CSE 539: Applied Cryptography Portfolio

Shachi Shah
School of Computing and Augmented Intelligence
Arizona State University Online
Azusa, California, United States of America
spshah22@asu.edu

I. Introduction

Throughout this course of Applied Cryptography, I have completed four programming projects individually that related to a multitude of aspects of the course that is necessary to the topic of cryptography.

A. Project 1- Steganography and Cryptanalysis

1) Part 1 - Steganography

Steganography is the practice of concealing messages and/or information within other nonsecret media; such as videos, images, files, or even messages. The first project, implementing an application of steganography, introduced a Microsoft bitmap image file (.bmp). I was able to conceal a message into that .bmp file without having to alter the image.

2) Part 2 - Cryptanalysis

Cryptanalysis involves breaking encrypted messages by identifying weaknesses in the encryption without having access to the key [4]. In the second part of this project, I was provided with a plaintext, ciphertext, date, and time corresponding to when the ciphertext was generated. The encryption function used a pseudo-random number generator to create a key and encrypt the plaintext. My task was to implement cryptanalysis to crack the encryption by exploiting the pseudo-random number generator's behavior.

B. Project 2 - Hash

Hash functions are functions where the output values are all the same number of bits in length, especially one used to encrypt or compress data or to generate incidences [2]. Passwords are usually stored as hash values, or hashes. The stored hash values are then compared against the entered value hash in order to verify and authenticate the inputted value. “Salt” values are added to make the hashes more secure as they are random bits that make up the uniqueness in the hash. Salts help in protection from enabling attackers from recomputing hashes to passwords, such as rainbow table attacks.

Within this project, I was given a one-byte salt value and had to perform a birthday attack to find two strings with the same first five bytes of the MD5 hashing algorithm.

C. Project 3 - Diffie-Hellman and Encryption

The Diffie-Hellman key exchange is a digital encryption technique that allows two parties to securely exchange cryptographic keys over a public channel without transmitting their communication across the internet [1]. This process is often illustrated using the Alice and Bob analogy, as follows:

1) Alice and Bob agree on values for g and N, where N is a prime number and g is a primitive root modulo N.
2) Alice selects a secret integer x, then gives Bob $X = g^x \bmod N$.
3) Bob selects a secret integer y, then gives Alice $Y = g^y \bmod N$.
4) Alice calculates the shared secret s using the equation $s = Y^x \bmod N$.
5) Bob calculates the shared secret s using the equation $s = X^y \bmod N$.
6) Now, Alice and Bob have the same secret, s.

I was provided with g, N, x, and $g^y \bmod N$ (base 10), along with an encrypted text and its corresponding plaintext for this project. The objective was to compute the key, decrypt the given encrypted message, and then re-encrypted the decrypted message using 256-bit AES encryption.

D. Project 4 - RSA

The Rivest-Shamir-Adleman (RSA) encryption algorithm is an asymmetric encryption algorithm that is widely used in many products and services [3]. RSA is vital in cryptography because it allows users to secure messages before they send them. For my project, I was given p and q, a ciphertext, and a plaintext to implement the RSA algorithm. The goal similarly to the prior project, I was to calculate the key, decrypt the given ciphertext. and encrypt the given plaintext.

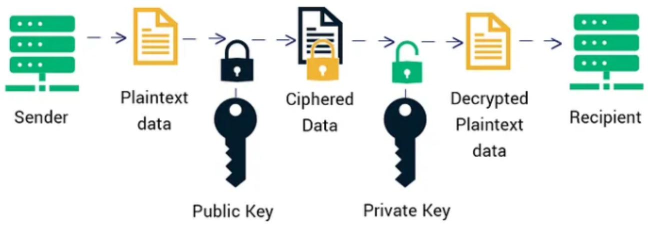

Fig. 1 - RSA Algorithm

The RSA Key generation works as follows (see Fig. 1):

1) Choose two large prime numbers: p and q.
2) Compute n by multiplying the two prime numbers to get n, where $n = p \times q$.
3) Calculate Euler’s Totient Function $\varphi(n)$, where it’s calculated as: $\varphi(n) = (p - 1) \times (q - 1)$.
4) Select an integer e where that $1 < e < 1\ \varphi(n)$ and $gcd(\varphi(n), e) = 1$.
5) Compute the decryption exponent d, using the extended euclidean algorithm such that $e \times d = 1 \bmod \varphi(n)$.
6) Now have the generated public key $\{d, n\}$ for encryption and the private key $\{n, d\}$ for decryption.

