Kriptografi Visual dengan *Plain Partition* dan Skema (n-1,n)

Hadi Saloko - 13504157[1)]
1) Jurusan Teknik Informatika STEI, ITB, Bandung 40132, email: if14157@students.if.itb.ac.id

Abstract *– Kriptografi visual adalah bidang dalam kriptografi yang mengenkripsi gambar (sebagai plain), dan menghasilkan beberapa share (sebagai cipher). Share yang satu merupakan kunci dari share yang lainnya. Dengan mendekripsi share-share (cipher), gambar awal (plain) akan dapat diperoleh kembali.*

Terdapat berbagai varian dalam kriptografi visual. Ada kriptografi visual dengan proses dekripsi yang rumit (diperlukan komputer sebagai alat bantu). Sebaliknya, ada kriptografi visual dengan proses dekripsi yang sederhana (cukup menumpukkan transparansi share-nya).

Ada kriptografi visual yang yang terang-terangan menyatakan share-nya (share-nya mencurigakan, berupa gambar tanpa makna. Sebaliknya, ada krptografi visual yang menyembunyikan share-nya dengan steganografi (share-nya disembunyikan ke dalam gambar dengan makna berbeda).

Ada juga kriptografi visual yang memerlukan keseluruhan dari share-nya untuk proses dekripsi. Sebaliknya, ada krptografi visual yang hanya memerlukan beberapa/sebagian dari share-nya untuk proses dekripsi.

Kriptografi visual yang diangkat menjadi topik, memerlukan proses dekripsi yang sederhana (menumpukkan transparansi share-nya). Transparansi sharenya berupa gambar tanpa makna, karena itu tergolong kriptografi visual yang terang-terangan menyatakan share-nya (share-nya mencurigakan). Untuk proses dekripsi, secara umum hanya diperlukan (n-1) buah share dari total n buah share. Akan tetapi dari n-1 buah share tersebut hanya akan diperoleh bagian/partisi dari gambar awal (plain partition).

Dengan kata lain, dengan mencoba seluruh kombinasi (n-1) buah share, barulah diperoleh keseluruhan gambar awal (plain).

Kata Kunci*: kriptografi visual, plain, cipher, share, plain partition.*

1. PENDAHULUAN

Berbeda dengan skema (k,n) yang memerlukan minimal k buah *share* untuk proses dekripsi, kriptografi visual usulan memerlukan tepat (n-1) buah *share* untuk proses dekripsi (walau hanya menghasilkan *plain partition*).

Selain itu, jika pada skema (k,n) kombinasi manapun dari minimal k buah *share* akan menghasilkan gambar hasil yang relatif sama (yaitu gambar awal (*plain*)), kriptografi visual usulan dengan kombinasi (n-1) buah *share* tertentu akan menghasilkan gambar hasil yang berbeda-beda (*plain partition* yang berbeda). Misalnya saja, kombinasi (n-1) buah *share* tertentu akan menghasilkan *plain partition* sebelah kiri, sementara kombinasi yang lainnya akan menghasilkan *plain partition* sebelah kanan.

Pada kenyataannya tidak seluruh kombinasi (n-1) buah *share* akan menghasilkan *plain partition*. Seandainya dipaksa agar seluruh kombinasi (n-1) buah *share* menghasilkan *plain partition*, *share* yang dihasilkan proses enkripsi akan sangat mirip dengan gambar awal (*plain*). Hal ini tentunya akan menghilangkan aspek kriptografi. Untuk kombinasi yang tidak menghasilkan *plain partition*, akan menghasilkan *noise* (akan dibahas di bagian selanjutnya).

Kriptografi visual usulan memiliki berbagai kelebihan. Salah satunya adalah konsep baru yang berbeda dengan skema (k,n). Konsep dimana memang tidak diperlukan seluruh n buah *share*, tetapi dengan (n-1) *share* hanya akan menghasilkan *plain partition* (atau untuk beberapa kombinasi akan menghasilkan *noise*). Dengan kata lain dengan sebagian *share* hanya akan dihasilkan *plain partition*. Sebagian kunci hanya akan menghasilkan sebagian jawaban.

Selain itu sebenarnya jumlah kombinasi yang akan menghasilkan *noise* dapat diatur juga, bahkan *noise* yang dihasilkan juga dapat diatur. Dari sini didapatkan keuntungan bahwa jika pihak yang tidak berwenang (dengan suatu cara) berhasil memperoleh (n-1) buah *share*, ada kemungkinan kombinasi tersebut hanya akan menghasilkan *noise* (jika dipilih *noise* yang bermakna, tentunya akan mengecoh pihak yang tidak berwenang tersebut).

2. ANALISIS DAN PERANCANGAN

2.1. *Plain* dan *cipher*

Walaupun masih belum mempertimbangkan gambar berwarna sebagai *plain*, kriptografi visual usulan menerima *plain* berupa gambar *grayscale*. *Cipher*

yang dihasilkan juga berupa gambar *grayscale*. Hal ini sebenarnya ditujukan untuk mengatasi kelemahan kriptografi visual dengan *plain* dan *cipher* berupa gambar *black and white* (BW), yang umumnya menerapkan operasi OR untuk proses dekripsi.

Dengan menggunakan operasi OR, dari sebuah *share* sebenarnya terdapat dua buah kelemahan.

Kelemahan pertama adalah dapat diperolehnya informasi mengenai *plain*. Seandainya *pixel* posisi (x,y) pada *share* merupakan *pixel* hitam, berarti *pixel* posisi (x,y) pada *plain* pasti juga merupakan *pixel* hitam. Dapat dikatakan bahwa hal ini mengurangi aspek kriptografi, yakni bahwa data tidak ter-enkripsi dengan baik.

