Sistem Partikel Sebagai Sebuah Metode Kriptografi dan Steganografi Modern

Christian (13207033)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
qruz_tyan@students.itb.ac.id

***Abstrak*—Kriptografi sebagai sebuah metode penyimpanan data rahasia melalui rekombinasi komponen isi di dalam data tersebut sudah memasuki perkembangan yang melibatkan bukan saja kerumitan algoritma kriptografi itu sendiri, tetapi juga tingkat keacakan algoritma kriptografi itu sendiri yang akan menyulitkan serangan terhadap kriptografi di berbagai tingkat. Hal yang sama juga berlaku di dalam ruang linkup steganografi yang menjadi media penyimpanan data secara rahasia melalui penyisipan secara acak di dalam media lainnya. Pada dasarnya algoritma kriptografi dan steganografi tidak terpaku pada manipulasi proses enkripsi dan dekripsi saja atau pengacakan penyimpanan bit-bit data di dalam stego-file secara algoritmik, melainkan juga dapat diperluas ke banyak metode dan bidang fisis lainnya. Secara khusus di dalam makalah ini dipaparkan pemakaian metode sistem partikel di dalam bidang kriptografi dan steganografi yang akan menghasilkan suatu bentuk metode enkripsi dan dekripsi data yang lebih aman bila dibandingkan dengan pembangkitan bilangan acak yang sudah terintegrasi di dalam library dasar bahasa pemrograman yang digunakan oleh kriptografer.**

***Index Terms*—Kriptografi, Particle System, Smoothed Particles Hydrodynamics**

I. Latar Belakang

Kriptografi sebagai sebuah metode manipulasi data atau suatu *file* bertumpu pada kerumitan metode manipulasi data atau isi *file* tersebut terhadap struktur atau susunan aslinya. Demikian halnya dengan steganografi yang menitikberatkan kerahasiaan penyimpanan informasi di dalam media perantaranya melalui suatu tingkat keacakan penyimpanan informasi rahasia tersebut di dalam media perantara yang digunakannya. Kedua prinsip dasar tersebut berkembang seiring dengan kemajuan teknologi informasi yang ada di dunia saat ini yang sejalan juga dengan penelitian-penelitian yang ada di dalam bidang ini. Akan tetapi, tingkat keacakan yang tinggi di dalam sebuah metode enkripsi data yang ada di dalam algoritma-algoritma kriptografi modern saat ini bertumpu kuat pada struktur algoritma yang tidak kalah rumit serta panjang, tetapi bersifat terstruktur rapi dan tereatur. Hal ini menjadi salah satu kekurangan utama yang ada di dalam permasalahan tingkat kerumitan itu sendiri, yakni keteraturan yang dengan sangat jelas terlihat dari algoritma kriptografi itu sendiri. Demikian halnya dengan steganografi yang pada prinsipnya secara sederhana dapat dikerjakan dengan membangkitkan bilangan acak untuk penempatan setiap bit data informasi di dalam *stego-file* yang digunakan. Hal ini tentunya berbeda konteks dengan perihal jumlah kemungkinan kunci yang dapat digunakan untuk setiap algoritma yang telah ada tersebut tentunya karena penemuan kunci yang paling sesuai di dalam kriptanalisis tentunya akan semakin mengerucut seiring dengan pengerjaan metode-metode analisis yang dapat digunakan. Permasalahan yang penulis angkat merupakan perihal tingkat keacakan yang tidak teratur yang menjadi kunci keamanan informasi data di dalam metode enkripsi serta steganografi.

II. Rumusan Masalah

Di dalam topik ini, penulis membagi permasalahan yang ada ke dalam beberapa hal utama, yakni:

- Bagaimana model fisis dipergunakan di dalam algoritma kriptografi dan steganografi?
- Bagaimana perhitungan fisis dapat merepresentasikan keacakan suatu susunan di dalam perhitungan algoritma?
- Apa kelemahan dan kelebihan dari sistem partikel di dalam implementasi untuk kriptografi dan steganografi?

III. Metode Penelitian

Penyusunan makalah ini bertepatan dengan penulis tugas akhir penulis yang berdasar atas sistem partikel di dalam komputasi simulasi fluida sehingga penelitian secara khusus mengenai topik ini telah dilakukan dalam waktu yang lama sebelum mata kuliah ini diambil dan dalam topik sistem partikel ini secara khusus, penulis melakukan studi literatur mengenai sistem partikel itu sendiri dan metode-metode kriptografi serta steganografi yang umum digunakan secara modern saat ini.

IV. SUMBER DATA

Seluruh sumber yang penulis gunakan merupakan literatur yang ada mengenai sistem partikel, kriptografi, dan steganografi.

