CSE 539 MCS Portfolio Inclusion Report

Kevin Medina

School of Computing and Augmented Intelligence
Arizona State University
Tempe, AZ 85281
Kmmedin4@asu.edu

Index Terms - Steganography, Cryptanalysis, Applied Cryptography, RSA, Diffie-Hellman.

I. INTRODUCTION

The projects in CSE539 Applied Cryptography entailed learning and applying various topics in the realm of applied cryptography using the C# language along with .NET 6 using the command line interface. For this course, there were four projects that were auto graded. The first project had two separate parts. Part one was about performing steganography, the practice of hiding some secret message in another medium that is public such as an image. The steganography was performed on provided bitmap bytes using unknown hexadecimal values supplied from the command line command argument options. Part two was performing cryptanalysis on the pseudo random number generator Random class by finding a specific seed in C# that was used to produce a key for DES encryption of plaintext to ciphertext. The plaintext and ciphertext were provided through the command line and unknown before execution. The second project entailed finding hash collisions using the MD5 hashing algorithm. The third project involved using Diffie-Hellman algorithm to encrypt and decrypt provided text via the command line. The fourth project required encryption and decryption of provided text using the RSA algorithm also via the command line. All these projects were completed using Visual Studio Code in Linux Ubuntu 20.04.

II. SOLUTIONS

A. Steganography and Cryptanalysis Project

The solution to the steganography portion consisted of looping through the 12 hexadecimal digits, provided via the command line, and applying the XOR bitwise operator with the hexadecimal digit and the provided bitmap bytes. The foreach loop first converts the provided hexadecimal value to integer form, then binary form, and pads the binary number in case it does not contain 8 digits. The 8 digits are necessary in the binary value as it plays a crucial role in the XOR operation that is performed bit by bit with the bitmap bytes. There is then a for loop in the foreach loop that iterates through each single hexadecimal digit using a counter that increments after each iteration. When it iterates through a hexadecimal digit, it performs the XOR bitwise operation of the provided hex digit and a selected bitmap byte value. After this is done, the produced byte value is added to a list which is used to store the changed bytes of the bitmap. The use of two counters allows the process of iterating through the bitmap bytes after performing the XOR computation until all the bitmap bytes have been changed. This can be seen in Fig. 1. Finally, the changes bitmap bytes are printed to show that they are slightly different from the originals.

```
for (i = 1;i<bitstring.Length;i++)
{
        if (i % 2 != 0)
        {
          char x = bitstring[i];  //saving first letter of bit string from hex digits inputted in
          char y = bitstring[i-1];  // same as above but second bit//
          string x1 = Char.ToString(x);
          string y1 = Char.ToString(y);
          string yx = y1 + x1; //2 bits of hex digits combined, 4 different sets total//
          int inputbyte = Convert.ToByte(yx,2); // convert the two combined chars to bytes//
          var selected = bmpBytes.Skip(place).Take(4).ToArray(); //selecting 4 bytes from bitmap
          var selectedbyte = selected.GetValue(startc);
          int bitmapvalue = Convert.ToInt32(selectedbyte);
          byte result = (byte)( bitmapvalue ^ inputbyte ); // bitmap value - provided bitmap, inp
          byte[] result1 = new byte[] {result};
          list.AddRange(result1);  // add changed byte to byte list//
          startc=startc + 1; // iterate through provided hex digits//
        }
}
place = place + 4; // iterate through bitmap bytes//
```

Fig .1. For loop used to XOR provided hexadecimals values with bitmap

The Cryptanalysis part involved using the Random class in C# to find a specific seed that was used to create a key for DES encryption. The seed that was used to create the key was generated using the DateTime class of C#. It is the current time, in seconds since a specified time and date, with minute precision. Per the project instructions, the example solution shows that the seed is an integer. Through debugging of the supplied code, it was found this integer represents the specific time that the seed was used for key generation. In the instructions, it specifies that the window for when the seed was used is only one day. Thus, the algorithm to find the seed is simply to iterate through each minute of that specific day until the seed produces an encrypted text that matches the provided encrypted text using the provided plaintext. These steps can be seen in Fig.2. Once they match, the seed is displayed in the output. If no seed is found, that means there the seed was not generated during the timeframe specified.

```
Random rng = new Random((int)timevalue); // Generate random value from ts//
byte[] key = BitConverter.GetBytes(rng.NextDouble()); // key for DES//

for (i = 0;i<=1440;i++) // 1440 = minutes in a day
{
    rng = new Random((int)timevalue);
    key = BitConverter.GetBytes(rng.NextDouble());
    if (encryString == Encrypt(key,secretString)) // checking if provided encrypted text matches
    {
        Console.WriteLine(timevalue); // print seedd//
        break;
    }
    else
    {
        timevalue = timevalue + 1;  // iterate until entire day is complete//
    }
}
```