Kelemahan kedua adalah jika *pixel* posisi (x,y) pada *plain* merupakan *pixel* putih, maka *pixel* posisi (x,y) pada seluruh *share* haruslah merupakan *pixel* putih. Dapat dikatakan bahwa hal ini mengurangi variasi dari *cipher*/*share* hasil enkripsi. Untuk *plain* merupakan gambar yang relatif cerah, seluruh *share* yang dihasilkan proses enkripsi akan merupakan gambar yang relatif cerah juga.

Walaupun dapat menangani *plain* berupa gambar *grayscale*, *plain* yang akan diuji coba adalah gambar BW. Hal ini ditujukan agar pengujian dapat lebih mudah dilakukan. Yakni pengujian seberapa baik hasil enkripsi dan dekripsi.

Sedangkan *cipher* akan berupa gambar *grayscale*, untuk mengatasi dua kelemahan operasi OR pada kriptografi visual dengan *plain* dan *cipher* berupa gambar BW. Kedua kelemahan tersebut (dan ternyata banyak kelemahan lainnya) akan dibahas di bagian selanjutnya.

2.2. Pempartisian Plain

Sebelum dikenakan proses enkripsi, *plain* terlebih dahulu dipartisi kedalam *plain partition.* Pempartisian ini sangat tergantung banyak partisi dan besar partisi yang diinginkan pengguna.

Dari banyak partisi, secara otomatis perangkat lunak menentukan pempartisian yang baik yang akan menghasilkan banyak partisi yang diinginkan. Pempartisian yang baik adalah pempartisian yang seimbang secara horizotal maupun vertikal. Misalnya saja jika pengguna menginginkan 7 partisi, perangkat lunak secara otomatis akan membagi *plain* menjadi 3 kolom dan 2 baris (3 * 2 = 6 = 7-1). Pengurangan satu disini menunjukan banyak noise yang terlibat (dibahas di bagian selanjutnya).

Dari besar partisi, secara otomatis pula perangkat lunakn menentukan ukuran (panjang dan lebar, *width* dan *heigh*) dari tiap partisi. Termasuk mengatur *overlapped-area*-nya, sehingga tidak ada data pada plain yang hilang (diabaikan).

2.3. Operasi Pixel

Seperti yang telah dijelaskan di atas, proses dekripsi kriptografi visual usulan termasuk proses dekripsi yang sederhana (cukup menumpukkan transparansi *share*-nya). Tetapi untuk mengatasi kekurangan operasi OR, diperlukan operasi *pixel* yang berbeda.

Operasi *pixel* yang dipilih mungkin sedikit sulit untuk proses enkripsi. Akan tetapi perlu diingat bahwa operasi *pixel* tersebut haruslah tetap sederhana pada proses dekripsi (cukup menumpukkan transparansi *share*-nya).

Konversi Alpha

Operasi *pixel*-nya merlibatkan perhitungan *alpha* suatu *pixel* (yang berwarna hitam). Dengan kata lain dibutuhkan suatu konversi nilai RGB ke dalam nilai ARGB. Untungnya konversi ini termasuk konversi yang cukup sederhana (didasarkan pada kesetaraan/kesebangunan).

Pixel hitam (dengan R=0, G=0, B=0) dikonversi menjadi *pixel* hitam dengan nilai *alpha*=0 (dengan *alpha*=255, R=0, G=0, B=0). Pixel putih (dengan R=255, G=255, B=255) dikonversi menjadi *pixel* hitam dengan nilai *alpha*=0 (dengan *alpha*=0, R=0, G=0, B=0). Pixel abu-abu (hitam 50%, R=128, G=128, B=128) merupakan *pixel* hitam dengan nilai *alpha*=128 (dengan *alpha*=128, R=0, G=0, B=0).

Sebenarnya, konversi nilai RGB ke nilai ARGB bukanlah hal yang esensial pada operasi *pixel*. Konversi ini diperlukan agar *share* yang dihasilkan proses dekripsi bisa langsung ditumpukan di segala aplikasi (termasuk aplikasi pengolahan gambar ataupun aplikasi lainnya). Sehingga untuk mencoba hasilnya tidak diperlukan proses manual seperti pembuatan transparansi *share*-nya dan penumpukan transparansi *share*-nya.

Setelah dikonversi dari nilai RGB ke nilai ARGB, barulah dilakukan operasi “penumpukan” *pixel* itu sendiri.

Operasi Penumpukan

Sama dengan operasi OR, *pixel* hitam ditumpuk dengan *pixel* putih akan menghasilkan *pixel* hitam. Tetapi *pixel* abu-abu ditumpukkan *pixel* abu-abu tidak menghasilkan *pixel* hitam, melainkan akan menghasilkan *pixel* abu-abu gelap (hitam 75%) yang merupakan *pixel* hitam dengan nilai *alpha*=192 (dengan *alpha*=192, R=0, G=0, B=0).

Dari perhitungan *alpha* suatu *pixel*, intensitas cahaya yang mampu menembus.*pixel* tersebut dapat dihitung. *Pixel* abu-abu (*pixel* hitam dengan nilai *alpha*=128) akan menembuskan 50% (100% * 128/256) intensitas cahaya yang masuk.

Cahaya yang tembus (50% cahaya semula) akan melewati *pixel* abu-abu kedua dengan nilai *alpha*=128 (*pixel* hitam dengan nilai *alpha*=128) dan akan menembuskan 25% dari cahaya semula (50%*50%=25% dari cahaya semua).

Hal ini dapat dikatakan bahwa dengan menumpuk *pixel* abu-abu (hitam 50%) dengan *pixel* abu-abu (hitam 50%) yang lain akan terlihat sebagai *pixel* abu-abu gelap (hitam 75%, yang akan memembuskan 25% dari cahaya semula).

Implikasi Operasi Perkalian

Operasi seperti ini mengimplikasikan dua hal penting yang sangat berpengaruh pada analisis dan implementasi perangkat lunak dengan konsep kriptografi visual usulan.

Implikasi yang pertama, bahwa persamaan yang terlibat bukanlah persamaan linier. Jadi tidak dapat dicari solusinya dengan metode *matrix* biasa.

