Iteration 0

# Summary

During Summer 2015, I began working on narrowing down my Master’s project. While collecting resources for my research, I had also starting implementing some of the visual cryptography schemes in Java. Below is a summary of my progress.

VCSTests.java contains the main method of the project. It produces a menu of options: encrypt a photo for secure sharing using the size invariant visual cryptography scheme; reveal the secret message from photo shares created by the size invariant visual cryptography scheme; encrypt a photo for secure sharing using the extended visual cryptography scheme; reveal the secret message from photo shares created by extended visual cryptography scheme; and terminate the program.

The size invariant visual cryptography scheme takes in a secret image and breaks the pixels into n shares. The user provides the n, the number of shares to be created by the program, and k, the minimum number of shares needed to reveal the encrypted secret. Since the user only gives the secret file, all of the resulting shares look like static. Hence, this style of cryptography is not very covert. Playing around with the size invariant scheme allowed me to get the two forms of the (k, n) scheme working: (2, n) and (3, 3). Other inputs I tried, like (3, 5) and (5, 5), resulted in the secret not being hidden in the shares. Instead, the shares were inverted versions of the secret message. This problem was associated with the matrices used to encrypt the secret.

As for decryption with the size invariant scheme, the user provides the image shares and the program superimposes the shares into a stack. A new image file reveals the secret message.

The extended visual cryptography scheme takes in a secret image file and multiple “innocent” image files. Since I was trying to understand the scheme, I only allowed two innocent images. This scheme hides the secret message in the innocent files. Two new shares are produced that look like updated forms of the innocent files. Currently, this only works for black and white images, not gray scale.

As for decryption with the extended scheme, it works similar to that of the size invariant scheme. The encoded innocent images are superimposed, and the secret is revealed.