Fig .2. For loop used to find seed value used for encryption

B. *Hash Project*

This project's solution focused on finding two text strings, with a salt added at the end, that produces collisions through a birthday attack using the MD5 hash algorithm with a command line argument of a one-byte salt. The salt is added, in byte form, to the text strings and then the entire byte array of the text is hashed. The text is undefined and must only contain alphanumeric characters, per the instructions. Following the example solution provided, I limited the solution text to 10 characters. First, I created a dictionary with the keys being strings consisting of 10 random alphanumeric characters. The corresponding values to the keys of the dictionary were the first two characters of the MD5 hash values of the strings with the salt byte appended. For computational time concerns and per the instructions, only the first five bytes needed to match so I only saved the first two characters of the hash, since each character is four bits in the MD5 digest size [1]. The creation of the dictionary is shown in Fig.3. The function Randomstring() is a function that creates a random 10 alphanumeric character string. I set the number of entries in the dictionary to one million. Then I used the GroupBy method for dictionaries in C# to find two values, MD5 hashes of the random strings with the salt, that were identical in different entries. Finally, the two random 10 character strings that produced identical first five bytes in the MD5 hash value are printed in the terminal.

```
IDictionary<string,string> valuesHash = new Dictionary<string,string>();
for (i = 0; i <=1000000; i++) // i - number of times to iterate to find collisions//
{
    randomstring1 = Randomstring(10);
    byte[] newb1 = Encoding.UTF8.GetBytes(randomstring1);
    List<byte> list = new List<byte>();
    list.AddRange(newb1);
    list.AddRange(inSaltb);
    byte[] myarrayb;
    myarrayb = list.ToArray();
    resultbytes1 = CreateMD5(myarrayb);
    string rb1 = resultbytes1.Substring(0,2);
    valuesHash.Add(randomstring1, rb1);
}

var lookup = valuesHash.GroupBy(x => x.Value).Where(x => x.Count() > 1);
```

Fig .3. For loop used to create dictionary of random strings and hashes

C. *Diffie-Hellman and Encryption*

This project's solution involved calculating and using the key to utilize the Diffie-Hellman key exchange protocol. It also required encryption and decryption to be performed using AES. Multiple command arguments were supplied via the command line interface. From these variables and the instructions provided, it was possible to calculate the modulus N, generator g, and the key g^{xy}modN. I used the BigInteger.Parse method to convert the provided inputs to integer form. Then I used the BigInteger.Pow and BigInteger.ModPow methods to perform the calculations necessary to obtain the key. The crucial calculation of the key used the .ModPow [2] method with the provided g^{y}modN, x, and the calculated N. After, I used the encryption and decryptions functions for AES from [3]. The format of inputs for these functions was the same for both, text, key, and the Initialization Vector. All the inputs provided were converted to byte structure(byte[]) form as the encryption and decryption required this. After the encryption and decryption was done, the required plaintext and ciphertext are printed to the terminal. Fig. 4 shows how I decrypted and encrypted the texts after calculating the key – g^{xy}modN.

```
BigInteger Key = BigInteger.ModPow(bigGymodN,x,bigN); // Key - gxymodN//
byte[] KeyBytes = Key.ToByteArray();

foreach (var word in IVwords)
{
    byte newb = Convert.ToByte(word,16);
    byte[] result1 = new byte[] {newb};
    IVlist.AddRange(result1);
}
byte[] IVbytes = IVlist.ToArray();  // IV for encryption/decryption//

foreach (var word in cipherwords) // creating list from cipher//
{
    byte newb = Convert.ToByte(word,16);
    byte[] result1 = new byte[] {newb};
    cipherlist.AddRange(result1);
}
byte[] cipherbytes = cipherlist.ToArray(); // converting list to bytes//
string plaintext;
plaintext = Decrypt(cipherbytes, KeyBytes, IVbytes);
byte[] encryResults;
encryResults = Encrypt(plaint, KeyBytes, IVbytes);
Console.WriteLine(plaintext+","+BitConverter.ToString(encryResults).Replace("-"," "));
```