Persamaan yang terlibat dapat dimisalkan sebagai berikut:

$s_1 * s_2 = n$ ..... (persamaan 1)
$s_1 * s_3 = p_2$ ..... (persamaan 2)
$s_2 * s_2 = p_1$ ..... (persamaan 3)

dimana s_1, s_2, dan s_3 adalah ketiga buah *share* yang dihasilkan, n adalah *noise* (akan dibahas di bagian selanjutnya), p_1 dan p_1 adalah *plain partition* (hasil pempartisian dari *plain*).

Dari ke 6 variabel di atas, 3 diantaranya sudah diberikan (n, p_1, dan p_2) dan 3 sisanya merupakan variabel yang dicari. (s_1, s_2, dan s_3).

Walaupun kita hendak mencari 3 variabel dari 3 persamaan, solusinya tidak dapat dicari dengan metode *matrix* biasa (karena persamaan yang terlibat bukan persamaan linier).

Solusinya dapat diperoleh dengan mengalikan seluruh persamaan yang terlibat. Hasil perkalian dari ketiga contoh persamaan di atas akan menghasilkan persamaan sebagai berikut:

$(s_1)^2 * (s_2)^2 * (s_3)^2 = n * p_2 * p_1$ ..... (persamaan 4)

Setelah ruas kanan dari (persamaan 4) dihitung, nilai s_1 dapat diperoleh dengan mengakarkan (persamaan 4) (dalam contoh ini digunakan akar kuadrat), kemudian membaginya dengan (persamaan 3).

$$s_1 = \frac{\sqrt{n * p_2 * p_1}}{p_1} = \frac{\sqrt{n * p_2}}{\sqrt{p_1}}$$

Nilai s_2 dan s_3 dapat juga diperoleh dengan cara yang serupa (melibatkan (persamaan 1) dan (persamaan 2)).

Implikasi yang kedua, bahwa diperlukan n buah perkalian untuk n buah *plain partition*, yang akan menghasilkan hasil perkalian dengan nilai maksimum yang beragam (bergantung pada n).

Ada dua alternatif, alternatif pertama dengan mengalikan setelah sebelumnya dikonversi tipe datanya (ke dalam tipe data *float* atau *double*). Alternatif ini cukup sederhana, menghemat waktu eksekusi dan memori, tetapi terdapat suatu ketidak akuratan dalam perhitungan yang mempengaruhi hasil.

Karena itu dipilihlah alternatif kedua yakni mengalikan apa adanya, dengan catatan diperlukan suatu tipe data yang nilai maksimumnya beragam (kapasitasnya besar).

Dalam makalah ini, alternatif kedua dipilih dengan menggunakan kelas BigInteger yang sudah ada (http://www.codeproject.com/csharp/biginteger.asp).

2.4. Toleransi

Salah satu kelemahan yang ingin diatasi adalah dengan operasi OR, jika *pixel* posisi (x,y) pada *plain* adalah *pixel* putih, maka *pixel* posisi (x,y) pada seluruh *share* haruslah *pixel* putih. Karena itu diusulkan suatu konsep tentang toleransi, artinya tiap *pixel* tidak dienkripsi apa adanya. Yang dienkripsi adalah *pixel* yang sudah dikonversi dengan toleransi tertentu.

Dengan kata lain gambar awal sebelum dienkripsi akan berbeda dengan gambar hasil dekripsi (bersifat *lossy*). Dengan membuang sifat *lossless* dari hasil dekripsi, kita akan mendapatkan hasil enkripsi yang lebih baik. Untuk *plain* yang cerah, *share* hasil enkripsi tidak melulu merupakan *share* yang cerah. Semakin tinggi tingkat toleransinya, semakin gelap *share* yang dapat dihasilkan proses enkripsi.

Toleransi ini terutama ditujukan untuk mengkonversi *pixel* putih menjadi *pixel* abu-abu (pada *plain*) sebelum masuk ke proses enkripsi.

2.5. Noise

Yang pertama-tama perlu ditelaah adalah untuk n buah *share* akan terdapat n buah kombinasi (n-1) buah *share*.(karena ${}_nC_{n-1} = n$).

Jika kita tinjau suatu posisi (x,y), n buah kombinasi dapat dikatakan sebagai n buah persamaan. Sedangkan tiap *share* dapat dikatakan sebagai variabel (contoh persamaan dan variabel ini dapat dilihat di bagian sebelumnya).

Karena terdapat n buah persamaan untuk n buah variabel, berarti n buah variabel tersebut merupakan variabel terikat. Untuk n persamaan yang sama, hanya ada satu solusi (unik) untuk tiap-tiap variabel.

Sekarang kita tinjau suatu posisi (a,b), jika nilai *pixel* pada posisi tersebut sama persis dengan posisi (x,y) (untuk tiap *plain partition*); maka nilai *pixel* posisi (a,b) akan sama dengan posisi (x,y) (untuk tiap *share*).

Kemungkinan nilai *pixel* pada posisi (a,b) sama dengan nilai *pixel* pada posisi (x,y) sangatlah besar (karena gambar natural, yang biasa kita gunakan, merupakan gambar yang kontinu). Terutama untuk *pixel* pada posisi (x,y) yang bersebelahan dengan *pixel* pada posisi (a,b). Berarti *share* yang dihasilkan akan merupakan gambar yang kontinu juga, dan hal ini mengurangi tingkat keacakan *share* hasil proses enkripsi.

Untuk mengatasi hal tersebut, diperlukan suatu *noise*. Dengan menggunakan satu buah *noise*, dapat dikatakan kita menambah jumlah persamaan yang terlibat. Karena terdapat (n+1=m) buah persamaan untuk (n=m-1) buah variabel, berariti n buah vaiabel tersebut merupakan variabel bebas. Atau lebih tepatnya dikatakan terikat terhadap *plain* ditambah *noise* (dimana *noise* tidak terikat apapun).

