10-2017

Simple implementation of an ElGamal Digital Signature and a Brute Force attack on it

Valeriia Laryoshyna
Embry-Riddle Aeronautical University

10-13-2017

Simple implementation of an ElGamal Digital Signature and an attack on it

By Valeriia Laryoshyna (laryoshv@my.erau.edu)
Mentored by Dr. Paul Hriljac (hriljap@my.erau.edu)

Simple implementation of an ElGamal Digital Signature and a Brute Force attack on it

Directed Study

Presented to the Honors Program

Of Embry-Riddle Aeronautical University – Prescott Campus

In Partial Fulfillment of the Requirements

For the Honors Program

In the Degree of

Bachelors of Science in Cyber Intelligence and Security

By Valeriia Laryoshyna

October 13, 2017

Table of Contents

Page

Abstract

This study is an attempt to show a basic mathematical usage of the concepts behind digital signatures and to provide a simple approach and understanding to cracking basic digital signatures. The approach takes on simple C programming of the ElGamal digital signature to identify some limits that can be encountered and provide considerations for making more complex code. Additionally, there is a literature review of the ElGamal digital signature and the brute force attack.

The research component of this project provides a list of possible ways to crack the basic implementations and classifies the different approaches that could be taken to break the signature. One of those methods, brute force, is taken and applied to effectively break the math behind the digital signature model that was used. Analysis of the brute force attack is provided to show the trends with in the primitive roots. Countermeasures are then developed to show effectiveness and recommendations are included on how to develop the data more effectively.

Introduction

Digital signatures are verification methods used by providers and users who send traffic over the network. The initial goals of the project were to code up a digital signature so that its process could be observed and one of its vulnerabilities can be exploited. Going into this project, the process was to take the procedure used in the *Cryptography and Network Security: Principles and Practices* 7th *edition* book to recreate the signature. The C language was used because this program was not dealing with an object-oriented focus. However, the calculations could be considered objects so it could also be coded in an object-oriented language. The main object that could have been created was the message but it was not planned to be used. Then the code was planned to have functions for each of the following: primitive roots, prime numbers, and major variable calculations like the first part of the signature. Most of the goals for this whole project were also to focus on how to program a lot of the calculations such as the inverses and the greatest common denominators. Moreover, there were planned verification methods for the calculations with the process explained in the book. Some of the other goals included getting a better understanding of how the keys are created for digital signatures and how they are distributed. Eventually, the objective moved to exploiting the ElGamal signature and implementing one approach to that.

ElGamal Literature Review

The ElGamal algorithm is a public key cryptographic algorithm that uses asymmetric key encryption. Similar to RSA, ElGamal is faster than RSA and uses a key exchange process that is similr to Diffie Hellmen (Boomija & Kasmir Raja, 2016). Over the years, ElGamal has been implemented in a variety of different situations such as key encryption, clouds, and even different combinations of other algorithms in order to improve them. The ElGamal's usage has transformed, "originally used for digital signature, but later was modified to be used for encryption and decryption" (Satvika Iswari, 2016). Additionally, because the ElGamal is public-key focused, it main downside is speed (Satvika Iswari, 2016). However, in perspective, if that is the only downside to an algorithm the guarantees security, it is not necessarily the most problematic. The idea of modifying other algorithms with ElGamal has been around for years and applied in several ways. For instance, using ElGamal to help create dynamic public keys and reliable transmission with the Kerberos protocol (Cui et al., 2014). There is no doubt, that the ElGamal algorithm provides some benefits within cryptography.