# Source Code

VCSTests.java

1 import java.awt.Color;  
 2 import java.awt.image.BufferedImage;  
 3 import java.io.File;  
 4 import java.io.IOException;  
 5 import java.util.Arrays;  
 6 import java.util.Random;  
 7 import java.util.Scanner;  
 8   
 9 import javax.imageio.\*;  
 10   
 11 public class VCSTests  
 12 {  
 13 public static void main(String[] args)  
 14 {  
 15 Scanner input = new Scanner(System.in);  
 16 int userChoice;  
 17 do  
 18 {  
 19 System.out.println("Please select which operation you wish to perform by entering the number associated with it.");  
 20 System.out.println("\t1. Encrypt a photo for secure sharing using the size invariant visual cryptography scheme");  
 21 System.out.println("\t2. Reveal the secret message from photo shares created by size invariant visual cryptography scheme");  
 22 System.out.println("\t3. Encrypt a photo for secure sharing using the extended visual cryptography scheme");  
 23 System.out.println("\t4. Reveal the secret message from photo shares created by extended visual cryptography scheme");  
 24 System.out.println("\t5. Terminate the program");  
 25 System.out.print("\nYour choice: ");  
 26 userChoice = Integer.parseInt(input.nextLine());  
 27   
 28 switch(userChoice)  
 29 {  
 30 case 1:  
 31 String imgFileName;  
 32 int numOfShares;  
 33 int numToStack;  
 34   
 35 //Get inputs  
 36 System.out.print("\nEnter the name of your secret image file: ");  
 37 imgFileName = input.nextLine();  
 38   
 39 System.out.print("With this scheme there are two values you need to enter. n represents\n");  
 40 System.out.print("the number of shares you want created, while k represents the number of\n");  
 41 System.out.print("shares needed to stack in order to display the secret message. Please\n");  
 42 System.out.print("enter your k and n values.\n");  
 43 boolean userError = true;  
 44 do  
 45 {  
 46 System.out.print("n: ");  
 47 numOfShares = Integer.parseInt(input.nextLine());  
 48 System.out.print("k: ");  
 49 numToStack = Integer.parseInt(input.nextLine());  
 50 if(numToStack <= numOfShares)  
 51 {  
 52 userError = false;  
 53 }  
 54 else  
 55 {  
 56 System.out.println("\nERROR: Your k cannot be greater than n. Please try again.");  
 57 }  
 58 } while(userError);  
 59   
 60 /\*  
 61 Try to get the image file stored as a BufferedImage  
 62 \*/  
 63 BufferedImage secretMsg = null;  
 64 boolean foundFile;  
 65 try  
 66 {  
 67 //Supports GIF, PNG, JPEG, BMP, and WBMP  
 68 //Plug-ins for TIFF and JPEG 2000  
 69 secretMsg = ImageIO.read(new File(imgFileName));  
 70 foundFile = true;  
 71 }  
 72 catch (IOException e)  
 73 {  
 74 System.out.println("Error: The file you tried to encrypt does not exist.");  
 75 foundFile = false;  
 76 }  
 77   
 78 /\*  
 79 If the file is found, create an array for pixel info storage.  
 80 \*/  
 81 if(foundFile)  
 82 {  
 83 SizeInvariantVCS mySIVCS = new SizeInvariantVCS(numToStack, numOfShares, secretMsg);  
 84 mySIVCS.encryptImage();  
 85   
 86 //Create file names for the shares the message is hidden in  
 87 System.out.print("Do you wish to store these shared file in a subdirectory of this folder? Yes or No: ");  
 88 String folderResponse = input.nextLine();  
 89 boolean storeInFolder = (folderResponse.trim().charAt(0) == 'y' || folderResponse.trim().charAt(0) == 'Y');  
 90 String folderName = "";  
 91 if(storeInFolder)  
 92 {  
 93 System.out.print("Please enter the name of the folder without a trailing slash: ");  
 94 folderName = input.nextLine();  
 95 File directory = new File(folderName);  
 96 directory.mkdir();  
 97 }  
 98 String[] shareFileNames = new String[numOfShares];  
 99 for(int i = 0; i < numOfShares; i++)  
100 {  
101 if(storeInFolder)  
102 {  
103 shareFileNames[i] = folderName + "/share" + (i + 1) + ".png";  
104 }  
105 else  
106 {  
107 shareFileNames[i] = "share" + (i + 1) + ".png";  
108 }  
109 }  
110   
111 for(int i = 0; i < numOfShares; i++)  
112 {  
113 try  
114 {  
115 //Takes the pixel array and creates a new buffered image  
116 BufferedImage tempShare = new BufferedImage(mySIVCS.getImgWidth(), mySIVCS.getImgHeight(),   
117 BufferedImage.TYPE\_INT\_ARGB);  
118 tempShare.setRGB(0, 0, mySIVCS.getImgWidth(), mySIVCS.getImgHeight(),   