Noise yang Baik

Lalu bagaimanakah *noise* yang baik? Noise sebaiknya tidak terikat apapun. Walaupun demikian, *noise* yang baik masih memiliki sifat yang mirip dengan *plain*.

Sifat disini diantaranya saja tingkat kecerahan. Untuk *plain* yang cerah (banyak *pixel* putih, banyak *pixel* dengan tingkat *alpha* rendah), sebaiknya *noise* yang digunakan juga *noise* yang cerah. Dengan memakai *noise* yang sifatnya mirip dengan *plain*, akan menghasilkan *cipher* yang relatif sama sifatnya satu sama lain.

Jika untuk *plain* yang cerah kita menggunakan (sebuah) *noise* yang gelap, maka akan terdapat (sebuah) *cipher* yang berbeda tingkat kecerahannya jika dibandingkan *cipher* yang lain.

Untuk makalah ini, *noise* yang digunakan akan dibatasi baik jumlah maupun cara memperolehnya. Yang akan digunakan hanyalah sebuah *noise* yang di-*generate* secara acak dari *plain* (agar relatif sama tingkat kecerahannya).

Sebelumnya pernah dicoba *noise* yang benar-benar tidak terikat apapun, tetapi seperti yang telah dijelaskan sebelumnya, muncul sebuah *cipher* yang berbeda tingkat kecerahannya jika dibandingkan *cipher* yang lain (misalnya saja sebuah *cipher* yang cerah diantara 3 *cipher* lain yang gelap).

Pengembangan Noise Selanjutnya

Untuk pengembangan selanjutnya, jumlah *noise* dapat saja diatur bahkan diharapkan isi dari *noise*-nya sendiri dapat diatur.

Dengan mengatur isi *noise* (memakai *noise* yang bermakna), dari *cipher* semakin sulit diperoleh informasi yang mengarah ke *plain*. Bahkan untuk pihak yang tidak berwenang (yang tidak tahu bahwa gambar tersebut merupakan *noise*), justru diperoleh informasi yang mengarah ke *noise* (gambar yang bermakna, namun makna yang palse/salah).

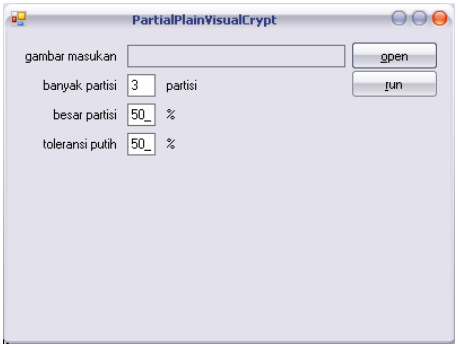
Noise juga sebaiknya digunakan untuk meng-*cross-check* apakah *cipher* yang kita dekripsi memang *cipher* yang benar.

Tinjau saja kasus dimana ada seseorang yang mengaku memiliki *cipher* dan ingin mencoba mendekripsi (dengan meminjam *cipher* yang lain). Ternyata setelah didekripsi hanya menghasilkan gambar tidak bermakna.

Untuk menghindari kemungkinan bahwa orang tersebut hanyalah orang yang tidak berwenang dan ingin mencuri *cipher* kita, gambar hasil dekripsi dapat di-*cross-check* dengan *noise* (yang terlibat saat proses enkripsi). Jika gambar hasil dekripsi tidak sesuai dengan *noise* tersebut, maka orang tersebut pastilah orang yang tidak berwenang dan ingin mencuri *cipher* kita.

3. HASIL DAN PEMBAHASAN

3.1. Analisis



Gambar 1 *screenshot* perangkat lunak

Seperti yang dapat dilihat di screenshot di atas, perangkat lunak yang dihasilkan meperlukan beberapa parameter (selain *plain*). Perangkat lunak ini dibuat dengan Microsoft Visual Studio C#, karena itu memerlukan .Net Framework untuk dapat menjalankannya.

Parameter pertama menentukan banyaknya partisi. Jika ditentukan banyak partisi=4 akan dihasilkan 4 *cipher* yang kombinasikan akan menghasilkan 3 *plain partition* + 1 *noise*.

Parameter kedua menentukan besarnya partisi. Jika ditentukan besar partisi=75%, dengan mengombinasikan *cipher-cipher* akan dihasilkan *plain partition* berukuran 75% dari *plain*. Jika diset besar partisi=100%, maka *cipher* yang dihasilkan akan sesuai dengan skema (n-1,n) biasa (tanpa pempartisian pada *plain*).

Parameter ketiga menentukan toleransi putih. Jika ditentukan toleransi putih=25%, maka sebelum dienkripsi tiap-tiap *pixel* akan dikonversi 25% lebih hitam. Misalnya saja *pixel* putih (hitam 0%) akan dikonversi menjadi abu-abu muda (hitam 25%).

Awalnya gabungan ketiga parameter (ditambah dengan faktor utama yaitu gambar *plain*), diperkirakan akan dapat menentukan seberapa baik hasil enkripsi dan dekripsi.

Hasil yang baik dapat dilihat dari hasil enkripsi yang tidak bermakna (*cipher* yang sama sekali tidak terlihat polanya), dan hasil dekripsi yang cukup jelas (gambar hasil dekripsi yang mirip dengan *plain*).

Akan tetapi setelah diuji coba, ketiga parameter tersebut sangat berkaitan, sehingga untuk memperoleh hasil enkripsi dan dekripsi yang baik tidak dapat dirumuskan kedalam ketiga parameter itu saja. Belum lagi ditambah dengan adanya faktor acak dalam pembentukan *noise*.