Then, in order to encrypt, you would have to represent the message as an integer such that $0 \leq m < n$ and encrypt the message with the formula $c = m^e \bmod n$. This would then transform the plaintext m into ciphertext c.

II. Explanations of Solutions

A. Project 1

1) Part 1 - Steganography

In this solution to the steganography problem, my goal was to hide or reveal data within a bitmap image by modifying its bytes. I started by reading the hidden data provided as a command-line argument, parsing it as hexadecimal values, and storing it in a byte array. The bitmap image bytes were predefined in the program, and I set the starting index to skip over the image header, ensuring that the metadata remained unaffected.

The main logic involved processing each byte of the hidden data. I extracted 2 bits at a time from each data byte and XORed them with the corresponding 4 bytes in the bitmap. This allowed me to subtly alter the image data, embedding the hidden message without causing significant visual changes to the image.

```
for (int i = 0; i < hiddenData.Length; i++) {
        byte dataByte = hiddenData[i];
        for (int j = 0; j < 4; j++) {
           byte bits = (byte)((dataByte >> (6 - 2 * j)) & 0x03);
           bmpBytes[startIndex + i * 4 + j] ^= bits;
        }
    }
```

Finally, I printed the modified bitmap bytes as a sequence of hexadecimal values, showing how data can be embedded in an image by changing its least significant bits. This approach demonstrates the use of bitwise operations in steganography for concealing data within an image.

2) Part 2 - Cryptanalysis

In this cryptanalysis solution, my objective was to perform a brute-force attack on an encrypted message to recover the secret key used for encryption. The approach leverages a known time window during which the encryption key was generated. My task was to iterate through all possible time-based keys within this window and attempt decryption to find the correct one.

I first set up the program to accept two command-line arguments: the plaintext (expected original message) and the ciphertext (the encrypted message). I then defined the start and end times for the encryption window, calculating the total number of minutes between these two points. This gave me a clear range of possible encryption keys to test.

```
 { …
DateTime startTime = new DateTime(2020, 7, 3, 11, 0, 0);
 DateTime endTime = new DateTime(2020, 7, 4, 11, 0, 0);
    TimeSpan timeSpan = endTime - startTime;
    int totalMinutes = (int)timeSpan.TotalMinutes;
    for (int i = 0; i <= totalMinutes; i++)
    {
      DateTime dt = startTime.AddMinutes(i);
      TimeSpan ts = dt.Subtract(new DateTime(1970, 1,
1));
       int seed = (int)ts.TotalMinutes;
       byte[] key = BitConverter.GetBytes(new
Random(seed).NextDouble());
// Attempt to decrypt the ciphertext using the generated key
if (Decrypt(ciphertext, key) == plaintext)
{
   // If decryption matches the plaintext, print the seed and
exit
    Console.WriteLine(seed);
     return;
}
```

The brute-force process involved iterating over every possible minute within the time frame. For each iteration, I calculated the total number of minutes since the Unix epoch (January 1, 1970) for the current time. I used this value as a seed to generate the encryption key, simulating how the original key might have been derived.

For each generated key, I attempted to decrypt the ciphertext. If the decryption result matched the provided plaintext, it meant I had successfully found the key, and I printed the seed used to generate it. If no matching key was found by the end of the loop, the program indicated that the key was not discovered within the provided time frame.

The decryption process itself involved converting the Base64-encoded ciphertext into bytes and using the DES (Data Encryption Standard) algorithm to attempt decryption with the generated key. I set up a memory stream and a crypto stream to process the decryption, and once complete, I compared the decrypted output to the provided plaintext to determine if the correct key was used.