119 mySIVCS.getRGBPixelsForShares()[i], 0, mySIVCS.getImgWidth());  
120   
121 //Creates the file name of the new image  
122 File tempOutput = new File(shareFileNames[i]);  
123   
124 //Writes the buffered image to a png file  
125 ImageIO.write(tempShare, "png", tempOutput);  
126 }  
127 catch (IOException e)  
128 {  
129 System.out.println("Error!");  
130 }  
131 }  
132 }  
133 System.out.println("\n");  
134 break;  
135   
136 case 2:  
137 //Get number of shares to be stacked  
138 System.out.print("Please enter the number of shares you have: ");  
139 int numOfCurrShares = Integer.parseInt(input.nextLine());  
140   
141 //Get filenames of those shares  
142 String[] shareFileNames = new String[numOfCurrShares];  
143 for(int i = 0; i < numOfCurrShares; i++)  
144 {  
145 System.out.print("Please enter the filename for one of your shares: ");  
146 shareFileNames[i] = input.nextLine();  
147 }  
148   
149 //Convert images to BufferedImage objects  
150 BufferedImage[] shares = new BufferedImage[numOfCurrShares];  
151 boolean foundFiles = false;  
152 for(int i = 0; i < numOfCurrShares; i++)  
153 {  
154 try  
155 {  
156 //Supports GIF, PNG, JPEG, BMP, and WBMP  
157 //Plug-ins for TIFF and JPEG 2000  
158 shares[i] = ImageIO.read(new File(shareFileNames[i]));  
159 foundFiles = true;  
160 }  
161 catch (IOException e)  
162 {  
163 System.out.println("Error: The file you tried to stack does not exist.");  
164 foundFiles = false;  
165 }  
166 }  
167   
168 if(foundFiles)  
169 {  
170 SizeInvariantVCS mySIVCS = new SizeInvariantVCS(numOfCurrShares, shares);  
171 mySIVCS.decryptImage();  
172   
173 //Write the new image  
174 System.out.print("Please enter the file name you wish to give your stacked image: ");  
175 String secretMsgFileName = input.nextLine();  
176 int imgFileExtension = secretMsgFileName.indexOf('.');  
177 String extension = secretMsgFileName.substring(imgFileExtension + 1);  
178 try  
179 {  
180 //Takes the pixel array and creates a new buffered image  
181 BufferedImage secretImage = new BufferedImage(mySIVCS.getImgWidth(), mySIVCS.getImgHeight(),   
182 BufferedImage.TYPE\_INT\_ARGB);  
183 secretImage.setRGB(0, 0, mySIVCS.getImgWidth(), mySIVCS.getImgHeight(),   
184 mySIVCS.getDecryptImgPixels(), 0, mySIVCS.getImgWidth());  
185   
186 //Creates the file name of the new image  
187 File tempOutput = new File(secretMsgFileName);  
188   
189 //Writes the buffered image to a png file  
190 ImageIO.write(secretImage, extension, tempOutput);  
191 }  
192 catch (IOException e)  
193 {  
194 System.out.println("Error!");  
195 }  
196   
197 }  
198 System.out.println("\n");  
199 break;  
200   
201 case 3:  
202 System.out.println("\nNote, this portion is currently a (2,2)-EVCS. Changes will be made later.");  
203   
204 String secretFile;  
205 String[] innocentFiles = new String[2];  
206 //get name of secret msg file  
207 System.out.print("Enter the name of your secret image file: ");  
208 secretFile = input.nextLine();  
209   
210 BufferedImage secretImage = null;  
211 boolean fileFound;  
212 try  
213 {  
214 secretImage = ImageIO.read(new File(secretFile));  
215 fileFound = true;  
216 }  
217 catch(IOException e)  
218 {  
219 System.out.println("Error: The file you tried to encrypt does not exist.");  
220 fileFound = false;  
221 }  
222   
223 if(fileFound)  
224 {  
225 //get name of innocent 1  
226 //get name of innocent 2  
227 for(int i = 0; i < 2; i++)  
228 {  
229 System.out.print("Please enter the name of one of the \"innocent\" file shares: ");  
230 innocentFiles[i] = input.nextLine();  
231 }  
232 BufferedImage[] innocentShares = new BufferedImage[2];  
233 for(int i = 0; i < 2; i++)  
234 {  
235 try  
236 {  
237 innocentShares[i] = ImageIO.read(new File(innocentFiles[i]));  
238 fileFound = true;  
239 }  
240 catch(IOException e)  
241 {  
242 String errorString = "Error: The file \"" + innocentFiles[i] + "\" does not exist.";  
243 System.out.print(errorString);  
244 fileFound = false;  
245 }  
246 }  
247   
248 //if all files found  
249 if(fileFound)  
250 {  
251 //pass to extendedvcs obj  
252 ExtendedVCS myEVCS = new ExtendedVCS(secretImage, innocentShares);  
253 //encrypt  
