Ira A. Fulton Schools of Engineering
Arizona State University

CSE 539: Applied Cryptography

Steganography and Cryptanalysis Project

Purpose

The goal of this project is to gain experience with steganography, cryptanalysis, and random number generators. You will work at the byte level of files to create a bitmap image from scratch. You will programmatically modify such a file, hiding a message inside the image. In the second part, you will break an encryption by finding the key through a weakness in the random number generation.

Objectives

Students will be able to:

- Apply number theory concepts to solve real-world problems.
- Apply algorithms for random number generation, hash functions and encryption.
- Analyze encryption and hash algorithms to find weaknesses and attacks.

Technology Requirements

Technology requirements will be found on the class GitHub README (Click or copy the link below)

https://github.com/GiveThanksAlways/CSE-539-Applied-Cryptography-2021-Fall-B

Directions

Part 1 - Steganography

With a hex editor, you can manually create a bitmap (.bmp) file by typing in the bytes. Using either a downloaded hex editor or one you find online, create a bitmap using these bytes:

> 42 4D 4C 00 00 00 00 00 00 00 1A 00 00 00 0C 00 00 00 04 00 04 00 01 00 18 00 00 00 FF FF FF FF 00 00 FF FF FF FF FF FF FF 00 00 00 FF FF FF 00 00 00 FF 00 00 FF FF FF FF 00 00 FF FF FF FF FF FF 00 00 00 FF FF FF 00 00 00

If you save your hex file as a .bmp file, you can view your image. It will only be 4x4 pixels large, so you will need to zoom in. Figure 1 shows the expected image:

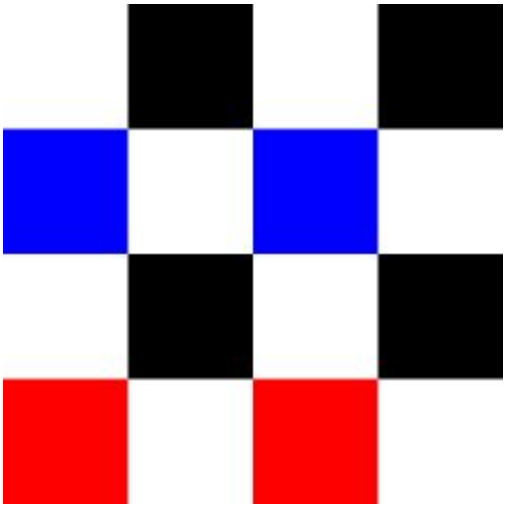

Figure 1: Manually created bitmap

In order to perform steganography, you must first understand the bmp file structure. The first 26 bytes are header bytes (i.e., the magic numbers). We cannot easily hide data inside a file header without potentially corrupting the file, especially with more complex file headers such as png.

First, there are some general header bytes: 42 4D 4C 00 00 00 00 00 00 00 1A 00 00 00 0C 00 00 00

The next two bytes specify the width (note that files are written in little endian order - each byte is in expected order, but the order of bytes is opposite what we would expect): 04 00

The next two bytes specify the height: 04 00

Finally there are four more header bytes: 01 00 18 00

Starting after the header bytes, every set of three bytes forms one pixel. Each of these sets contains the color representation in BGR (Blue Green Red) format. For example, the color blue is represented by FF 00 00. Also note that colors are defined from left to right and bottom to top.

Let us consider each of the bytes as being part of a Galois Field. In other words, we can treat each byte as a polynomial. Consider the byte 0xB4. This byte represents the magnitude of either the Blue, Green, or Red of a color. In binary, this byte is: 10110100. As part of $GF(2^7)$ (i.e., the Galois Field of 2^7), we can treat this byte as a polynomial of the form:

$$1*x^7 + 0*x^6 + 1*x^5 + 1*x^4 + 0*x^3 + 1*x^2 + 0*x^1 + 0*x^0$$

Since $O(x^7) >> O(x^1)$, we can modify the bits corresponding to x^1 and x^0 with a negligible effect on that color. Try this yourself. For example, FF = 11111111. We can change the last two bits 11111100 = FC. Change some of the bytes in this way, resave as a .bmp, and compare it to the original to see if you can tell the difference.