Fig .4. Code for encrypting and decrypting texts after calculating the key

D. *RSA Project*

The solution to this project was similar to the solution for the Diffie-Hellman project. The algorithm to solve for *p* and *q*, both immense prime numbers, was like solving for N and g with the BigInteger Class and its methods. Then, I calculated the modulus N, and $\varphi(N)$, which is coprime to the *e*. The value of *e* was provided, so I then implemented the Extended Euclidean algorithm to find *d*. I did this by using the provided equation,

$$e \bullet d \bmod(\varphi(N)) = 1 \tag{1}$$

and substituting *d* with $(\varphi(N) \bullet \text{counter} + 1) / e$ so that it turns into

$$e \bullet (\varphi(N) \bullet \text{counter} + 1) / e) \bmod(\varphi(N)) = 1 \tag{2}$$

which then leaves the equation in the following form.

$$(\varphi(N) \bullet \text{counter} + 1) \bmod(\varphi(N)) = 1 \tag{3}$$

Because we already know the variable *e*, we can use it in attempting to find d, (2). One can notice that for (3), it will always be true because any multiple of the $\varphi(N)$ plus one will always yield one when modulus of $\varphi(N)$ is applied. Through debugging, I found I needed to do a million iterations of

$$d = (\varphi(N) \bullet \text{counter} + 1) / e \tag{4}$$

to find a single value *d* that satisfies (1). I accomplished this through using a for loop that iterated a million times by increasing the *counter* by one after each iteration. During each iteration, I checked to see if (1) was valid for the current *d*, if

not I continued until the loop found a match. If no d was found, I would increase the number of iterations to be executed.

Once *d* was found and verified, the encryption and decryption were performed using the BigInteger.ModPow method [3]. The encryption and decryption inputs were text, plain for encryption or cipher for decryption, along with the values *e* for encryption and *d* for decryption, and finally the modulus N. The obtained encrypted and decrypted texts were then outputted to the terminal. These steps can be seen in

```
BigInteger d;
BigInteger d2;
BigInteger FinalD;
FinalD = 0;
for (int i = 0;i<1000000;i++)
{   d = ( (phiN * i) + 1) / e ; // trying to find coprime of e//
    d2 = (e * d) % phiN; // calculating modulus arthimetic//
    if (d2 == 1)
    {
        FinalD = d; // d such that e*d mod(phiN) = 1//
    }
}
BigInteger Decrypted;
Decrypted = BigInteger.ModPow(cipher_, FinalD,N);
BigInteger Encrypted;
Encrypted = BigInteger.ModPow(plain_, e,N);
Console.WriteLine(Decrypted+","+Encrypted);
```

Fig .5. Code to find coprime of e and then decrypt and encrypt using RSA

III. Results

A. *Steganography and Cryptanalysis Project*

My results for the steganography part of the project were the final bytes of the bitmap after hiding the 12 bytes via the XOR bitwise function with the original bitmap bytes and the provided hexadecimal digits. For my submission online, the 12 hexadecimal digits provided were 8E 14 83 9B 23 80 AD 19 FE D6 7C 17. These hex values produced correct final byte values of 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 00 FC FD FF FE 01 00 FD FF FF FC FD FE FD 03 00 02 FF FC FD 00 00 00 FD 02 03 FE FF FE FD 01 03 FC FC FD FC FE FE 02 01 03 FC FF FF 01 01 03. For comparison, the original bytes were 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 FF 00 00 00 FF FF FF 00 00 00. One can see that the changed bitmap bytes are slightly different from the original bytes when comparing each hex value to its corresponding past value.

For the Cryptanalysis part, my provided plaintext of “gyYCT7yC” with provided ciphertext of “pOxKhUkTDbM++ZuREAlaFQ==” yielded the correct answer of 26562900. Thus, knowing when someone creates a seed using a random number generator, such as C#’s Random class, is a huge vulnerability that cannot be overlooked. It also shows the Random class is not a cryptographically secure random number generator [4].

B. *Hash Project*

My results for the project were two alphanumeric strings that resulted in the same first five bytes. The first string was “F087RNOvdz” while the second was “zx589cP51v”. I was unable to double check these answers as the auto grader feedback did not provide the salt added before hashing. Also, these results were not consistent even with the same salt inputted during execution. This is due to the fact that the generation of entries in the dictionary was not deterministic so each solution was different to the previous.