V. TEORI DASAR

A. Sistem Partikel

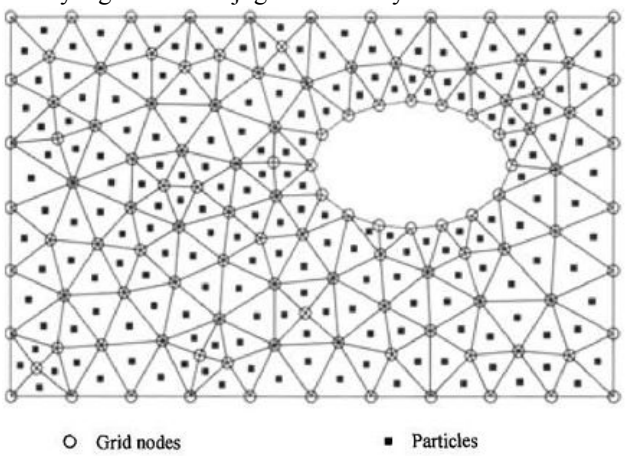
Sistem Partikel merupakan suatu metode pemodelan partikel secara makroskopik di dalam dunia komputasi yang melibatkan sifat-sifat tertentu dari partikel yang dimodelkan. Sangat banyak metode yang telah ada saat ini di dalam pemodelan sistem partikel, yakni Molecular Dynamics (MD), Monte Carlo (MC), Dissipative Particles Dynamics (DPD), Lattice Gas Cellular Automata (CA), Particle-in-Cell (PIC), Discrete Element Method (DEM), Smoothed Particles Hydrodynamics (SPH), Metode Vortex, dan lain-lain. Keseluruhan metode tersebut tergolong sebagai Meshfree Particles Method (MPM) yang merupakan kombinasi antara metode berbasis grid dan metode meshfree. Metode berbasis grid terbagi ke dalam dua macam metode, yakni metode Lagrange dan metode Euler.

Metode Lagrange merupakan metode pemodelan sistem partikel dengan menetapkan grid yang kaku atau tetap di dalam ruang sehingga seluruh partikel yang ada dimodelkan sebagai partikel yang bergerak dan berpindah dari satu grid ke grid yang lain dengan batasan ukuran di dalam setiap gridnya bagi partikel-partikel yang bersangkutan. Berbeda halnya dengan metode Euler yang melibatkan grid yang fleksibel terhadap partikel-partikel yang bergerak. Akan tetapi, pendeteksian posisi dan keadaan setiap partikel menjadi lebih mudah bila dilakukan di dalam metode Lagrange karena sifat grid yang kaku tersebut, sedangkan di dalam metode Euler, pendeteksian setiap partikel menjadi sulit karena melibatkan grid-grid yang bergerak menyesuaikan diri dengan kondisi setiap partikelnya. Pada dasarnya metode berbasis grid ini memiliki beberapa kelemahan signifikan yang pada akhirnya membuat sistem sulit dimodelkan bagi objek-objek yang memiliki bentuk tidak umum (*irregular*) sehingga dibutuhkan sebuah metode baru.

Tabel 5-1 Persamaan Kekekalan Massa, Momentum, dan Energi dalam Bentuk Diferensial Parsial Untuk Model Lagrange dan Euler

Konservasi	Deskripsi Lagrangian	Deskripsi Eulerian
Massa	$\frac{D\rho}{Dt} = -\rho \frac{\partial \boldsymbol{v}^{\beta}}{\partial \boldsymbol{x}^{\beta}}$	$\frac{D\rho}{Dt} + v^{\beta} \frac{\partial \rho}{\partial x^{\beta}} = -\rho \frac{\partial \boldsymbol{v}^{\beta}}{\partial \boldsymbol{x}^{\beta}}$
Momentum	$\frac{D\boldsymbol{v}^{\beta}}{Dt} = -\frac{1}{\rho} \frac{\partial p}{\partial \boldsymbol{x}^{\beta}}$	$\frac{D\boldsymbol{v}^{\beta}}{Dt} + v^{\alpha} \frac{\partial v^{\beta}}{\partial v^{\alpha}} = -\frac{1}{\rho} \frac{\partial p}{\partial \boldsymbol{x}^{\beta}}$
Energi	$\frac{De}{Dt} = -\frac{p}{\rho} \frac{\partial \boldsymbol{v}^{\beta}}{\partial \boldsymbol{x}^{\beta}}$	$\frac{De}{Dt} + v^{\beta} \frac{\partial e}{\partial x^{\beta}} = -\frac{p}{\rho} \frac{\partial \boldsymbol{v}^{\beta}}{\partial \boldsymbol{x}^{\beta}}$