Here is a piece of a program that will print out the bytes of the bitmap discussed earlier (you can start with this code for this part of the assignment):

```
byte[] bmpBytes = new byte[] {
   0x42,0x4D,0x4C,0x00,0x00,0x00,0x00,0x00,
   0x00,0x00,0x1A,0x00,0x00,0x00,0x0C,0x00,
   0x00,0x00,0x04,0x00,0x04,0x00,0x01,0x00,
   0x18,0x00,0x00,0x00,0xFF,0xFF,0xFF,0xFF,
   0x00,0x00,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
   0xFF,0x00,0x00,0x00,0xFF,0xFF,0xFF,0x00,
   0x00,0x00,0xFF,0x00,0x00,0xFF,0xFF,0xFF,
   0xFF,0x00,0x00,0xFF,0xFF,0xFF,0xFF,0xFF,
   0xFF,0x00,0x00,0x00,0xFF,0xFF,0xFF,0x00,
   0x00,0x00
};

Console.WriteLine(BitConverter.ToString(bmpBytes).Replace("-", " "));
```

From the command line (not from keyboard input), you will receive a string of 12 hexadecimal digits (e.g., B1 FF FF CC 98 80 09 EA 04 48 7E C9). Since our image has 16 pixels, there are 3*16 = 48 color bytes. Hiding 2 bits per byte allows us to hide 48 * 2 = 96 bits / 8 bits/byte = 12 bytes of data.

Modify your image to hide those 12 bytes in the color bytes and print out the result in the same format as the sample code does. For example, suppose the first byte is B1. In binary, this is 10110001. We hide the values using the XOR operator. The first 4 color bytes in the above image are 00 00 FF FF (you must start after the header ends — note that the header ends at

0x00, not 0x18). In binary, these bytes are 00000000 00000000 11111111 11111111. Taking two bits at a time, we XOR with each byte. We start with bits 10, then 11, 00, and finally 01. Thus, our 4 bytes become 00000010 00000011 11111111 11111110.

Hint: You can convert a string such as “F8” to a byte using Convert.ToByte(“F8”, 16). 16 represents the base you are converting from.

Using the sample input from above, here is the command to run the program and the expected output:

```
dotnet run "B1 FF FF CC 98 80 09 EA 04 48 7E C9"

42 4D 4C 00 00 00 00 00 00 00 1A 00 00 00 0C 00 00 00 04 00 04 00 01 00 18 00 02 03
FF FE FC FC 03 03 FC FC FC FC FC FF FC 00 02 01 FD FF FD 00 00 00 FF 00 02 FE
FC FD FD 02 00 FF FE FF FE FF FD 00 01 03 FC FD FC 00 02 01
```

Figure 2 shows what this bitmap looks like. Despite hiding a secret message in it, the change is impossible to see:

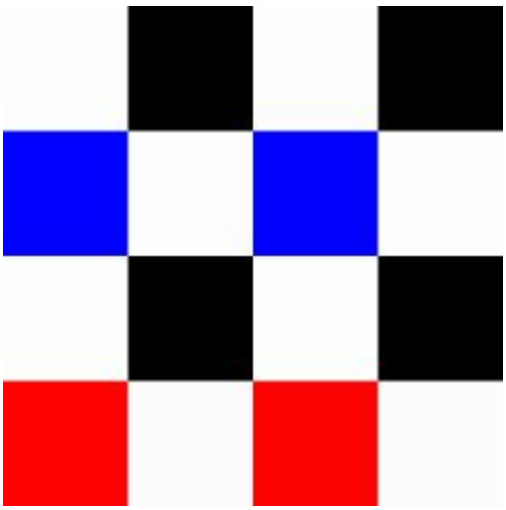

Figure 2: Bitmap with secret message

Part 2 - Cryptanalysis

There are two general types of random number generators. The first is the type that ships with most programming languages, a pseudo-random number generator (PRNG). The second is a more secure type, the cryptographically secure pseudo-random number generator (CSPRNG).