254 myEVCS.encryptImage();  
255   
256 //get rgbs of new innocent files  
257 int [][] newInnocentRGB = myEVCS.getRGBPixelsForShares();  
258   
259 //print to image files  
260 System.out.print("Do you wish to store these shared file in a subdirectory of this folder? Yes or No: ");  
261 String folderResponse = input.nextLine();  
262 boolean storeInFolder = (folderResponse.trim().charAt(0) == 'y' || folderResponse.trim().charAt(0) == 'Y');  
263 String folderName = "";  
264 if(storeInFolder)  
265 {  
266 System.out.print("Please enter the name of the folder without a trailing slash: ");  
267 folderName = input.nextLine();  
268 File directory = new File(folderName);  
269 directory.mkdir();  
270 }  
271 String[] shareFiles = new String[2];  
272 for(int i = 0; i < 2; i++)  
273 {  
274 if(storeInFolder)  
275 {  
276 shareFiles[i] = folderName + "/share" + (i + 1) + ".png";  
277 }  
278 else  
279 {  
280 shareFiles[i] = "share" + (i + 1) + ".png";  
281 }  
282 }  
283   
284 for(int i = 0; i < 2; i++)  
285 {  
286 try  
287 {  
288 //Takes the pixel array and creates a new buffered image  
289 BufferedImage tempShare = new BufferedImage(myEVCS.getImgWidth(), myEVCS.getImgHeight(),   
290 BufferedImage.TYPE\_INT\_ARGB);  
291 tempShare.setRGB(0, 0, myEVCS.getImgWidth(), myEVCS.getImgHeight(),   
292 newInnocentRGB[i], 0, myEVCS.getImgWidth());  
293   
294 //Creates the file name of the new image  
295 File tempOutput = new File(shareFiles[i]);  
296   
297 //Writes the buffered image to a png file  
298 ImageIO.write(tempShare, "png", tempOutput);  
299 }  
300 catch (IOException e)  
301 {  
302 System.out.println("Error!");  
303 }  
304 }  
305 }  
306 System.out.println("\n");   
307 }  
308 break;  
309   
310 case 4:  
311 System.out.println("\nNote, this portion is currently a (2,2)-EVCS. Changes will be made later.");  
312   
313 //Get number of shares to be stacked  
314 //System.out.print("Please enter the number of shares you have: ");  
315 //int numOfCurrShares = Integer.parseInt(input.nextLine());  
316   
317 //Get filenames of those shares  
318 String[] shareFiles = new String[2];  
319 for(int i = 0; i < 2; i++)  
320 {  
321 System.out.print("Please enter the filename for one of your shares: ");  
322 shareFiles[i] = input.nextLine();  
323 }  
324   
325 //Convert images to BufferedImage objects  
326 BufferedImage[] sharesEVCS = new BufferedImage[2];  
327 boolean foundFilesEVCS = false;  
328 for(int i = 0; i < 2; i++)  
329 {  
330 try  
331 {  
332 //Supports GIF, PNG, JPEG, BMP, and WBMP  
333 //Plug-ins for TIFF and JPEG 2000  
334 sharesEVCS[i] = ImageIO.read(new File(shareFiles[i]));  
335 foundFilesEVCS = true;  
336 }  
337 catch (IOException e)  
338 {  
339 String errorString = "Error: The file \"" + shareFiles[i] + "\" does not exist, so it cannot be superimposed.";  
340 System.out.print(errorString);  
341 foundFilesEVCS = false;  
342 }  
343 }  
344   
345 if(foundFilesEVCS)  
346 {  
347 ExtendedVCS myEVCS = new ExtendedVCS(sharesEVCS);  
348 myEVCS.decryptImage();  
349   
350 //Write the new image  
351 System.out.print("Please enter the file name you wish to give your stacked image: ");  
352 String decryptedFileName = input.nextLine();  
353 int fileExtension = decryptedFileName.indexOf('.');  
354 String extensionEVCS = decryptedFileName.substring(fileExtension + 1);  
355 try  
356 {  
357 //Takes the pixel array and creates a new buffered image  
358 BufferedImage decryptedImage = new BufferedImage(myEVCS.getImgWidth(), myEVCS.getImgHeight(),   
359 BufferedImage.TYPE\_INT\_ARGB);  
360 decryptedImage.setRGB(0, 0, myEVCS.getImgWidth(), myEVCS.getImgHeight(),   
361 myEVCS.getDecryptImgPixels(), 0, myEVCS.getImgWidth());  
362   
363 //Creates the file name of the new image  
364 File tempOutput = new File(decryptedFileName);  
365   
366 //Writes the buffered image to a png file  
367 ImageIO.write(decryptedImage, extensionEVCS, tempOutput);  
368 }  
369 catch (IOException e)  
370 {  
371 System.out.println("Error!");  
372 }  
373   
374 }  
375 System.out.println("\n");  
376   
377 break;  
378   
379 case 5:  
380 System.out.println("Terminating program...");  
381 }  
382 } while(userChoice != 5);  
383   
384 }  
385 }