Di dalam metode meshfree, tidak ada keterkaitan antara setiap partikel dalam hal pergerakannya di dalam ruang sehingga pemodelannya pun menjadi lebih mudah dan sederhana, terutama untuk representasi geometri yang rumit. Hal ini merupakan perkembangan lebih lanjut dari kondisi di dalam metode berbasis grid sebelumnya. Salah satu metode partikel berbasis meshfree ini yang sangat umum digunakan yaitu Smoothed Particles Hydrodynamics. Dengan sifat meshfree dan beberapa sifat berbasis grid yang utama, pendefinisian sistem menjadi lebih mudah karena setiap grid memiliki bentuk yang fleksibel dan terdefinisi sesuai dengan jumlah partikel yang terdefinisi juga di dalamnya.



Gambar 5-1 Meshfree Particles Method (MPM) Grid

Di dalam MPM, dilakukan pendekatan nilai fungsi, integral, dan derivatif suatu partikel dengan pendekatan partikel menggunakan informasi dari partikel-partikel tetangganya. Area pengaruh suatu partikel ditentukan oleh domain pengaruh atau support domain. Variabel lapangan yang mendefinisikan suatu partikel dapat didekati oleh:

$$u(x) = \sum_{i=1}^{N} \phi_i(\boldsymbol{x}) u_i$$

Dengan N adalah jumlah partikel di dalam support domain suatu partikel di posisi x. Variabel u_i merupakan variabel lapangan pada partikel i, dan ϕ_i merupakan fungsi bentuk pada partikel i yang dibentuk dengan menggunakan seluruh informasi partikel-partikel di dalam support domain partikel di posisi x tersebut.

B. Smoothed Particles Hydrodynamics (SPH)

Beberapa hal penting yang digunakan di dalam metode SPH yaitu:

- Domain masalah direpresentasikan dengan sekumpulan partikel yang terdistribusi secara acak jika domain tersebut belum terdapat di dalam bentuk partikel-partikel, dan tidak dibutuhkan hubungan antara partikel-partikel ini (*meshfree*);
- Metode representasi integral digunakan untuk pendekatan fungsi lapangannya yang disebut sebagai pendekatan Kernel;
- Pendekatan kernel kemudian didekati lebih lanjut dengan menggunakan partikel-partikel yang disebut sebagai pendekatan partikel dengan

mengganti integrasi di dalam representasi integral dari fungsi lapangannya dan derivatifnya dengan penjumlahan di seluruh nilai yang berkaitan pada partikel-partikel tetangganya di dalam sebuah domain lokal yang disebut sebagai support domain (*compact support*);

- Pendekatan partikel dibentuk pada setiap step waktu sehingga penggunaan partikel-partikelnya bergantung pada distribusi lokal saat itu dari seluruh partikelnya (*adaptive*);
- Pendekatan partikel ditunjukkan untuk seluruh variabel yang berhubungan dengan fungsi lapangan di dalam persamaan diferensial parsialnya untuk menghasilkan sejumlah persamaan diferensial umum di dalam bentuk diskrit terhadap waktu (*Lagrangian*);
- Persamaan diferensial umum diselesaikan dengan menggunakan algoritma integrasi yang eksplisit untuk mencapai waktu step yang cepat dan untuk memperoleh catatan waktu dari seluruh variabel lapangan untuk semua partikelnya (dinamis);

C. Kriptografi

Kriptografi merupakan sebuah seni atau metode merahasiakan suatu data dengan melakukan operasi-operasi tertentu pada data yang digunakan sehingga dihasilkan data dengan ukuran yang sama atau dapat berbeda dan memiliki nilai data yang berbeda dengan data semula mengacu pada suatu kunci tertentu untuk memecahkan data asli tersebut. Di dalam kriptografi, data hasil enkripsi disebut sebagai ciphertext yang merupakan fungsi dari plaintext (C = E(P)), sedangkan plaintext itu sendiri merupakan fungsi dekripsi dari ciphertext yang dihasilkan melalui proses enkripsi awal (P = D(C)). Kunci yang digunakan pada enkripsi dan dekripsi dapat bersifat simetri atau asimetri. Bila suatu proses enkripsi dan dekripsi menggunakan kunci simetri maka pada kedua proses tersebut digunakan kunci yang sama. Hal yang sebaliknya berlaku untuk penggunaan kunci asimetri.