Untuk pengembangan selanjutnya, sebaiknya faktor acak dalam pembentukan *noise* dapat dihilangkan, sehingga dapat dirumuskan gabungan ketiga parameter mana yang menghasilkan hasil enkripsi dan dekripsi yang baik. Sehingga parameter toleransi dapat ditentukan secara otomatis oleh perangkat lunak. Pengguna hanya cukup menentukan banyak dan besar partisi yang diinginkan. Toleransi putih diset secara otomatis oleh perangkat lunak, sedemikian sehingga hasil yang diperoleh merupakan hasil yang terbaik.

Secara umum konsep ini dapat diimplentasikan dengan cukup baik, hal ini dapat dilihat di bagian selanjutnya yaitu hasil pengujian. Walaupun demikian terdapat beberapa kendala mengingat hasil pengujian ini sangat terkait dengan indera manusia dan media penyampaiannya.

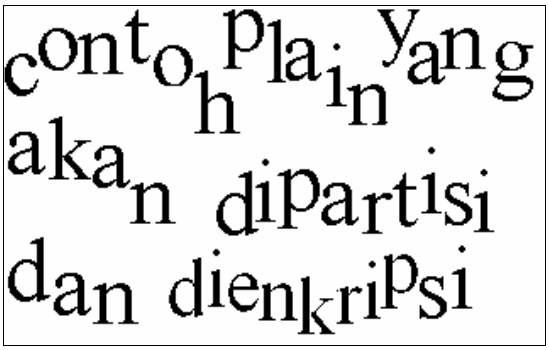
Bagi sebagian orang mungkin suatu pola cukup terlihat, sementara bagi sebagian orang yang lain tidak terlihat. Tergantung pada tingkat kepekaan dan indera visual orang-orang tersebut.

Bagi sebagian media mungkin akan memperlihatkan suatu pola, sementara sebagian media tidak akan memperlihatkan pola tersebut. Tergantung pada tingkat kecerahan, kontras, atau resolusi (pada monitor ataupun pada printer).

Walaupun demikian telah dicoba bahwa dengan printer bahwa hasil fisik (pada transparansi) dapat dianggap sama dengan hasil visualisasi (pada monitor dan dokumen). Hasil penumpukkannya pun dapat dianggap sama.

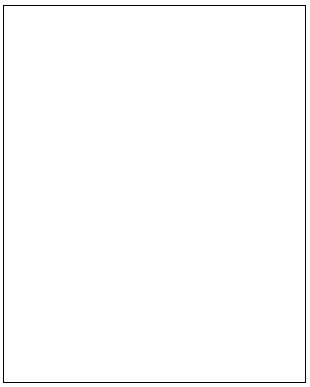
3.2. Hasil pengujian

Berikut disertakan hasil pengujian dengan kombinasi ketiga parameternya. Plain berupa gambar yang relatif cerah sebagai berikut.

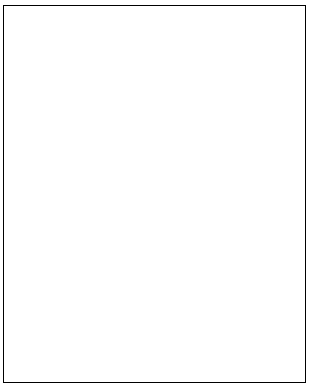


Gambar 2 *plain* (90% ukuran asli)

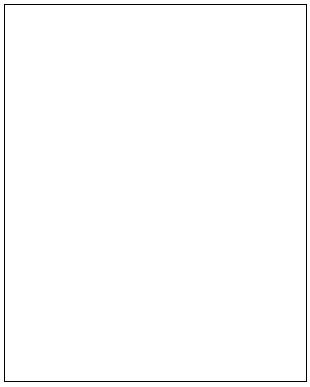
Untuk parameter: banyak partisi=3, besar partisi=50%, dan toleransi putih=60%; dihasilkan *cipher* sebagai berikut:



Gambar 3 *cipher* 1

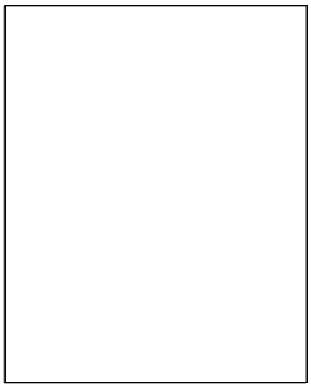


Gambar 4 *cipher* 2

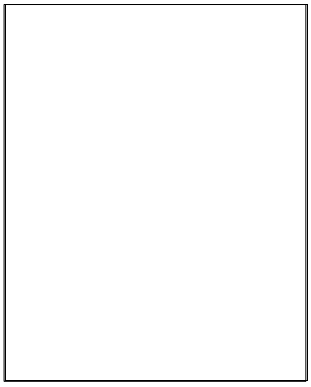


Gambar 5 *cipher* 3

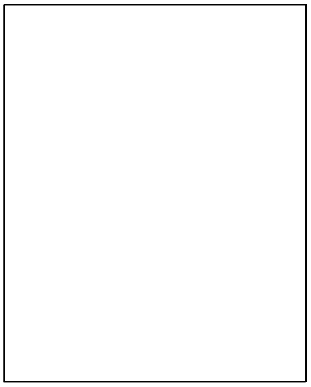
Dari ketiga *cipher* diatas, jika didekripsi (dengan cara menumpukkan *cipher*) akan diperoleh *noise* (*cipher* 1 + *cipher* 2), *plain partition* 1 (*cipher* 1 + *cipher* 3), dan *plain partition* 2 (*cipher* 2 + *cipher* 3). Jika dibandingkan dengan *plain*, dapat dilihat jika *plain partition* 1 merupakan 50% bagian kiri *plain* dan *plain partition* 2 merupakan 50% bagian kanan *plain*.



Gambar 6 *noise* 1



Gambar 7 *plain partition* 2



Gambar 8 *plain partition* 1

Masih terdapat hasil pengujian lainnya yang berbeda parameternya ataupun berbeda *plain*-nya, yang dapat dilihat di lampiran.