SizeInvariantVCS.java

1 import java.awt.Color;  
 2 import java.awt.image.BufferedImage;  
 3 import java.util.Arrays;  
 4 import java.util.Random;  
 5   
 6 public class SizeInvariantVCS  
 7 {  
 8 private int k;  
 9 private int n;  
 10 private int imgWidth;  
 11 private int imgHeight;  
 12 private int numColumns;  
 13 private BufferedImage secretMsg;  
 14 private int[][] shareRGBPixels;  
 15   
 16 private int numSharesToDecrypt;  
 17 private BufferedImage[] sharesToDecrypt;  
 18 private int[] secretMsgPixels;  
 19   
 20 //Matrices  
 21 int[][] c0 = null;  
 22 int[][] c1 = null;  
 23 int[][] s0 = null;  
 24 int[][] s1 = null;  
 25   
 26 SizeInvariantVCS(int numToStack, int numOfShares, BufferedImage secretMsgIn)  
 27 {  
 28 k = numToStack;  
 29 n = numOfShares;  
 30 secretMsg = secretMsgIn;  
 31 imgWidth = secretMsg.getWidth();  
 32 imgHeight = secretMsg.getHeight();  
 33 }  
 34   
 35 SizeInvariantVCS(int numOfCurrShares, BufferedImage[] shareImgs)  
 36 {  
 37 numSharesToDecrypt = numOfCurrShares;  
 38 sharesToDecrypt = shareImgs;  
 39 imgWidth = shareImgs[0].getWidth();  
 40 imgHeight = shareImgs[0].getHeight();  
 41 }  
 42   
 43 int getImgWidth()  
 44 {  
 45 return imgWidth;  
 46 }  
 47   
 48 int getImgHeight()  
 49 {  
 50 return imgHeight;  
 51 }  
 52   
 53 int[][] getRGBPixelsForShares()  
 54 {  
 55 return shareRGBPixels;  
 56 }  
 57   
 58 int[] getDecryptImgPixels()  
 59 {  
 60 return secretMsgPixels;  
 61 }  
 62   
 63 void encryptImage()  
 64 {  
 65 int[] secretRGB = new int[imgWidth \* imgHeight];  
 66 secretMsg.getRGB(0, 0, imgWidth, imgHeight, secretRGB, 0, imgWidth);  
 67 createCMatrices();  
 68 randomSMatrix(s0, c0);  
 69 randomSMatrix(s1, c1);  
 70 createPixelsOfShares(secretRGB);  
 71 }  
 72   
 73 void createCMatrices()  
 74 {  
 75 if(k == 2)  
 76 {  
 77 //Can be used for any scheme that has k = 2 (aka (2,N)-VCS)  
 78 numColumns = n;  
 79 c0 = new int[n][n];  
 80 s0 = new int[n][n];  
 81 for(int r = 0; r < n; r++)  
 82 {  
 83 c0[r][0] = 1;  
 84 }  
 85   
 86 c1 = new int[n][n];  
 87 s1 = new int[n][n];  
 88 for(int i = 0; i < n; i++)  
 89 {  
 90 c1[i][i] = 1;  
 91 }  
 92 }  
 93 /\*  
 94 The portion below only works if (3,3)-SIVCS  
 95 \*/  
 96 else if(k == 3) //had as k==n  
 97 {  
 98 numColumns = (int)Math.pow(2, (n - 1));  
 99 c0 = new int[n][numColumns];  
100 s0 = new int[n][numColumns];  
101 for(int c = 1; c < numColumns; c++)  
102 {  
103 for(int r = 0; r < n; r++)  
104 {  
105 if(c <= n)  
106 {  
107 if(r != (c - 1))  
108 {  
109 c0[r][c] = 1;  
110 }  
111 }  
112 else  
113 {  
114 int tempC = c % (n + 1);  
115 if(tempC != 0)  
116 {  
117 if(r != (tempC - 1))  
118 {  
119 c0[r][c] = 1;  
120 }  
121 }  
122 }  
123 }  
124 }  
125   
126 c1 = new int[n][numColumns];  
127 s1 = new int[n][numColumns];  
128 for(int c = 0; c < numColumns; c++)  
129 {  
130 for(int r = 0; r < n; r++)  
131 {  
132 if(c < n)  
133 {  
134 if(c == (n - 1 - r))  
135 {  
136 c1[r][c] = 1;  
137 }  
138 }  
139 else  
140 {  
141 int tempC = c % (n + 1);  
142 if(tempC == n)  
143 {  
144 c1[r][c] = 1;  
145 }  
146 else  
147 {  
148 if(tempC == (n - 1 - r))  
149 {  
150 c1[r][c] = 1;  
151 }  
152 }  
153 }  
154 }  
155 }  
156 }  
157 else  