B. Project 2 - Hash

In my hash project, I implemented a solution to perform a birthday attack on MD5 hashes with a specific one-byte salt. The program starts by validating that the user provides a valid one-byte salt as a command-line argument. After conversion of the salt from a hexadecimal string to a byte, the program searches for two different strings that produce the same first five bytes of their MD5 hashes with the given salt.

```
if (args.Length != 1 || args[0].Length != 2)
      {
         Console.WriteLine("Please provide a valid 1-byte
salt as a command line argument.");
         return;
      }
      byte salt = Convert.ToByte(args[0], 16);
      var collision = FindCollision(salt);
      if (collision != null)
      {

Console.WriteLine($"{collision.Item1},{collision.Item2}");
      }
      else
      {
         Console.WriteLine("Collision not found.");
      }
```

The main logic is in the FindCollision method, which generates random 8-character alphanumeric strings and computes their MD5 hashes with the salt. It then checks if the first five bytes of the hash match any previously stored hash prefixes using a dictionary. If a match is found, it returns the two strings that collided.

The ComputeMD5HashWithSalt method calculates the MD5 hash of a string combined with the salt, converting the result to a hexadecimal string. The GenerateRandomString method creates random alphanumeric strings for testing. This approach demonstrates the birthday attack by identifying collisions in hash prefixes efficiently.

```
public static string ComputeMD5HashWithSalt(string input, byte salt)
   {  using (MD5 md5 = MD5.Create())
      {
         byte[] inputBytes = Encoding.UTF8.GetBytes(input);
         byte[] saltedInputBytes = new byte[inputBytes.Length + 1];
         Buffer.BlockCopy(inputBytes, 0, saltedInputBytes, 0, inputBytes.Length);
         saltedInputBytes[saltedInputBytes.Length - 1] = salt;
         byte[] hashBytes = md5.ComputeHash(saltedInputBytes);
         StringBuilder sb = new StringBuilder();
         foreach (byte b in hashBytes)
         {
            sb.Append(b.ToString("X2"));
         }
         return sb.ToString();
      }
   }
```

C. Project 3 - Diffie-Hellman and Encryption

In my Diffie-Hellman and encryption project, I developed a solution that combines the Diffie-Hellman key exchange with AES encryption and decryption. The program first parses command-line arguments to retrieve the initialization vector (IV), parameters for the Diffie-Hellman key exchange, and the ciphertext and plaintext for encryption and decryption.

The Diffie-Hellman process begins by calculating the base *g* and modulus *N* using the provided exponents and constants. With these, I compute the shared key by raising g^y mod *N* to the power of *x* modulo *N*. This shared key is then used to derive a 256-bit AES key by resizing it to 32 bytes.

```
BigInteger sharedKey = BigInteger.ModPow(gyModN, x, N);
byte[] aesKey = sharedKey.ToByteArray();
Array.Resize(ref aesKey, 32);
```

The solution performs decryption of the provided ciphertext using AES-256 in CBC mode, applying the derived key and IV. It then encrypts the given plaintext with the same key and IV. The results, including the decrypted plaintext and the newly encrypted ciphertext, are output in a formatted manner.

The methods DecryptAES and EncryptAES handle the AES operations, ensuring the proper use of CBC mode and PKCS7 padding. Additionally, the ConvertHexStringToByteArray method assists in converting hexadecimal input to byte arrays for cryptographic processing. This approach integrates cryptographic techniques to securely exchange and process data.

```
static byte[] ConvertHexStringToByteArray(string hex)
{
   hex = hex.Replace(" ", ""); // Remove any spaces
   byte[] bytes = new byte[hex.Length / 2];
   for (int i = 0; i < hex.Length; i += 2)
   {
      bytes[i / 2] = Convert.ToByte(hex.Substring(i, 2), 16);
   }
    return bytes; // Return the byte array
}
```

D. Project 4 - RSA

For my RSA project, I implemented a solution to handle RSA encryption and decryption by first calculating the necessary cryptographic components based on given parameters. The program begins by parsing command-line arguments to retrieve the exponents and constants for calculating the prime numbers *p* and *q*, as well as the ciphertext and plaintext.

```
BigInteger ciphertext = BigInteger.Parse(args[4]);
BigInteger plaintext = BigInteger.Parse(args[5]);
```

I calculated the prime numbers *p* and *q* using the formula $2^e - c$ where *e* and *c* are provided as arguments. Using these primes, I computed n as the product of *p* and *q*, and *φ(n)* as *(p−1)×(q−1)(p-1)* \times *(q-1)(p−1)×(q−1)*. With *e* set to 65537, a common choice for the public exponent, I then determined the private exponent d using the modular inverse of e modulo *φ(n)*, which was computed through the Extended Euclidean Algorithm.