4. KESIMPULAN

Konsep kriptografi visual dengan *plain partition* dan skema (n-1,n) dirasa cukup memiliki nilai tambah. Untuk pengembangan selanjutnya pun dirasa masih cukup terbuka karena masih banya, yang dapat diekplorasi lebih lanjut. Diantaranya saja tentang jumlah *noise*, isi *noise*, tingkat toleransi putih, dan keterkaitan ketiga parameter diatas (sehingga memberikan hasil yang maksimal).

DAFTAR REFERENSI

[1] Munir, Rinaldi, Diktat Kuliah IF5054 Kriptografi, Institut Teknologi Bandung, 2006.

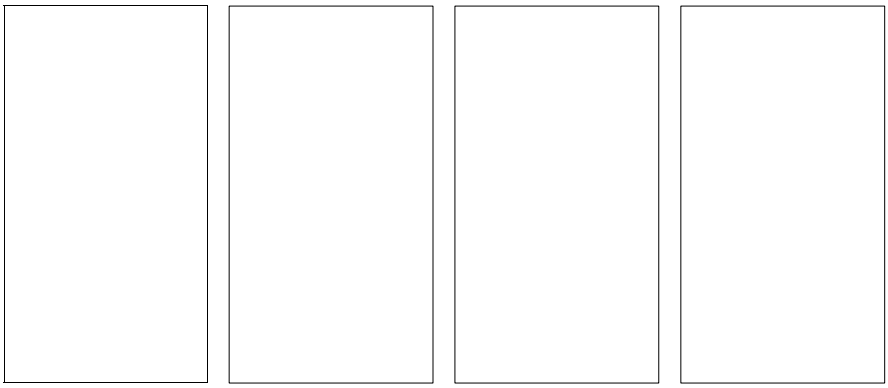
[2] Munir, Rinaldi, Kriptografi Visual: (Visual Cryptography), Bahan tambahan IF5054 Kriptografi, Institut Teknologi Bandung, 2006.

[3] Munir, Rinaldi, Skema Pembagian Data Rahasia (Secret Sharing Schemes), Bahan tambahan IF5054 Kriptografi, Institut Teknologi Bandung, 2006.

[4] Baharsyah, M.Pramana, Pemanfaatan Steganografi dalam Kriptografi Visual, Makalah IF5054, Institut Teknologi Bandung, 2006.

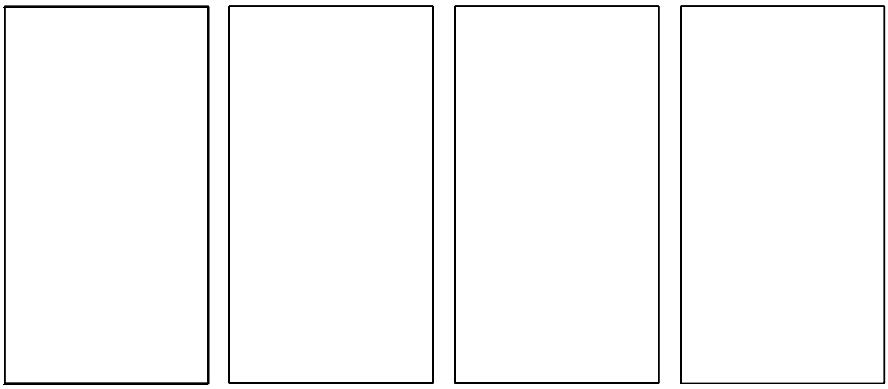
Lampiran

Untuk *plain* di bagian sebelumnya; dengan parameter: banyak partisi=4, besar partisi=34%, dan toleransi putih=35%; dihasilkan *cipher* sebagai berikut:



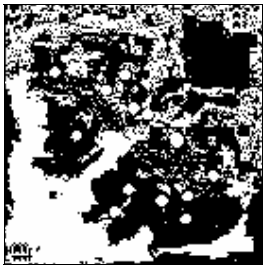
cipher 1 *cipher* 2 *cipher* 3 *cipher* 4

Dari keempat *cipher* diatas, jika didekripsi akan diperoleh: *noise* (*cipher* 1 + *cipher* 2 + *cipher* 3), *plain partition* 3 (*cipher* 1 + *cipher* 2 + *cipher* 4), *plain partition* 2 (*cipher* 1 + *cipher* 3 + *cipher* 4), dan *plain partition* 1 (*cipher* 2 + *cipher* 3 + *cipher* 4).



noise *plain partition* 3 *plain partition* 2 *plain partition* 1

Plain kedua berupa gambar yang relatif gelap sebagai berikut:



Untuk banyak partisi=3, besar partisi=50%, dan toleransi putih=45%; dihasilkan *cipher* sebagai berikut:

cipher 1 cipher2 *cipher* 3

Dari ketiga *cipher* diatas, jika didekripsi akan diperoleh: *noise* (*cipher* 1 + *cipher* 2), *plain partition* 2 (*cipher* 1 + *cipher* 3), dan *plain partition* 1 (*cipher* 2 + *cipher* 3).

noise

plain partition 2

plain partition 1

Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Runtime.InteropServices;
using System.Drawing.Imaging;

namespace PPVC
{
  unsafe public partial class Form1 : Form
  {
    private Bitmap inputBitmap;
    private Bitmap[] outputBitmap;
    private BitmapData myBitmapData;
    private byte*** arrInputBitmap;
    private byte**** arrPartisiBitmap;
    private byte**** arrOutputBitmap;
    private int inputBitmapWidth;
    private int inputBitmapHeight;
    private int inputBitmapPixelCount;
    private int banyakPartisi = 3;
    private int banyakPartisiPlain = 2;
    private int besarPartisi = 50;
    private int toleransi = 60;
    private int banyakPartisiH;
    private int banyakPartisiV;
    private int partisiWidth;
    private int partisiHeight;
    private int partisiPixelCount;
    private int overlapPartisiWidth;
    private int overlapPartisiHeight;

    public Form1()
    {
      InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
      if(openFileDialog1.ShowDialog().Equals(DialogResult.OK))
      {
        textBox1.Text = openFileDialog1.FileName;
        Bitmap tempInputBitmap = new Bitmap(textBox1.Text);
        inputBitmap = new Bitmap(tempInputBitmap);
        inputBitmapWidth = inputBitmap.Width;
        inputBitmapHeight = inputBitmap.Height;
        inputBitmapPixelCount = inputBitmapWidth*inputBitmapHeight;
        button2.Enabled = true;
      }
    }

    private void button2_Click(object sender, EventArgs e)
    {
      this.Enabled = false;
      int tempInt = (int)Math.Ceiling(Math.Sqrt((double)banyakPartisiPlain));
      while (banyakPartisiPlain % tempInt != 0)
      {
        tempInt++;
      }
      if (inputBitmapWidth > inputBitmapHeight)
```

