 0 dari suatu *random walk* yang didefinisikan dengan jumlah *cumulative* yang disesuaikan dari digit -1, +1 pada barisan. Uji ini bertujuan untuk menentukan apakah *cumulative sum* yang terjadi pada barisan parsial yang diuji terlalu besar atau terlalu kecil sesuai dengan *comulative sum* untuk barisan acak.

n. *Random Excursions Test*
Random Excursions Test merupakan uji yang digunakan untuk mengetahui jumlah *visit* pada *state* dalam suatu *cycle* (siklus) yang diharapkan suatu barisan acak.

o. *Random Excursions Varian Test*
Random Excursions Varian Test merupakan uji yang digunakan untuk mengetahui deviasi dari jumlah *visit* yang diharapkan dari berbagai *state* dalam *random walk.*

3. METODE PENELITIAN

3.1 Metodologi Penelitian

Metode penelitian yang digunakan dalam penelitian ini adalah metode kepustakaan dan eksperimen. Metode kepustakaan dilakukan dengan mengumpulkan, mempelajari, dan memahami teori-teori terkait penelitian dengan sumber yang digunakan adalah buku, *paper*, tugas akhir, jurnal, standar NIST, maupun sumber lainnya. Metode eksperimen dilakukan dengan melakukan implementasi desain dalam bahasa pemrograman Python dan pengujian keacakan terhadap keluaran PRNG menggunakan uji yang direkomendasikan dalam NIST SP 800-22 revisi 1a.

3.2 Tahapan Penelitian

Tahapan penelitian yang dilakukan dalam penelitian ini antara lain:

a. Kajian kepustakaan yang dilakukan meliputi teori-teori yang berhubungan dengan konsep PRNG, permutasi GIMLI, desain PRNG berdasarkan NIST SP 800-90A revisi 1, uji yang direkomendasikan pada NIST SP 800-22 revisi 1a. Kajian tersebut dilakukan dengan mengumpulkan, mempelajari, dan memahami teori-teori terkait penelitian yang bersumber pada buku, *paper*, tugas akhir, standar NIST, serta sumber lain yang mendukung.

b. Pembuatan desain PRNG berbasis fungsi *hash* berdasarkan standar skema dalam NIST SP 800-90A revisi 1 menggunakan GIMLI_*HASH* sebagai fungsi pembangkit rangkaian bit. Desain yang dihasilkan diimplementasikan dalam bahasa pemrograman Python.

c. Pembangkitan 1000 *seed* dengan panjang 440 bit yang berasal dari algoritme Mersenne Twister menggunakan fungsi *random* pada bahasa pemrograman Python.

d. Pembangkitan sampel uji dengan ketentuan sebagai berikut:
   1) Pembangkitan 1000 sampel uji yang diperoleh dari keluaran PRNG dengan masukan *seed* yang telah dibangkitkan pada poin c, parameter *additional_input*, *constant*, dan *reseed _counter* masing-masing adalah *null*, nilai tergantung pada *seed*, dan satu.

2) Pembangkitan 1000 sampel uji yang diperoleh dari keluaran PRNG dengan masukan *seed* yang telah dibangkitkan pada poin c, parameter *additional_input*, *constant*, dan *reseed _counter* masing-masing adalah nilai acak dengan panjang 64 bit, nilai tergantung pada *seed*, dan satu.

e. Pengujian menggunakan 15 uji keacakan yang direkomendasikan pada NIST SP 800-22 revisi 1a dengan *tools* sts-2.1.2. Pengujian dilakukan dengan dua skenario, yaitu pengujian terhadap dua sampel yang telah dijelaskan pada poin d. Pengujian seperti dijelaskan pada poin d dilakukan untuk mengetahui pengaruh perubahan nilai *additional_input* terhadap keluaran PRNG.

f. Penarikan kesimpulan dari hasil pengujian keacakan rangkaian bit keluaran PRNG menggunakan pendekatan proporsi yang lulus uji. Hipotesis yang digunakan dalam uji yaitu:
H_0 : proporsi rangkaian uji diterima,
H_1 : proporsi rangkaian uji tidak diterima.
Proporsi diterima (H_0) apabila banyak rangkaian yang lulus uji $(p - value \geq \alpha)$ masuk dalam rentang $\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$ dengan $\hat{p} = 1 - \alpha$, dan m merupakan ukuran sampel. Apabila, banyak rangkaian lulus uji tidak memenuhi, maka H_0 ditolak.