158 {  
159 //TBD  
160 }  
161 }  
162   
163 void randomSMatrix(int[][] sMatrix, int[][] cMatrix)  
164 {  
165 boolean[] colPlaced = new boolean[numColumns];  
166 int[] colOrder = new int[0];  
167 int colCount = 0;  
168   
169 do  
170 {  
171 Random randomColGen = new Random();  
172 int randomColumn = randomColGen.nextInt(numColumns);  
173 if(!colPlaced[randomColumn])  
174 {  
175 colOrder = Arrays.copyOf(colOrder, colOrder.length + 1);  
176 colOrder[colCount] = randomColumn;  
177 colPlaced[randomColumn] = true;  
178 colCount += 1;  
179 }  
180 } while(colOrder.length != numColumns);  
181   
182 colCount = 0;  
183 for(int c = 0; c < numColumns; c++)  
184 {  
185 for(int r = 0; r < n; r++)  
186 {  
187 sMatrix[r][c] = cMatrix[r][colOrder[c]];  
188 }  
189 }  
190 }  
191   
192 void createPixelsOfShares(int[] origImgRGB)  
193 {  
194 shareRGBPixels = new int[n][imgWidth \* imgHeight];  
195 for(int i = 0; i < origImgRGB.length; i++)  
196 {  
197 int redVal = (origImgRGB[i] & 0x00ff0000) >> 16;  
198 int greenVal = (origImgRGB[i] & 0x0000ff00) >> 8;  
199 int blueVal = (origImgRGB[i] & 0x000000ff);  
200   
201 //If pixel is white  
202 if(redVal >= 128 || greenVal >= 128 || blueVal >= 128)  
203 {  
204 //randomly choose column from s0  
205 Random randomGen = new Random();  
206 int randomColumn = randomGen.nextInt(numColumns);  
207   
208 for(int j = 0; j < n; j++)  
209 {  
210 //if value 0, store white  
211 if(s0[j][randomColumn] == 0)  
212 {  
213 shareRGBPixels[j][i] = Color.WHITE.getRGB();  
214 }  
215 //if value 1, store black  
216 else  
217 {  
218 shareRGBPixels[j][i] = Color.BLACK.getRGB();  
219 }  
220 }  
221 }  
222 //If pixel is black  
223 else  
224 {  
225 //randomly choose column from s1  
226 Random randomGen = new Random();  
227 int randomColumn = randomGen.nextInt(numColumns);  
228   
229 for(int j = 0; j < n; j++)  
230 {  
231 //if value 0, store white  
232 if(s1[j][randomColumn] == 0)  
233 {  
234 shareRGBPixels[j][i] = Color.WHITE.getRGB();  
235 }  
236 //if value 1, store black  
237 else  
238 {  
239 shareRGBPixels[j][i] = Color.BLACK.getRGB();  
240 }  
241 }  
242 }  
243 }  
244 }  
245   
246 void decryptImage()  
247 {  
248 //Make a 2d array of pixel arrays  
249 int[][] pixelsToCompare = new int[numSharesToDecrypt][imgWidth \* imgHeight];  
250 secretMsgPixels = new int[imgWidth \* imgHeight];  
251   
252 //getRGB pixels of BufferedImages  
253 for(int i = 0; i < numSharesToDecrypt; i++)  
254 {  
255 sharesToDecrypt[i].getRGB(0, 0, imgWidth, imgHeight, pixelsToCompare[i], 0, imgWidth);  
256 }  
257   
258 //Logical OR pixel with all three share values  
259 int numOfPixels = pixelsToCompare[0].length;  
260 for(int i = 0; i < numOfPixels; i++)  
261 {  
262 int pixelColor = 0;  
263 for(int j = 0; j < numSharesToDecrypt; j++)  
264 {  
265 if(pixelsToCompare[j][i] == Color.WHITE.getRGB())  
266 {  
267 pixelColor = pixelColor | 0;  
268 }  
269 else  
270 {  
271 pixelColor = pixelColor | 1;  
272 }  
273 }  
274   
275 //Store the result in an array after converting to WHITE and BLACK  
276 if(pixelColor == 1)  
277 {  
278 secretMsgPixels[i] = Color.BLACK.getRGB();  
279 }  
280 else  
281 {  
282 secretMsgPixels[i] = Color.WHITE.getRGB();  
283 }  
284 }  
285 }  
286   
287 }