D. Steganografi

Steganografi merupakan seni tentang penyisipan pesan rahasia di dalam suatu file atau media lainnya yang membuat pesan asli yang disisipkan harus terlebih dahulu diekstrasi oleh pihak penerima pesan untuk dapat membaca pesan yang dikirimkan melalui media lain tersebut. Metode penyisipan yang umum digunakan yaitu melalui pembangkitan bilangan acak sebagai posisi karakter atau bit-bit pesan yang akan disisipkan di dalam file stego tersebut secara berurutan dari pesan aslinya. Metode steganografi yang menggunakan kunci secara umum melibatkan kunci simetri sehingga di dalam penyisipan pesan dan ekstraksi pesan digunakan kunci yang sama.

VI. Perancangan dan Implementasi

A. Penggunaan Metode SPH dalam Kriptografi

Metode SPH pada dasarnya menjelaskan sifat partikel yang digunakan di dalam sistem partikel sesuai dengan persamaan kekekalan massa, momentum, dan energi yang telah disebutkan di atas. Ide utama yang digunakan di dalam metode ini dalam kaitannya dengan algoritma enkripsi dan dekripsi data yaitu tumbukan partikel yang dapat terjadi sebagai interaksi antara setiap partikel di dalam sistem yang didefinisikan di dalam program yang dibuat. Ukuran sistem melibatkan parameter jumlah grid, jumlah partikel maksimum sebagai "tetangga" yang berada di dalam satu grid yang sama, jumlah partikel total di dalam sistem, massa partikel, volume partikel, dan kecepatan partikel di dalam sistem. Kondisi yang dapat dibuat di dalam sistem yaitu partikel-partikel secara perlahan dimunculkan dari satu sumber atau beberapa sumber atau dapat juga berada di dalam kondisi telah ada secara acak di dalam sistem atau ruang sistem tersebut dalam gerakan acak juga. Intinya, keseluruhan sifat partikel tersebut dapat dimanipulasi sesuai dengan karakteristik sistem yang diinginkan dan dalam hal ini digunakan setiap kondisi tersebut dibatasi untuk bersifat statis, yakni tidak berubah terhadap waktu secara makro.