```
      {
        banyakPartisiH = tempInt;
        banyakPartisiV=banyakPartisiPlain/banyakPartisiH;
      }
      else
      {
        banyakPartisiV = tempInt;
        banyakPartisiH=banyakPartisiPlain/banyakPartisiV;
      }
      double ratio = Math.Sqrt((double)besarPartisi * banyakPartisiPlain / 100);
      if (ratio > banyakPartisiH)
      {
        partisiWidth = inputBitmapWidth;
        partisiHeight = (inputBitmapPixelCount*besarPartisi) / (partisiWidth*100);
      }
      else if (ratio > banyakPartisiV)
      {
        partisiHeight = inputBitmapHeight;
        partisiWidth = (inputBitmapPixelCount*besarPartisi) / (partisiHeight*100);
      }
      else
      {
        partisiWidth = (int)Math.Ceiling((double)(ratio*inputBitmapWidth) / banyakPartisiH);
        partisiHeight = (int)Math.Ceiling((double)(ratio*inputBitmapHeight) / banyakPartisiV);
      }
      partisiPixelCount = partisiWidth * partisiHeight;

      if (banyakPartisiH > 1)
      {
        overlapPartisiWidth = (banyakPartisiH * partisiWidth - inputBitmapWidth) /
                              (banyakPartisiH - 1);
      }
      else
      {
        overlapPartisiWidth = 0;
      }
      if (banyakPartisiV > 1)
      {
        overlapPartisiHeight = (banyakPartisiV * partisiHeight - inputBitmapHeight) /
                               (banyakPartisiV - 1);
      }
      else
      {
        overlapPartisiHeight = 0;
      }
      Console.WriteLine("ratio=" + ratio);
      Console.WriteLine("toleransi=" + toleransi);
      Console.WriteLine("banyakPartisiH="+banyakPartisiH);
      Console.WriteLine("banyakPartisiV="+banyakPartisiV);
      Console.WriteLine("partisiWidth="+partisiWidth);
      Console.WriteLine("partisiHeight="+partisiHeight);
      Console.WriteLine("overlapPartisiWidth=" + overlapPartisiWidth);
      Console.WriteLine("overlapPartisiHeight=" + overlapPartisiHeight);

      myBitmapData = inputBitmap.LockBits(new Rectangle(0, 0, inputBitmapWidth,
                     inputBitmapHeight), ImageLockMode.ReadWrite,
                     PixelFormat.Format32bppArgb);

      byte* tempArr= (byte*)Marshal.AllocHGlobal(inputBitmapPixelCount * 4 *
                     sizeof(byte)).ToPointer();

      arrInputBitmap = (byte***)Marshal.AllocHGlobal(inputBitmapHeight *
                       sizeof(byte**)).ToPointer();
      for (int i = 0; i < inputBitmapHeight; i++)
      {
        arrInputBitmap[i] = (byte**)Marshal.AllocHGlobal(inputBitmapWidth *
                            sizeof(byte*)).ToPointer();
        tempInt = i * inputBitmapWidth;
        for (int j = 0; j < inputBitmapWidth; j++)
        {
          arrInputBitmap[i][j] = &tempArr[(tempInt + j) * 4];
        }
      }

      uint* myUIntPtr = (uint*)tempArr;
      uint* myUIntPtr2 = (uint*)myBitmapData.Scan0.ToPointer();
      for (int i = 0; i < inputBitmapPixelCount; i++)
```

```
      {
        myUIntPtr[i] = myUIntPtr2[i];
      }

      inputBitmap.UnlockBits(myBitmapData);

      arrPartisiBitmap = (byte****)Marshal.AllocHGlobal(banyakPartisi *
                         sizeof(byte***)).ToPointer();
      for (int i = 0; i < banyakPartisi; i++)
      {
        tempArr = (byte*)Marshal.AllocHGlobal(partisiPixelCount * 4 *
                  sizeof(byte)).ToPointer();

        arrPartisiBitmap[i] = (byte***)Marshal.AllocHGlobal(partisiHeight *
                              sizeof(byte**)).ToPointer();
        for (int j = 0; j < partisiHeight; j++)
        {
          arrPartisiBitmap[i][j] = (byte**)Marshal.AllocHGlobal(partisiWidth *
                                   sizeof(byte*)).ToPointer();
          tempInt = j * partisiWidth;
          for (int k = 0; k < partisiWidth; k++)
          {
            arrPartisiBitmap[i][j][k] = &tempArr[(tempInt + k) * 4];
          }
        }
      }