ExtendedVCS.java

1 import java.awt.Color;  
 2 import java.awt.image.BufferedImage;  
 3 import java.util.Arrays;  
 4 import java.util.Random;  
 5   
 6 /\*  
 7 Currently hardcoding this to be a (2,2)-EVCS  
 8 \*/  
 9 public class ExtendedVCS  
 10 {  
 11 private int k;  
 12 private int n;  
 13 private int imgWidth;  
 14 private int imgHeight;  
 15 private int numColumns;  
 16 private BufferedImage secretMsg;  
 17 private BufferedImage[] innocentShares;  
 18 //private int[2][] shareOrigRGBPixels;  
 19 private int[][] encryptedShareRGB;  
 20   
 21 private int numSharesToDecrypt;  
 22 private BufferedImage[] sharesToDecrypt;  
 23 private int[] secretMsgPixels;  
 24   
 25 //Matrices  
 26 int[][] wwSw = new int[][]{  
 27 {1, 0, 0, 1},  
 28 {1, 0, 0, 0} };  
 29 int[][] wwSb = new int[][]{  
 30 {1, 0, 0, 1},  
 31 {0, 1, 1, 0} };  
 32 int[][] wbSw = new int [][]{  
 33 {1, 0, 0, 1},  
 34 {1, 0, 1, 1} };  
 35 int[][] wbSb = new int [][]{  
 36 {1, 0, 0, 1},  
 37 {0, 1, 1, 1} };  
 38 int[][] bwSw = new int [][]{  
 39 {1, 0, 1, 1},  
 40 {1, 0, 1, 0} };  
 41 int[][] bwSb = new int [][]{  
 42 {1, 0, 1, 1},  
 43 {0, 1, 1, 0} };  
 44 int[][] bbSw = new int [][]{  
 45 {1, 0, 1, 1},  
 46 {1, 0, 1, 1} };  
 47 int[][] bbSb = new int [][]{  
 48 {1, 0, 1, 1},  
 49 {0, 1, 1, 1} };  
 50   
 51   
 52 //For encryption purposes  
 53 ExtendedVCS(BufferedImage secretMsgIn, BufferedImage[] innocentSharesIn)  
 54 {  
 55 k = 2;  
 56 n = 2;  
 57 secretMsg = secretMsgIn;  
 58 imgWidth = secretMsg.getWidth();  
 59 imgHeight = secretMsg.getHeight();  
 60 innocentShares = innocentSharesIn;  
 61 }  
 62   
 63 //For decryption purposes  
 64 ExtendedVCS(BufferedImage[] shareImgs)  
 65 {  
 66 numSharesToDecrypt = 2;  
 67 sharesToDecrypt = shareImgs;  
 68 imgWidth = shareImgs[0].getWidth();  
 69 imgHeight = shareImgs[0].getHeight();  
 70 }  
 71   
 72 int getImgWidth()  
 73 {  
 74 return imgWidth;  
 75 }  
 76   
 77 int getImgHeight()  
 78 {  
 79 return imgHeight;  
 80 }  
 81   
 82 int[][] getRGBPixelsForShares()  
 83 {  
 84 return encryptedShareRGB;  
 85 }  
 86   
 87 int[] getDecryptImgPixels()  
 88 {  
 89 return secretMsgPixels;  
 90 }  
 91   
 92 void encryptImage()  
 93 {  
 94 int[] secretRGB = new int[imgWidth \* imgHeight];  
 95 int[][] shareOrigRGB = new int[2][imgWidth \* imgHeight];  
 96 secretMsg.getRGB(0, 0, imgWidth, imgHeight, secretRGB, 0, imgWidth);  
 97 innocentShares[0].getRGB(0, 0, imgWidth, imgHeight, shareOrigRGB[0], 0, imgWidth);  
 98 innocentShares[1].getRGB(0, 0, imgWidth, imgHeight, shareOrigRGB[1], 0, imgWidth);  
 99 createPixelsOfShares(secretRGB, shareOrigRGB);  
100 }  
101   
102 void createPixelsOfShares(int[] secretImgRGB, int[][] shareOriginalRGB)  
103 {  
104 encryptedShareRGB = new int[2][imgWidth \* imgHeight];  
105   
106 for(int i = 0; i < secretImgRGB.length; i++)  
107 {  
108 int redVal = (secretImgRGB[i] & 0x00ff0000) >> 16;  
109 int greenVal = (secretImgRGB[i] & 0x0000ff00) >> 8;  
110 int blueVal = (secretImgRGB[i] & 0x000000ff);  
111 PixelVCS orig = new PixelVCS(redVal, greenVal, blueVal);  
112   
113 redVal = (shareOriginalRGB[0][i] & 0x00ff0000) >> 16;  
114 greenVal = (shareOriginalRGB[0][i] & 0x0000ff00) >> 8;  
115 blueVal = (shareOriginalRGB[0][i] & 0x000000ff);  
116 PixelVCS innocent0 = new PixelVCS(redVal, greenVal, blueVal);  
117   
118 redVal = (shareOriginalRGB[1][i] & 0x00ff0000) >> 16;  
119 greenVal = (shareOriginalRGB[1][i] & 0x0000ff00) >> 8;  
120 blueVal = (shareOriginalRGB[1][i] & 0x000000ff);  
121 PixelVCS innocent1 = new PixelVCS(redVal, greenVal, blueVal);  
122   
123 Random randomGen = new Random();  
124 int randomColumn = randomGen.nextInt(4);  
125   
126 //If pixel is white  
127 if(innocent0.isMoreWhiteThanBlack())  
128 {   
129 if(innocent1.isMoreWhiteThanBlack())  
130 {  
131 if(orig.isMoreWhiteThanBlack())  
132 {  
133 //Want to use matrix wwSw  
134 if(wwSw[0][randomColumn] == 0)  
135 encryptedShareRGB[0][i] = Color.WHITE.getRGB();  
136 else  
137 encryptedShareRGB[0][i] = Color.BLACK.getRGB();  
138   
139 if(wwSw[1][randomColumn] == 0)  
140 encryptedShareRGB[1][i] = Color.WHITE.getRGB();  
141 else  
142 encryptedShareRGB[1][i] = Color.BLACK.getRGB();  
143 }  
144 else  
145 {  
146 //Want to use matrix wwSb  
147 if(wwSb[0][randomColumn] == 0)  
148 encryptedShareRGB[0][i] = Color.WHITE.getRGB();  