Sesuai dengan sifat tersebut di atas dan beberapa pengkondisian lainnya, dapat didefinisikan pengkodean sistem secara pemrograman sebagai berikut.

```
#include <memory.h>
#include <math.h>

#include <windows.h>
#include <assert.h>

#include "cpu_sph.h"
#include "sph_common.h"
#include "config.h"

extern char collector_mode;
extern float collector_bottom;
extern float collector_top;
extern float collector_right;
extern float collector_left;
extern float collector_back;
extern float collector_front;

extern float blood_mass;
extern float sphere_radius;

#define POS(i) cpu->pos[(i)]
#define VEL(i) cpu->vel[(i)]
#define ACC(i) cpu->acc[(i)]
#define VELH(i) cpu->vel_half[(i)]

void create_cpu(CPU* cpu)
{

  create_grid(&cpu->grid);

  // Precompute kernel coefficients
  cpu->poly6_coef = 315.0f/(64.0f*PI*(float)pow(H, 9));
       // W(r,h) = coef * ...
  cpu->grad_poly6_coef = 945.0f/(32.0f*PI*(float)pow(H,
9));   // gradW(r,h) =
  cpu->lap_poly6_coef = 945.0f/(32.0f*PI*(float)pow(H,
9));   // lapW
  cpu->grad_spiky_coef = -45.0f/(PI*(float)pow(H, 6));
```

```
}

void cpu_sph_compute_density(CPU* cpu, int first)
{
  int i, j;
  static float h2 = H * H;
  NEIGHBOUR_LIST* nlist = &cpu->n_list;
  //reset density
  memset( cpu->density, 0, PARTICLES_N *
sizeof(float));

  for (i = 0; i < PARTICLES_N; i++)
  {
    if (!cpu->active[i])
      continue;
    for (j = 0; j < nlist->sizes[i]; j++)
    {
      float distsq;
      int nindex = nlist->neighbours[i][j].index;
      if (first)
        distsq = nlist->neighbours[i][j].distsq;
      else
      {
        distsq = vec3_distsq( &POS(i), &POS(nindex));
        nlist->neighbours[i][j].distsq = distsq;
      }

      if (distsq < h2)
      {
        float h2_r2 = h2 - distsq;
        //density = sph-
>mass[nindex]*h2_r2*h2_r2*h2_r2;
        float density = h2_r2 * h2_r2 * h2_r2;
        //sph->density[i] += sph-
>mass[nindex]*density;
        float md = blood_mass * density;
        cpu->density[i] += md;
        if (i != nindex)
          cpu->density[nindex] += md;
      }
    }
  }
  for (i = 0; i < PARTICLES_N; i++)
  {
    if (!cpu->active[i])
      continue;

    cpu->density[i] *= cpu->poly6_coef;
    cpu->pressure[i] = BLOOD_STIFF * ( cpu->density[i]
- BLOOD_DENSITY );
    cpu->density[i] = 1.0f / cpu->density[i];
  }
}

void cpu_sph_compute_force(CPU* cpu)
{
  int i;
  int j;
  vector3 force; //[N/kg^2]
  extern float blood_viscosity;
  NEIGHBOUR_LIST* nlist = &cpu->n_list;
  //reset accelaration
  memset( cpu->acc, 0, PARTICLES_N * sizeof(vector3));
  for (i = 0; i < PARTICLES_N; i++)
  {
    if (!cpu->active[i])
      continue;

    for (j = 1; j < nlist->sizes[i]; j++)
    {
      float r = (float) sqrt( nlist-
>neighbours[i][j].distsq);//r);
      int nindex = nlist->neighbours[i][j].index;

      if (r < H)
      {
        float h_r;
        float scale;
        vector3 diff;
        //PRESSURE FORCE
        diff.x = cpu->pos[i].x - cpu->pos[nindex].x;
        diff.y = cpu->pos[i].y - cpu->pos[nindex].y;
        diff.z = cpu->pos[i].z - cpu->pos[nindex].z;
        scale = -0.5f * ( cpu->pressure[i] + cpu-
>pressure[nindex] ) * cpu->grad_spiky_coef * h_r / r;
        force.x = scale * diff.x;
        force.y = scale * diff.y;
        force.z = scale * diff.z;
        // VISCOSITY FORCE
        diff.x = cpu->vel[nindex].x - cpu->vel[i].x;
        diff.y = cpu->vel[nindex].y - cpu->vel[i].y;
        diff.z = cpu->vel[nindex].z - cpu->vel[i].z;
        diff.x *= ( blood_viscosity * cpu-
>lap_vis_coef);
        diff.y *= ( blood_viscosity * cpu-
>lap_vis_coef);
        diff.z *= ( blood_viscosity * cpu-
>lap_vis_coef);
        force.x += diff.x;
        force.y += diff.y;
        force.z += diff.z;
        force.x *= ( h_r * cpu->density[i] * cpu-
>density[nindex]);
        force.y *= ( h_r * cpu->density[i] * cpu-
>density[nindex]);
        force.z *= ( h_r * cpu->density[i] * cpu-
>density[nindex]);

        // Apply force
        cpu->acc[i].x += ( blood_mass * force.x);
        cpu->acc[i].y += ( blood_mass * force.y);
        cpu->acc[i].z += ( blood_mass * force.z);
        cpu->acc[nindex].x -= ( blood_mass * force.x);
        cpu->acc[nindex].y -= ( blood_mass * force.y);
        cpu->acc[nindex].z -= ( blood_mass * force.z);
      }
    }
  }
}

_inline void compute_collision(vector3* pos, vector3*
acc, const vector3* vel, const vector3* n, float diff,
float stiff, float damp)
{
  float v0 = vec3_dot(n, vel);
  float reverse;// = stiff * diff - damp * v0;
#ifdef PUSHOUT
  vec3_scaleadd(p, p, diff, n);
#endif
#ifdef REFLECT_VEL
  vec3_scaleadd(col, col, -1.9*v0, n);
#else
  reverse = stiff * diff - damp * v0;
  vec3_scaleadd( acc, acc, reverse, n);
#endif
}

void cpu_sph_process_collision(CPU* cpu)
{
  int i;
  float diff;
  vector3 n;

  for (i = 0; i < PARTICLES_N; i++)
  {
    _ALIGNED vector3 pnext; /** Predicted pos **/
    vector3 acc = { 0.0f, 0.0f, 0.0f,};


    if (!cpu->active[i])
      continue;
    pnext.x = cpu->pos[i].x + cpu->timestep * cpu-
>vel_half[i].x;
    pnext.y = cpu->pos[i].y + cpu->timestep * cpu-
>vel_half[i].y;
    pnext.z = cpu->pos[i].z + cpu->timestep * cpu-
>vel_half[i].z;
    if(collector_mode)
    {
      //bottom
      diff = 2.0f*sphere_radius - (pnext.y -
collector_bottom);
      if ( (diff > EPSILON) &&
(pnext.x>collector_left) && (pnext.x<collector_right)
&& (pnext.z>collector_back) && (pnext.z<collector_front
) )
      {
        vec3_set(&n, 0.0f, 1.0f, 0.0f);
        compute_collision(&pnext, &acc, &cpu->vel[i],
&n, diff, COLLECTOR_STIFF, COLLECTOR_DAMP);
        goto label1;
```

```
      }

      //top
      diff = 2.0f*sphere_radius - ( collector_top -
pnext.y);
      if ( (diff > EPSILON) &&
(pnext.x>collector_left) && (pnext.x<collector_right)
&& (pnext.z>collector_back) && (pnext.z<collector_front