4. DESAIN PRNG BERBASIS FUNGSI *HASH* MENGGUNAKAN GIMLI

4.1 Skema PRNG Berbasis Fungsi *Hash* Menggunakan GIMLI

Berdasarkan NIST SP 800-90A revisi 1, suatu PRNG yang digunakan untuk menghasilkan bit-bit acak dapat dibangun dengan menggunakan *block cipher* dan fungsi *hash* yang disebut *Hash*_DRBG. Skema PRNG pada penelitian ini seperti dijelaskan pada Gambar 4.1 memiliki empat masukan yang terdiri dari tiga masukan utama yaitu *seed* (V), *constant* (C), *reseed _counter*, dan satu masukan yang bersifat *optional* yaitu *additional_input*. *Additional_input* merupakan masukan yang digunakan untuk melakukan pembaruan nilai *seed* sebelum masuk fungsi pembangkit. Skema PRNG pada penelitian ini merupakan skema berbasis fungsi *hash* yang menggunakan GIMLI_*HASH* sebagai fungsi pembangkit bit-bit acak dan menghasilkan empat keluaran yaitu *pseudorandom* bit, *seed* baru (V), *constant* (C), dan *reseed _counter*. Skema PRNG berbasis fungsi *hash* yang digunakan merupakan fungsi pembangkit dengan *seed* yang diperoleh dari Algoritme Mersenne Twister. Algoritme 5 merupakan penjelasan dari fungsi pembangkit.

Algoritme *hashgen* dan Algoritme *Hash_df* merupakan proses yang digunakan dalam fungsi pembangkit pada PRNG. Algoritme *hashgen* adalah fungsi yang digunakan untuk menghasilkan bit-bit acak dengan dua parameter masukan yaitu nilai V dan *request_number_of_bits*. Algoritme *hashgen* menggunakan GIMLI-*HASH* untuk menghasilkan keluaran rangkaian bit acak dengan masukan berupa nilai V. Operasi GIMLI-*HASH* terus dilakukan sampai didapatkan panjang keluaran bit-bit acak dengan panjang yang diinginkan sesuai parameter *request_number_of_bits*. Nilai V ditambah *counter* 1 setiap operasi GIMLI_*HASH*. *Hash-df* merupakan proses yang digunakan untuk menghasilkan nilai *constant* (C) menggunakan dua parameter masukan yaitu *input_string* $(0x00||V)$ dan *no_of_bits_to_return* (*seed len*). Nilai C yang dihasilkan tergantung pada nilai *seed* yang dipakai serta memiliki panjang yang sama dengan *seed.* Proses ini hanya dilakukan sekali, dan digunakan lagi apabila nilai *seed* yang dipakai diubah oleh *user* dan bukan nilai yang berasal dari pembaruan *seed* pada proses pembangkitan.

Algoritme 5 Fungsi Pembangkit

***INPUT*:**
$V, C, reseed\_counter, request\_number\_of\_bits, addit$
***OUTPUT*:**
$SUCCESS, returned\_bits, V, C, reseed\_counter$
If $reseed\_counter > reseed\_interval$, then
"**reseed** *needed*"
End if
If $additional\_input \neq Null$, **then**
$w = Hash(0x02||V||additional\_input)$
$V = (V + w) mod\ 2^{seedlen}$
End if
$Returned\_bits$
$= Hashgen(request\_number\_of\_bits, V)$
$H = Hash(0x03||V)$
$V = (V + H + C + reseed\_counter)\ mod\ 2^{seedlen}$
$reseed\_counter = reseed\_counter + 1$
Return
$(SUCCESS, returned\_bits, V, C, reseed\_counter)$