149 else  
150 encryptedShareRGB[0][i] = Color.BLACK.getRGB();  
151   
152 if(wwSb[1][randomColumn] == 0)  
153 encryptedShareRGB[1][i] = Color.WHITE.getRGB();  
154 else  
155 encryptedShareRGB[1][i] = Color.BLACK.getRGB();  
156 }  
157 }  
158 else  
159 {  
160 if(orig.isMoreWhiteThanBlack())  
161 {  
162 //Want to use matrix wbSw  
163 if(wbSw[0][randomColumn] == 0)  
164 encryptedShareRGB[0][i] = Color.WHITE.getRGB();  
165 else  
166 encryptedShareRGB[0][i] = Color.BLACK.getRGB();  
167   
168 if(wbSw[1][randomColumn] == 0)  
169 encryptedShareRGB[1][i] = Color.WHITE.getRGB();  
170 else  
171 encryptedShareRGB[1][i] = Color.BLACK.getRGB();  
172 }  
173 else  
174 {  
175 //Want to use matrix wbSb  
176 if(wbSb[0][randomColumn] == 0)  
177 encryptedShareRGB[0][i] = Color.WHITE.getRGB();  
178 else  
179 encryptedShareRGB[0][i] = Color.BLACK.getRGB();  
180   
181 if(wbSb[1][randomColumn] == 0)  
182 encryptedShareRGB[1][i] = Color.WHITE.getRGB();  
183 else  
184 encryptedShareRGB[1][i] = Color.BLACK.getRGB();  
185 }  
186 }  
187 }  
188 else  
189 {  
190 if(innocent1.isMoreWhiteThanBlack())  
191 {  
192 if(orig.isMoreWhiteThanBlack())  
193 {  
194 //Want to use matrix bwSw  
195 if(bwSw[0][randomColumn] == 0)  
196 encryptedShareRGB[0][i] = Color.WHITE.getRGB();  
197 else  
198 encryptedShareRGB[0][i] = Color.BLACK.getRGB();  
199   
200 if(bwSw[1][randomColumn] == 0)  
201 encryptedShareRGB[1][i] = Color.WHITE.getRGB();  
202 else  
203 encryptedShareRGB[1][i] = Color.BLACK.getRGB();  
204 }  
205 else  
206 {  
207 //Want to use matrix bwSb  
208 if(bwSb[0][randomColumn] == 0)  
209 encryptedShareRGB[0][i] = Color.WHITE.getRGB();  
210 else  
211 encryptedShareRGB[0][i] = Color.BLACK.getRGB();  
212   
213 if(bwSb[1][randomColumn] == 0)  
214 encryptedShareRGB[1][i] = Color.WHITE.getRGB();  
215 else  
216 encryptedShareRGB[1][i] = Color.BLACK.getRGB();  
217 }  
218 }  
219 else  
220 {  
221 if(orig.isMoreWhiteThanBlack())  
222 {  
223 //Want to use matrix bbSw  
224 if(bbSw[0][randomColumn] == 0)  
225 encryptedShareRGB[0][i] = Color.WHITE.getRGB();  
226 else  
227 encryptedShareRGB[0][i] = Color.BLACK.getRGB();  
228   
229 if(bbSw[1][randomColumn] == 0)  
230 encryptedShareRGB[1][i] = Color.WHITE.getRGB();  
231 else  
232 encryptedShareRGB[1][i] = Color.BLACK.getRGB();  
233 }  
234 else  
235 {  
236 //Want to use matrix bbSb  
237 if(bbSb[0][randomColumn] == 0)  
238 encryptedShareRGB[0][i] = Color.WHITE.getRGB();  
239 else  
240 encryptedShareRGB[0][i] = Color.BLACK.getRGB();  
241   
242 if(bbSb[1][randomColumn] == 0)  
243 encryptedShareRGB[1][i] = Color.WHITE.getRGB();  
244 else  
245 encryptedShareRGB[1][i] = Color.BLACK.getRGB();  
246 }  
247 }  
248 }  
249 }  
250 }  
251   
252 void decryptImage()  
253 {  
254 //Make a 2d array of pixel arrays  
255 int[][] pixelsToCompare = new int[numSharesToDecrypt][imgWidth \* imgHeight];  
256 secretMsgPixels = new int[imgWidth \* imgHeight];  
257   
258 //getRGB pixels of BufferedImages  
259 for(int i = 0; i < numSharesToDecrypt; i++)  
260 {  
261 sharesToDecrypt[i].getRGB(0, 0, imgWidth, imgHeight, pixelsToCompare[i], 0, imgWidth);  
262 }  
263   
264 //Logical OR pixel with all three share values  
265 int numOfPixels = pixelsToCompare[0].length;  
266 for(int i = 0; i < numOfPixels; i++)  
267 {  
268 int pixelColor = 0;  
269 for(int j = 0; j < numSharesToDecrypt; j++)  
270 {  
271 if(pixelsToCompare[j][i] == Color.WHITE.getRGB())  
272 {  
273 pixelColor = pixelColor | 0;  
274 }  
275 else  
276 {  
277 pixelColor = pixelColor | 1;  
278 }  
279 }  
280   
281 //Store the result in an array after converting to WHITE and BLACK  
282 if(pixelColor == 1)  
283 {  
284 secretMsgPixels[i] = Color.BLACK.getRGB();  
285 }  
286 else  
287 {  
288 secretMsgPixels[i] = Color.WHITE.getRGB();  
289 }  
290 }  
291 }  
292   
293 }

PixelVCS.java

1 import java.awt.Color;  
 2   
 3 public class PixelVCS  
 4 {  
 5 private int redVal;  
 6 private int greenVal;  
 7 private int blueVal;  
 8   
 9 public PixelVCS(int redIn, int greenIn, int blueIn)  
10 {  
11 redVal = redIn;  
12 greenVal = greenIn;  
13 blueVal = blueIn;  
14 }  
15   
16 //Used to determine if pixel is closer to white than black  
17 public boolean isMoreWhiteThanBlack()  
18 {  
19 int sum = redVal + greenVal + blueVal;  
20 int avg = sum / 3;  
21 return (avg >= 128);  
22 }  
23 }