) )
      {
        vec3_set(&n, 0.0f, -1.0f, 0.0f);
        compute_collision(&pnext, &acc, &cpu->vel[i],
&n, diff, COLLECTOR_STIFF, COLLECTOR_DAMP);
        goto label1;
      }

      //right
      diff = 2.0f*sphere_radius - (collector_right -
pnext.x);
      if ( (diff > EPSILON) &&
(pnext.z>collector_back) && (pnext.z<collector_front )
)
      {
        vec3_set(&n, -1.0f, 0.0f, 0.0f);
        compute_collision(&pnext, &acc, &cpu->vel[i],
&n, diff, COLLECTOR_STIFF, COLLECTOR_DAMP);
        goto label1;
      }

      //back
      diff = 2.0f*sphere_radius - (  pnext.z -
collector_back);
      if ( (diff > EPSILON) &&
(pnext.x>collector_left) && (pnext.x<collector_right))
      {
        vec3_set(&n, 0.0f, 0.0f, 1.0f);
        compute_collision(&pnext, &acc, &cpu->vel[i],
&n, diff, COLLECTOR_STIFF, COLLECTOR_DAMP);
        goto label1;
      }

      //front
      diff = 2.0f*sphere_radius - ( collector_front -
pnext.z);
      if ( (diff > EPSILON) &&
(pnext.x>collector_left) && (pnext.x<collector_right))
      {
        vec3_set(&n, 0.0f, 0.0f, -1.0f);
        compute_collision(&pnext, &acc, &cpu->vel[i],
&n, diff, COLLECTOR_STIFF, COLLECTOR_DAMP);
        goto label1;
      }
    }
label1:
    cpu->acc[i].x += acc.x;
    cpu->acc[i].y += acc.y;
    cpu->acc[i].z += acc.z;
  }
}

void cpu_grid_clear(CPU* cpu, int start, int end)
{
  float xmin;
  float xmax;
  float ymin;
  float ymax;
  float zmin;
  float zmax;
  int i, ngrid;
  const vector3* p;
  static const int _MAXGRID_ = GRID_NMAX;
  GRID* grid = &cpu->grid;
  xmin = ymin = zmin = +MAX_FLOAT;
  xmax = ymax = zmax = -MAX_FLOAT;

  for (i = start; i <= end; i++)
  {
    if (!cpu->active[i])
      continue;
    p = &cpu->pos[i];
    if (xmin > p->x)
      xmin = p->x;
    if (xmax < p->x)
      xmax = p->x;
    if (ymin > p->y)
      ymin = p->y;
    if (ymax < p->y)
      ymax = p->y;

    if (zmin > p->z)
      zmin = p->z;
    if (zmax < p->z)
      zmax = p->z;
  }
  grid->xmin = xmin;
  grid->ymin = ymin;
  grid->zmin = zmin;
  grid->nx = (int)( (xmax - xmin + grid->WIDTH) / grid-
>WIDTH);
  grid->ny = (int)( (ymax - ymin + grid->WIDTH) / grid-
>WIDTH);
  grid->nz = (int)( (zmax - zmin + grid->WIDTH) / grid-
>WIDTH);
  ngrid = grid->nx * grid->ny * grid->nz ;
  memset( grid->sizes, 0, _MAXGRID_ * sizeof(int));
  memset( grid->caps , 0, _MAXGRID_ * sizeof(int));
}
void cpu_sph_elapse(CPU* cpu, float dt)
{
  int i, j;
  static _ALIGNED vector3 gravity = {0.0f, -9.8f,
0.0f};

  cpu->timestep = dt;

  for (j = 0; j < LOOPS_MAX; j++)
  {
    if (j == 0)
    {
      sph_grid_clear(cpu->active, &cpu->grid, cpu-
>pos, 0, PARTICLES_N - 1);
      sph_grid_get_neighbours(cpu->active, &cpu->grid,
cpu->pos, 0, PARTICLES_N - 1, &cpu->n_list);
    }


    cpu_sph_compute_density( cpu, (j == 0)); //density
& pressure
    cpu_sph_compute_force( cpu);
    cpu_sph_process_collision( cpu);

    for ( i = 0; i < PARTICLES_N; i++)
    {
      _ALIGNED vector3 v_half;
      _ALIGNED vector3 final_acc;
      if (!cpu->active[i])
        continue;

      // ai = ai + g
      final_acc.x = cpu->acc[i].x + gravity.x;
   //GRAVITY_X;
      final_acc.y = cpu->acc[i].y + gravity.y;
//GRAVITY_Y;
      final_acc.z = cpu->acc[i].z + gravity.z;
//GRAVITY_Z;
      //vi+1/2 = vi-1/2 + t * ai
      v_half.x = cpu->vel_half[i].x + final_acc.x *
dt;
      v_half.y = cpu->vel_half[i].y + final_acc.y *
dt;
      v_half.z = cpu->vel_half[i].z + final_acc.z *
dt;
      // xi+1 = xi + t * vi+1/2
      cpu->pos[i].x += dt * v_half.x;
      cpu->pos[i].y += dt * v_half.y;
      cpu->pos[i].z += dt * v_half.z;

      // used for COLLISION
      // vi = (vi-1/2 + vi+1/2)/2
      cpu->vel[i].x += cpu->vel_half[i].x + v_half.x;
      cpu->vel[i].y += cpu->vel_half[i].y + v_half.y;
      cpu->vel[i].z += cpu->vel_half[i].z + v_half.z;
      cpu->vel[i].x /= 2.0f;
      cpu->vel[i].y /= 2.0f;
      cpu->vel[i].z /= 2.0f;
      memcpy( &cpu->vel_half[i], &v_half,
sizeof(vector3));
    }
  }
}
```

```
#define PI  3.14159265358979323846f   //(no dimension)
#define TIME_STEP 0.004f//0.004f (s)
#define DTIME_STEP 0.0001f//0.0001f (s)
#define EPSILON 1.0E-6f //(standar error)
#define BENCHMARK_ITR 500
#define LOOPS_MAX     1
#define STEPS_MAX     1
#define COLLISION_RADIUS 0.002f //(m), used for
collision detection
#define WINDOW_WIDTH     800
#define WINDOW_HEIGHT    600
#define SQRT2 1.41421356237309504880168872420 97f
#define SQRT2_P2 0.70710678118654752440084436210485f
#define PARTICLES_N
          20000//32000//18000

//Particles source
#define FLOW_RATE           2//30//80 //(particles/s)
//4000

#define SOURCE_RADIUS       0.3f//(VESSEL_RADIUS)
#define SOURCE_POS_X        -0.2f//-0.1f//(VESSEL_X0 +
SOURCE_RADIUS)     //(m)
#define SOURCE_POS_Y        0.0f//VESSEL_Y0    //(m)
#define SOURCE_POS_Z        0.0f//VESSEL_Z0    //(m)

#define SOURCE_VEL_X        0.5f//0.5f         //(m/s)
#define SOURCE_VEL_Y        0.0f      //(m/s)
#define SOURCE_VEL_Z        0.0f      //(m/s)

// GRID for sph

#define GRID_NMAX           256000 //64000
#define MAXNEIGHBOUR        500//350
#define MAX_PARTICLE_PERGRID350 //150

#define BLOOD_VISCOSITY     1.0f //(water=0.2)
#define BLOOD_STIFF         0.75f//(water=1.5)
#define BLOOD_MASS          0.0002f
//0.00021713951f(water=0.00020543f), mass partikel
untuk volume tertentu.
#define BLOOD_DENSITY       1057.0f//(kg/m3)

// SPH
#define SMOOTHING_LENGTH    0.01f     //0.01f
          // (in m), for density calc
#define H           SMOOTHING_LENGTH
#define SEARCH_RADIUS SMOOTHING_LENGTH //=grid width, &
for neighbour
#define R  SEARCH_RADIUS     //=grid width, & for
neighbour
#define ISO_THRESHOLD 600.0f //#define ISO_THRESHOLD
700.0f
#define MC_ISORADIUS (SMOOTHING_LENGTH) //0.0115f
#define MC_GRIDLEN (SMOOTHING_LENGTH/2.0f)
#define MC_RANGE   2
```