4.2 Implementasi Skema PRNG Berasis Fungsi *Hash* Menggunakan GIMLI

Implementasi skema PRNG dilakukan dengan program simulasi menggunakan bahasa pemrograman Python versi 3. Python dipilih karena dapat melakukan perhitungan bilangan besar. *Software Integrated Development Environment* (IDE) yang dipakai pada simulasi ini yaitu Pycharm 2019.3.

Simulasi dilakukan untuk mendapatkan rangkaian bit acak sepanjang satu juta bit sebanyak 1000 rangkaian. Implementasi program simulasi menghasilkan sebuah rangkaian bit sepanjang satu juta bit, oleh karena itu simulasi dilakukan sebanyak 1000 kali untuk mendapatkan jumlah yang telah ditentukan. Pembangkitan rangkaian bit acak dengan jumlah tersebut digunakan sebagai sampel dalam proses pengujian. *Seed* yang digunakan dalam program simulasi berasal dari algoritme Mersenne Twister yang diperoleh menggunakan fungsi *random*

pada bahasa pemrograman Python. Tabel 1 dan Tabel 2 merupakan contoh hasil simulasi skema PRNG berbasis fungsi *hash* menggunakan GIMLI dengan nilai masukan yang berbeda.

Tabel 1. Masukan PRNG

No	Masukan	Simulasi 1	Simulasi 2
1	*seed*	*0xe539c368cc21...e61d*	*0xe367167067e7...de9b*
2	*constant* (C)	*0x8c65fc46b201…7a46*	*0xe367167067e7...de9b*
3	*reseed_counter*	*1*	*1*
4	*additional_input*	*null*	*null*
5	*request_number_of_bits*	1.000.000 bit	1.000.000 bit

Tabel 2. Keluaran PRNG

No	Masukan	Simulasi 1	Simulasi 2
1	*returned_bits*	*0xdee99790b3…5b52*	*0x88f88176bf6...4a82*
2	*seed*	*0xe539c368c…b7fad*	*0xe367167067…14aa*
3	*constant* (C)	*0x8c65fc46b20…7a46*	*0x8c65fc46b20…7a46*
4	*reseed_counter*	*2*	*2*

5. PENGUJIAN DAN INTERPRETASI HASIL UJI

Pengujian terhadap desain PRNG berbasis fungsi hash menggunakan GIMLI dapat dilakukan dengan berbagai skenario uji berdasarkan kombinasi masukan pada PRNG. Dalam penelitian ini pengujian dilakukan dengan dua skenario uji. Pengujian pertama dilakukan terhadap sampel yang diperoleh dari *seed* yang dibangkitkan menggunakan algoritme Mersenne Twister dengan parameter masukan *additional_input* dan *reseed _counter* masing-masing adalah *null* dan satu. Pengujian kedua dilakukan terhadap sampel yang dihasilkan dari *seed* yang sama dengan pengujian pertama tetapi paremeter masukan *additional_input* dan *reseed _counter* masing-masing adalah nilai acak dan satu. Kedua pengujian dilakukan untuk mengetahui pengaruh dari nilai masukan *additional_input* terhadap keluaran PRNG yang dihasilkan. Analisis dilakukan dalam bentuk interpretasi terhadap sifat keacakan rangkaian bit pada setiap uji keacakan. Rangkaian bit keluaran PRNG dinyatakan lulus apabila semua sampel lulus dalam uji yang direkomendasikan dalam NIST SP 800-22 revisi 1a.