The RSA encryption formula $m^e \bmod n$ was used to encrypt the plaintext, while the decryption formula $c^d \bmod n$ was used to decrypt the ciphertext. The results are output as a comma-separated pair of the decrypted plaintext and the encrypted ciphertext.

The ModInverse method calculates the modular inverse of e using the Extended Euclidean Algorithm to ensure the proper decryption of the ciphertext. The ExtendedGCD method finds the greatest common divisor (GCD) of two numbers and the coefficients needed for the modular inverse. This approach integrates fundamental RSA principles to secure and process data efficiently.

III. Description of Results

A. Project 1 - Steganography and Cryptanalysis

1) Part 1 - Steganography

I had a bitmap made up of the bytes:

```
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
```

I was able to conceal a message made up of the following bytes:

```
40 C4 D6 38 6F 52 C0 65 F9 EA 7A E5
```

Then generated an identical bitmap made up of the following 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 01 00 FF FF FC FF 01 00 FC FE
FE FD FF FC FD 00 01 02 FC FC FE 01 00 02 FC 00 00
FF FE FD FE 01 03 FC FD FE FC FD FD 02 01 03 FD FD
FC 02 01 01
```

This successfully contained my secret message.

2) Part 2 - Cryptanalysis

From running my program with the plaintext: “fHQ7ZEMO” and ciphertext: “VlRIuero1hGGtEgdNwvwYQ==” generated my expected seed output of “26562900”.

B. Project 2 - Hash

Running my program with the salt byte "C5" produced two strings, "KX9R3MO8" and "an9yFXRn," which resulted in identical salted hashes.

C. Project 3 - Diffie-Hellman and Encryption

For my encrypted message that was generated within the IDE, my program correctly decrypted it. Comparably, the program encrypted the given message:

```
d4cNqFsXRBIoU4n2bMZFfdB6rmF6rpeDUZreeeqLZg4F
H
```

To the expected bytes:

```
FC FE 3F FB DE C9 9A E1 06 9C C7 23 2C AC 6F 0D BB
9D 76 D7 05 4F B5 5C C5 87 F1 54 DC 93 25 21 96 A2 A3
DB F8 46 82 4F AF E3 01 0D 48 55 AD 36
```

D. Project 4 - RSA

My program correctly decrypted the following value:

```
83679104004393820193530588721549950832570045963383
5106325734292585523284902502882786124466883649
```

```
0803180195765156612316772416107737703528168364235
975266
```

To the following decrypted plaintext:

```
83679104004393820193530588721549950832570045963321
13912517
```

IV. Skills and Knowledge Acquired

Each project provided valuable insights into various aspects of cryptography. In Project 1, I gained hands-on experience with steganography, cryptanalysis, and random number generators. Project 2 focused on modern hash functions, where I learned how to detect collisions using a birthday attack. Project 3 involved using a 256-bit key for secure key exchange and encryption. Finally, Project 4 taught me how to implement the Extended Euclidean Algorithm and utilize the RSA algorithm for encryption and decryption.

References

[1] Gillis, Alexander. “Diffie-Helllman key exchange (exponential key exchange),” *TechTarget*, 2020. pp. 1-2. [Online]. Available: https://www.techtarget.com/searchsecurity/definition/Diffie-Hellman-key-exchange

[2] Grassi, Paul. “Digital Identity Guidelines,” *NIST Special Publication 800-63-3,* 2007. pp. 47. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-3.pdf

[3] Herring, Cardon. “Rivest, Shamir Adleman Encryption Algorithm - How RSA Encryption Works?” *Encryption Consulting LLC*, 2022. pp. 1-2. [Online]. Available: https://www.encryptionconsulting.com/education-center/what-is-rsa/

[4] Swenson, Christopher. “Modern Cryptanalysis: Techniques for Advanced Code Breaking,” *Wiley Publishing, Inc*, 2008. pp. xix-xxviii. [Online]. Available: https://books.google.com/books?hl=en&lr=&id=oLoaWgdmFJ8C&oi=fnd&pg=PP1&dq=what+is+cryptanalysis&ots=T8hGfVKggV&sig=ARbD7vQpLcbKgM6ZR_eRufhzPbk#v=onepage&q=what%20is%20cryptanalysis&f=false.