      for (int i = 0; i < banyakPartisiV; i++)
      {
        tempInt = i * (partisiHeight-overlapPartisiHeight);
        for (int j = 0; j < banyakPartisiH; j++)
        {
          int tempInt2=i*banyakPartisiH+j;
          int tempInt3 = j * (partisiWidth - overlapPartisiWidth);
          for (int k = 0; k < partisiHeight; k++)
          {
            for (int l = 0; l < partisiWidth; l++)
            {
              myUIntPtr=(uint*)arrPartisiBitmap[tempInt2][k][l];
              if (((tempInt + k) < inputBitmapHeight) && ((tempInt3 + l) < inputBitmapWidth))
              {
                myUIntPtr2 = (uint*)arrInputBitmap[tempInt + k][tempInt3 + l];
                myUIntPtr[0] = myUIntPtr2[0];
              }
              else
              {
                myUIntPtr[0] = 4278190080;
              }
              byte* myBytePtr = (byte*)myUIntPtr;
              myBytePtr[0] = (byte)(255 - myBytePtr[0] * (100 - (byte)toleransi) / 100);
              myBytePtr[1] = myBytePtr[0];
              myBytePtr[2] = myBytePtr[0];
            }
          }
        }
      }
      Random myRandom=new Random();
      for (int i = 0; i < partisiHeight; i++)
      {
        for (int j = 0; j < partisiWidth; j++)
        {
          myUIntPtr = (uint*)arrPartisiBitmap[banyakPartisiPlain][i][j];
          myUIntPtr2 = (uint*)arrPartisiBitmap[(int)Math.Floor(myRandom.NextDouble() *
                       banyakPartisiPlain)][(int)Math.Floor(myRandom.NextDouble() *
                       partisiHeight)][(int)Math.Floor(myRandom.NextDouble() * partisiWidth)];
          myUIntPtr[0] = myUIntPtr2[0];
        }
      }

      arrOutputBitmap = (byte****)Marshal.AllocHGlobal(banyakPartisi *
                        sizeof(byte***)).ToPointer();
      for (int i = 0; i < banyakPartisi; i++)
      {
        tempArr = (byte*)Marshal.AllocHGlobal(partisiPixelCount * 4 *
                  sizeof(byte)).ToPointer();
```

```
        arrOutputBitmap[i] = (byte***)Marshal.AllocHGlobal(partisiHeight *
                             sizeof(byte**)).ToPointer();
        for (int j = 0; j < partisiHeight; j++)
        {
          arrOutputBitmap[i][j] = (byte**)Marshal.AllocHGlobal(partisiWidth *
                                  sizeof(byte*)).ToPointer();
          int tempInt2 = j * partisiWidth;
          for (int k = 0; k < partisiWidth; k++)
          {
            arrOutputBitmap[i][j][k] = &tempArr[(tempInt2 + k) * 4];
          }
        }
      }

      progressBar1.Maximum = partisiHeight-1;
      for (int i = 0; i < partisiHeight; i++)
      {
        progressBar1.Value = i;
        for (int j = 0; j < partisiWidth; j++)
        {
          BigInteger myBigInteger = new BigInteger(1);
          BigInteger myBigInteger3 = new BigInteger(1);
          for (int k = 0; k < banyakPartisi; k++)
          {
            myBigInteger *= new
                            BigInteger(Convert.ToInt64(Math.Pow((double)arrPartisiBitmap[k][i]
                            [j][0], (double)1 / banyakPartisiPlain)));
            myBigInteger3 *= new
                             BigInteger(Convert.ToInt64((double)arrPartisiBitmap[k][i][j][0]))
                             ;
          }
          BigInteger myBigInteger4 = new BigInteger(1);
          for (int k = 0; k < banyakPartisiPlain; k++)
          {
            myBigInteger4 *= myBigInteger;
          }
          while (myBigInteger4 < myBigInteger3)
          {
            myBigInteger++;
            myBigInteger4 = new BigInteger(1);
            for (int k = 0; k < banyakPartisiPlain; k++)
            {
              myBigInteger4 *= myBigInteger;
            }
          }
          for (int k = 0; k < banyakPartisi; k++)
          {
            BigInteger myBigInteger2 = new
                                       BigInteger(Convert.ToInt64((double)arrPartisiBitmap[k][i
                                       ][j][0]));
            myBigInteger2 = myBigInteger / myBigInteger2;
            arrOutputBitmap[k][i][j][0] = 0;
            arrOutputBitmap[k][i][j][1] = 0;
            arrOutputBitmap[k][i][j][2] = 0;
            arrOutputBitmap[k][i][j][3] = (byte)myBigInteger2.IntValue();
          }
        }
      }

      outputBitmap=new Bitmap[banyakPartisi];
      for (int i = 0; i < banyakPartisi; i++)
      {
        outputBitmap[i]= new Bitmap(partisiWidth, partisiHeight);
        myBitmapData = outputBitmap[i].LockBits(new Rectangle(0, 0, partisiWidth,
                       partisiHeight), ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);
        myUIntPtr = (uint*)myBitmapData.Scan0.ToPointer();
        myUIntPtr2 = (uint*)arrOutputBitmap[i][0][0];
        for (int j = 0; j < partisiPixelCount; j++)
        {
          myUIntPtr[j] = myUIntPtr2[j];
        }
        outputBitmap[i].UnlockBits(myBitmapData);
        saveFileDialog1.FileName="cipher" + i + ".bmp";
        while (!saveFileDialog1.ShowDialog().Equals(DialogResult.OK)) { }
        outputBitmap[i].Save(saveFileDialog1.FileName);
      }
      this.Enabled = true;
```

```
    }

    private void trackBar3_ValueChanged(object sender, EventArgs e)
    {
      toleransi = trackBar3.Value;
      label7.Text = toleransi + "%";
    }

    private void trackBar2_ValueChanged(object sender, EventArgs e)
    {
      besarPartisi = trackBar2.Value;
      label5.Text = besarPartisi + "%";
    }

    private void trackBar1_ValueChanged(object sender, EventArgs e)
    {
      banyakPartisi = trackBar1.Value;
      banyakPartisiPlain = banyakPartisi - 1;
      label3.Text = banyakPartisi + " partisi";
      trackBar2.Minimum = (int)Math.Ceiling((double)100 / banyakPartisiPlain);
      trackBar2.Value = trackBar2.Minimum;
      trackBar2_ValueChanged(sender,e);
    }
  }
}
```