5.1 Pengujian PRNG Berbasis Fungsi Hash Menggunakan GIMLI

Tabel 3 menunjukan hasil uji terhadap dua *sample* yaitu bit-bit keluaran PRNG yang berasal dari *seed* yang dibangkitkan oleh algoritme Mersenne Twister dan bit-bit keluaran PRNG. Berdasarkan hasil uji yang ditunjukan, disimpulkan bahwa seluruh *sample* lulus pada semua semua uji NIST.

Tabel 3. Hasil Pengujian Seluruh *Sample*

No	Uji NIST	Panjang sub rangkaian	Keterangan
1	*Frequency Test*	–	Lulus
2	*Frequency within blocks Test*	128	Lulus
3	*Cumulative sums Test*	–	Lulus
4	*Runs Test*	–	Lulus
5	*Longest run within blocks Test*	–	Lulus
6	*Binary rank Test*	–	Lulus
7	*FFT Test*	–	Lulus
8	*Non Overlapping Templates Test*	9	Lulus
9	*Overlapping Templates Test*	9	Lulus
10	*Maurer's Universal Test*	–	Lulus
11	*Approximate entropy Test*	10	Lulus
12	*Random excursion Test*	$X = -4$	Lulus
		$X = -3$	
		$X = -2$	
		$X = -1$	
		$X = 1$	
		$X = 2$	
		$X = 3$	
		$X = 4$	
13	*Random excursion variant Test*	$X = -9$	Lulus
		$X = -8$	
		$X = -7$	
		$X = -6$	
		$X = -5$	
		$X = -4$	
		$X = -3$	
		$X = -2$	
		$X = -1$	
		$X = 1$	
		$X = 2$	
		$X = 3$	
		$X = 4$	
		$X = 5$	
		$X = 6$	
		$X = 7$	
		$X = 8$	
		$X = 9$	
14	*Serial Test*	16	Lulus
		16	
15	*Linear complexity Test*	500	Lulus

5.2 Interpretasi Hasil Uji

Berdasarkan hasil uji pada Subbab 5.1 terbukti bahwa PRNG berbasis fungsi *hash* menggunakan GIMLI menghasilkan keluaran yang acak menggunakan uji NIST SP 800-22 revisi 1a dengan

pendekatan proporsi rangkaian yang lulus uji. Berikut interpretasi hasil uji pada masing-masing pengujian:

a. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada *frequency (monobit) test* yang berarti jumlah bit 0 dan bit 1 pada kedua sampel kurang lebih sama seperti yang diharapkan untuk barisan yang acak yaitu jumlah bit 0 dan 1 sama.
b. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada *frequency test within a block* yang berarti frekuensi bit 1 dalam M-bit blok rangkaian uji mendekati M dibagi 2.
c. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada *Runs test* yang berarti jumlah *run* dalam rangkaian uji yang memiliki panjang yang bervariasi seperti yang diharapkan seperti pada rangkaian acak.
d. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada *test for the longest run of ones in a block* yang berarti panjang *run* 1 terpanjang dalam barisan yang diuji konsisten dengan panjang *run* 1 terpanjang yang diharapkan dalam barisan acak.
e. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada *binary matriks rank test* yang berarti *rank* submatriks saling lepas dari seluruh rangkaian.
f. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada *discrete fourier transform (spectral) test* yang berarti rangkaian barisan uji tidak memiliki ciri periodik yaitu pola berulang yang letaknya saling berdekatan satu sama lain.
g. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada *non-overlapping template matching test* yang berarti rangkaian barisan uji tidak menghasilkan terlalu banyak kejadian (0 atau 1) pola *non periodik (aperiodik)*. Jika pola tidak ditemukan, maka *window* akan bergeser satu bit dan jika pola ditemukan maka *window* akan me-reset bit setelah pola ditemukan.
h. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada *overlapping template matching test* yang berarti rangkaian barisan uji tidak menghasilkan terlalu banyak kejadian (0 atau 1) pola *non periodik* (aperiodik) yaitu sama dengan poin g. Namun, ketika pola ditemukan *window* bergeser hanya satu bit sebelum pola berakhir.
i. Pada kedua sampel pengujian dinyatakan lulus pada *maurer's "universal statistical" test* yang berarti rangkaian uji dapat dikompres secara signifikan tanpa kehilangan informasi.
j. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada *linear complexity test* yang berarti rangkaian barisan uji cukup kompleks untuk dianggap acak.
k. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada serial test yang berarti rangkaian barisan uji jumlah kejadian dari 2^m m-bit pola *overlapping* mendekati seperti yang diharapkan untuk barisan acak.
l. Pada kedua sampel pengujian dinyatakan lulus pada *approximate entropy test* yang berarti frekuensi blok *overlapping* pada dua panjang yang berdekatan sama terhadap hasil yang diharapkan untuk barisan acak.
m. Pada sampel pengujian 1 dan pengujian 2 dinyatakan lulus pada *cumulative sums test* yang berarti *cumulative sum* yang terjadi pada barisan parsial yang diuji tidak terlalu besar atau terlalu kecil sesuai dengan *cumulative sum* untuk barisan acak.
n. Pada kedua sampel pengujian dinyatakan lulus pada *random excursions test* yang berarti jumlah *visit* pada *state* dalam suatu putaran tidak menyimpang dari yang diharapkan suatu barisan acak.
o. Pada kedua sampel pengujian dinyatakan lulus pada *random excursions variant test* yang berarti deviasi dari jumlah *visit* sesuai dengan yang diharapkan dari berbagai *state* dalam *random walk.*