C. *Diffie-Hellman and Encryption*

My results for the Diffie-Hellman key exchange passed the auto-grader with the decrypted and ciphertext matching up exactly. The decrypted text was “wiYRXM2beb03OV5uv6V7UooyOsYdML3RgHw4PXjpW BK2r” while the ciphertext was “FB 18 59 BF 4B 6B 56 DC 45 10 D3 C1 A3 72 E8 17 14 72 05 40 CD 16 19 26 C2 EE FA 91 CF 49 8F FD F7 F8 01 2E 81 D8 A4 B1 EF 11 2A 58 53 39 EA 02”. Once again, I am unable to verify because the auto-grader does not provide what the command line arguments are at the time of execution.

D. *RSA Project*

My results for this project were decrypted text of “1876215194” and cipher text of “12014029850127265823439329267361831533113828936398141365883411300244398694335412009031294228248983673089960972865829779087236991294761618450840269993 9101”. They matched the results of the auto grader. The most difficult aspect of achieving the correct result was knowing how many iterations would be needed to find a coprime of *e* while trying to not waste too much computational power as technically there is an unlimited amount of coprimes out there for *e*. Thus, I could put a billion, or more iterations which would likely take too long for the auto grader to correctly analyse the results.

I independently contributed to all the results of the projects with minimal guidance as all the projects were individually assigned.

IV. Skills, Techniques, Knowledge Acquired

A. *C# Programming Language Knowledge*

Due to these projects only being able to be written in the C# programming language for auto the auto grader, much hands-on debugging and testing experience was gained. Much of the experience involved knowing which form a variable was in and what form it needed to be for it to be used for computation or for printing to the terminal. For example, when parsing the input of the very large prime numbers into the BigInteger form for the RSA or the Diffie-Hellman projects, I had to first convert the input from the terminal into an integer using the Int32.Parse() method. Then I was able to convert it into BigInteger form. Writing and debugging these coding

projects expanded my knowledge of how the C# language can perform calculations with large numbers that are not able to be performed manually, that is by hand, or even with the regular integer form in C#. I also learned how to manually manipulate byes, perform XOR bitwise operations [5], and convert hexadecimal digits to bytes or strings within C#. In the Cryptanalysis part, I learned that the Random class in C# is not viable for using it as a seed for encryption. I now know what are some essential functions in the Systems.Security.Cryptography namespace by using them, mainly to use encryption and decryption functions. For the has project, I got hands on experience performing a birthday attack via a dictionary method utilizing C#. Much knowledge was also gained on how to handle multiple inputs from the command line and how to use them for computational purposes. For Diffie-Hellman, an understanding was gained on how to perform modular arithmetic in C# with the BigInteger class. This was also used in RSA, where I implemented the Extended Euclidean Algorithm in C# to find the correct solution.

All these projects were executed via the command line so therefore experience with the command line interface was gained. Specifically, how to use the .NET framework to execute programs from the command line with provided variables as command arguments. Also, how to debug via the command line when there is a misunderstanding of what type a variable is or if the provided inputs are being processed appropriately by the program.

B. Cryptographic Expertise

Expertise in cryptographic protocols was gained as part of the RSA and Diffie-Hellman projects. Number theory involved in these algorithms was applied to obtain the accurate response from the auto grader. These projects included calculating the correct keys for encryption and key exchange, which showed me how difficult it would be to attempt to crack these algorithms. I also was able to learn how one can attempt and maybe succeed in breaking the MD5 hash algorithm with colliding hashes. My competence on Man-In-the-Middle attacks also has increased as I can now see how easy it would be for a malicious entity to simply calculate the variables needed for Diffie-Hellman and use them to traverse communication between respective parties.

REFERENCES

[1] "MD5." Wikipedia.org. https://en.wikipedia.org/wiki/MD5 (May 6 2023).

[2] "BigInteger.ModPow(BigInteger, BigInteger, BigInteger) Method." Microsoft.com. https://learn.microsoft.com/en-us/dotnet/api/system.numerics.biginteger.modpow?view=net-8.0 (May 6 2023).

[3] M. Chand, "AES Encryption in C#." c-sharpcorner.com. https://www.c-sharpcorner.com/article/aes-encryption-in-c-sharp/ (May 7 2023).

[4] "Cryptographically secure pseudorandom number generator." Wikipedia.com. https://en.wikipedia.org/wiki/Cryptographically_secure__pseudorandom_number_generator (May 7 2023).

[5] "Bitwise operation." Wikipedia.com. https://en.wikipedia.org/wiki/Bitwise_operation (May 7 2023).