Typically, a PRNG is a function that generates a series of seemingly random numbers. The starting point for this calculation is given by the seed. Most languages do not require a seed to be passed in to a call for a random number. Some languages will use a default seed in this case (e.g., C uses 1 as the default seed: rand(3): pseudo-random number generator - Linux man

page). Other languages will seed the PRNG with a non-static seed (e.g., C#: Random Class (System)).

Experiment with the C# Random class. Given the same seed, note that the same series of numbers will be generated regardless of how many times you run your program.

For this exercise, you will break an encryption by finding the key through a weakness in the random number generation.

Scenario: Suppose you are able to access your friend's computer and find that they created an encryption program. Their key is generated randomly, but they chose to use the current time as the seed. By looking at the file properties, you find that the program was run at some point between 7/3/2020 (July 3) 11:00:00.000 and 7/4/2020 (July 4) 11:00:00.000 (1 day later).

Here is the code your friend used to perform the conversion and encryption:

```
DateTime dt = DateTime.Now;
TimeSpan ts = dt.Subtract(new DateTime(1970, 1, 1));

string secretString;
Random rng = new Random((int)ts.TotalMinutes);
byte[] key = BitConverter.GetBytes(rng.NextDouble());

Console.WriteLine(Encrypt(key, secretString));

private static string Encrypt(byte[] key, string secretString)
{
   DESCryptoServiceProvider csp = new DESCryptoServiceProvider();
   MemoryStream ms = new MemoryStream();
   CryptoStream cs = new CryptoStream(ms,
      csp.CreateEncryptor(key, key), CryptoStreamMode.Write);
   StreamWriter sw = new StreamWriter(cs);
   sw.Write(secretString);
   sw.Flush();
   cs.FlushFinalBlock();
   sw.Flush();
   return Convert.ToBase64String(ms.GetBuffer(), 0, (int)ms.Length);
}
```

Hint: Casting TotalMinutes to an int truncates the value. Thus, 7/3/2020 11:03:04.123 produces the same seed as 7/3/2020 11:03:00.000.

In a real world example, you may not have access to the plaintext directly. In a file example, you could use knowledge about the file headers to find the key. In World War II, the Germans made

use of their Enigma Cipher to encrypt radio traffic, such as reports and other military intelligence. In some of these reports, part of the plaintext was known; simple common phrases such as “nothing to report” could be inferred as being part of the radio traffic according to message length and similarity to other messages. This small part of the plaintext enabled the Allied forces to decrypt the messages through a key-finding algorithm. Similarly, you will assume that you have a phrase of the plaintext and will use it to find the key.

In this project, the plaintext and ciphertext will be passed to you as command line arguments. Your program should output the integer value used to seed the random number generator that your friend used for that plaintext and ciphertext.

Note: You will be outputting the seed, not the key. The seed will be shorter and will simplify testing and debugging. You can always get the key using the seed.

Below is a sample command and expected output:

```
dotnet run “Hello World” “RgdIKNgHn2Wg7jXwAykTlA==”

26564295
```

In this example, the correct seed corresponded to 7/4/2020 10:15:00.000.

Submission Directions for Project Deliverables

Upload your Program.cs file to Gradescope.

You will upload one Program.cs file for P1_1 and one Program.cs file for P1_2 (so two Program.cs files, one for each part of the project). The autograder will run your code with different inputs and evaluate the output of the "P1_1()" function or the "P1_2()" function.

These functions (P1_1() and P1_2()) are provided in the boilerplate code on GitHub for each part of the project.