Berdasarkan hasil pengujian dengan kedua skenario diatas, terbukti bahwa hasil keluaran PRNG tersebut memenuhi seluruh parameter keacakan dalam pengujian meliputi keacakan *frequency* (*monobit*), frekuensi bit 1 dalam M-bit blok mendekati M dibagi 2, dan seterusnya.

6. KESIMPULAN

Berdasarkan desain dan pengujian dalam penelitian yang telah dilakukan, disimpulkan sebagai berikut:

a. Desain PRNG berbasis fungsi *hash* menggunakan GIMLI merupakan desain PRNG dengan menggunakan skema DRBG_*HASH* yang direkomendasikan pada NIST SP 800-90 A revisi 1. Pada penerapannya hasil desain skema PRNG menggunakan fungsi pembangkit GIMLI_*HASH,* serta *seed* yang berasal dari algoritme Mersenne Twister.
b. Keluaran PRNG berbasis fungsi *hash* menggunakan GIMLI dinyatakan acak berdasarkan uji keacakan yang direkomendasikan pada NIST SP 800-22 revisi 1 dengan pendekatan proporsi menggunakan dua skenario yang telah dijelaskan pada Subbab 3.2.

REFERENSI

[1] Wulandari, Desi. 2014. Analisis Efek Serangan Penyisipan Terhadap Keacakan Barisan Bit yang Dihasilkan Algoritme SOSEMANUK dan HC-128 Menggunakan Uji NIST. Skripsi Sekolah Tinggi Sandi Negara.

[2] Bassham III, L. E., Rukhin, A. L., Soto, J., Nechvatal, J. R., Smid, M. E., Barker, E. B., ... &

Heckert, N. A. (2010). Sp 800-22 revisi 1a. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*.

[3] Stallings, W. & Horton, M. J., 2017. *Cryptography and Network Security : Principles and Practice.* 7th ed. England: Pearson Education Limited.

[4] Barker, E. B., & Kelsey, J. M. (2015). *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*,(NIST 800-90A rev1). *Retrieved September*,*3*, 2016.

[5] Bernstein, D. J., Kölbl, S., Lucks, S., Massolino, P. M. C., Mendel, F., Nawaz, K., ... & Viguier, B. (2017, September). Gimli: *A Cross-platform Permutation*. In *International Conference on Cryptographic Hardware and Embedded Systems* (pp. 299-320). Springer, Cham.

[6] A. J. Menezes, P. C. V. Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*, USA: CRC Press, 1996.