Adapun seluruh pendefinisian di atas menggunakan bahasa C. Sesuai dengan sifat di atas maka dapat didefinisikan satu komponen tambahan untuk setiap partikel, yakni memori untuk menyimpan byte data atau blok data yang akan dienkripsi dan dekripsi. Di dalam prosesnya, algoritma enkripsi dan dekripsi bertempat pada interaksi antarpartikelnya dengan memanipulasi nilai byte atau blok data dari kedua partikel yang saling bertumbukan tersebut di dalam satuan waktu tertentu. Misalkan dari sejumlah besar partikel terdapat dua partikel yang memiliki nilai 0xEF dan 0x33. Bila di algoritma yang digunakan untuk setiap tumbukan hanyalah operasi XOR untuk kedua nilai tersebut, maka partikel pertama dapat di XOR dengan kunci yang dimasukkan pengguna lalu di-XOR dengan partikel kedua dan demikian juga sebaliknya dengan partikel kedua terhadap partikel pertama. Di dalam praktiknya, setiap partikel dapat merupakan sebuah blok data yang berukuran 8 byte, 16 byte, atau lebih. Dengan demikian, operasi yang diberlakukan pada setiap partikel merupakan operasi pada setiap blok data dengan keteraturan yang bergantung dari keacakan partikel. Untuk *generate* partikel di kondisi awal digunakan pembangkit bilangan acak yang fungsinya sudah terintegrasi pada *library* dari bahasa pemrograman yang digunakan. Tentunya, pembangkitan bilangan acak tersebut akan selalu menghasilkan urutan keacakan yang sama untuk *seed* yang sama juga di setiap pembangkitannya sehingga posisi setiap partikel akan selalu sama ketika pertama kali dibangkitkan. Dalam hal ini, tentu saja pola tumbukan yang akan terjadi pun menjadi sama. Tetapi, hal yang merupakan kelebihan utama dari metode ini yaitu tidak adanya kepastian yang dapat dihitung secara empirik tanpa adanya pengkondisian sistem yang sama dengan kondisi enkripsinya. Selain itu, jumlah tumbukan yang terjadi pun bergantung pada lamanya partikel-partikel tersebut dibiarkan bertumbuk (lamanya proses enkripsi). Proses enkripsi dihentikan sesuai dengan total waktu yang diambil dari kunci yang dimasukkan oleh pengguna. Dengan begitu, kunci yang dimasukkan oleh pengguna berfungsi sebagai pembangkit bilangan acak untuk posisi partikel di dalam ruang, total waktu enkripsi data (total waktu partikel-partikel bertumbukan), dan kunci untuk algoritma enkripsinya sendiri.

Secara struktur, algoritma keseluruhan dari metode ini hampir sama dengan metode kriptografi modern yang dipakai secara umum, yakni membentuk cipher berulang untuk algoritma yang sama dan membagi operasi yang terjadi di dalam keseluruhan proses ke dalam beberapa bagian melalui jaringan feistel. Perbedaan utama yaitu di dalam keacakan cipher berulang dan interaksi yang dibentuk antarpartikelnya yang bersifat acak (sebanding dengan blok cipher berantai).

B. Penggunaan SPH dalam Steganografi

Di dalam bidang steganografi, prinsip tumbukan partikel SPH dapat diimplementasikan di dalam pembangkitan keacakan posisi penyisipan bit-bit data di dalam stego-file yang digunakan. Pada dasarnya pengacakan posisi penyisipan pada awalnya dibangkitkan dari pembangkit bilangan acak yang terintegrasi dengan bahasa pemrograman yang digunakan pada program yang bersangkutan atau melalui fungsi yang dibuat sendiri. Akan tetapi, tingkat keamanan ini masih rentan terhadap pemecahan oleh kriptanalis yang dapat menemukan kunci (*seed*) pembangkit bilangan acak itu. Dengan adanya tumbukan-tumbukan dari partikel-partikel yang berisi nilai posisi bit di dalam stego-file maka tingkat keacakan penyisipan pun menjadi semakin sulit dipecahkan dengan hanya melakukan kriptanalisis umum. Algoritma yang digunakan dapat berupa algoritma sederhana berupa penukaran nilai bit-bit yang disimpan di dalam setiap partikel pada saat mereka bertumbukan dengan indeks partikel menyatakan nilai posisi awal.

IV. KESIMPULAN

Dengan menggunakan sifat SPH dan metode pengacakan nilai-nilai data yang bersangkutan maka dapat dibentuk ciphertext yang sangat sulit dipecahkan dengan metode pemecahan umum karena tingkat keacakan susunan cipher berulang yang terjadi selama proses enkripsi di luar jangkauan pendefinisian satu fungsi sederhana oleh program pada umumnya. Kekurangan dari metode ini yaitu jika waktu enkripsi yang didefinisikan dari kunci enkripsinya cukup panjang, maka proses enkripsinya akan menjadi berkalilipat lebih lama dari proses enkripsi file yang sama dengan algoritma enkripsi lainnya. Akan tetapi, hal ini sebanding dengan tingkat keamanan yang tinggi dari enkripsi dan dekripsi yang dilakukan. Salah satu cara untuk mengatasi kelemahan ini yaitu dengan melakukan seluruh proses ini menggunakan parallel processor yang membuat sejumlah besar operasi dapat dilakukan dalam waktu yang bersamaan.

REFERENCES

[1] G. R. Liu dan M. B. Liu, "Smoothed Particles Hydrodynamics," Singapore: World Scientific Publishing, 2003, pp. 5–27.

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, 23 Maret 2011

Christian (13207033)