C Programming Application

To program the ElGamal digital signature the code in Figure 1 of Appendix A was used. This followed the basic mathematical calculations presented in *Cryptography and Network Security: Principles and Practices* 7th *edition.* At the start, the function that would check for the primitive roots would only work for up to number 17. However, this part was later improved by taking the modulation of each increasing exponential factor. This code would create the signature pair for one of the values as shown below in Figure 1 and Figure 2 but would depend on a person's ability to check each number.

```
Welcome to the ElGamal digital signature this works with primes up to 17

The global elements of this signature are a prime q
and p, which is the primitive root of q

Enter a prime number q: 19
 This is in fact prime

Enter a primitive root of p: 10

 Starting PrimeCombo 171
10^1 mod 19 is 10
10^2 mod 19 is 5
10^3 mod 19 is 12
10^4 mod 19 is 6
10^5 mod 19 is 3
10^6 mod 19 is 11
10^7 mod 19 is 15
10^8 mod 19 is 17
10^9 mod 19 is 18
10^10 mod 19 is 9
10^11 mod 19 is 14
10^12 mod 19 is 7
10^13 mod 19 is 13
10^14 mod 19 is 16
10^15 mod 19 is 8
10^16 mod 19 is 4
10^17 mod 19 is 2
10^18 mod 19 is 1
Did any value repeat?(y for yes or n for no) n

This is in fact a primitive root


Next User A generates a private/public key pair. First your random integer is generated
Would you like to enter a number greater than 1 but less than 18, y for yes or n for no: y
Enter your random integer: 16
```

Figure 1: The ElGamal primitive check

Since the goal of the code was to make the primitive checking process automatic, it was later improved as shown by code in Figure 2 in Appendix A. This simple code only selected a

random value for the versus using an actual hash to keep it simple for the calculation. However, this could be taken and improved to include a well-known hash algorithm.

```
Next we computer Ya = p^Xa mod q
User A's Ya is 4
A's private key Xa is 16, A's public key is {q, p, Ya} {19,10,4}


Please enter your message:we try

Next we are gonna give the user Message M a hash value m.
Would you like to enter one? y/n: n

The hash value is 7

A then forms a digital signature as follows
  The a random integer K is generated 11

  Next we compute S1 = p^K mod q
  S1 is 14

  Next we compute K^-1 mod(q-1)
  K^-1 is 5 because 5*11 mod 18 = 1

  Next we compute S2 = K^-1(m-XaS1)mod(q-1)
  S2 is 13 because it is 5*(7 - 16*14)mod(18)

The signature consists of (S1, S2) which are 14 , 13
Would you like to verify? y/n: y


To check first we calculate V1 = p^m mod q

Then we calculate V2 = (Ya^S1 * S1^S2)mod q

Finally, we check whether they equal each other
The signature is valid V1 15 == V2 15
```

Figure 2: ElGamal digital signature

Although coding up the signature is a good starting point, an attack can be implemented with separate code on separate systems. Another recommendation about this attack is that it is good practice to create a setup with different systems and hosts to practice different ways to perform an attack on the digital signature. It is also important to get permission from software products used if the attack is to be done legally and within the terms of the user agreements that were agreed upon with those products, but this depends on the type of attack performed.

Classification of Different Attacks

Type of attack	What it is	How it would work
Brute Force	Develop a program that test all the values that could be the key	The message/file will have a specific range that the key can be in. This code would mathematically test every value possible
Exploit the rand() function	The code depends on a random number generator, however it could be said that the rand() function is not truly a "random" number generator because the same random values are generated each time the program runs	The attacker can predict the values that will be generated based on part of the code and therefore be able to use those values to decrypt the whole signature
Hash cracking	Breaking the hash used in the signature would give access to the values for the signature	Creating a program that references the different hash online calculators so that the hash that is transferred can be decrypted giving access to the values
Organized Crime	Unsuccessful attempts at breaking digital signature leads to unconventional tactics by the intruder	In this case either bribery, burglary, or tactics like battery are used to get the necessary information
Third Party Software	In order to obtain the signatures and their information, the agreement that the users signs allows the 3rd party to transfer the signature to different locations in the computer, network, or even outside of the network	This gives the 3rd party access to the signature allowing it to further dissect its components
Manipulation of bits within the environment	Either using physical or system means to changing the interpretation of data	This could allow false information to be seen as correct furthermore accepting a false signature
Classic man-in-the-middle	Upon the transfer of the data with the digital signature, a third device intercepts during the transfers and obtain the information	This could give the user of the third device to forge a similar document onto which a second signature is maintained so in the process the 3rd party is able to obtain both digital signatures of the communicating parties

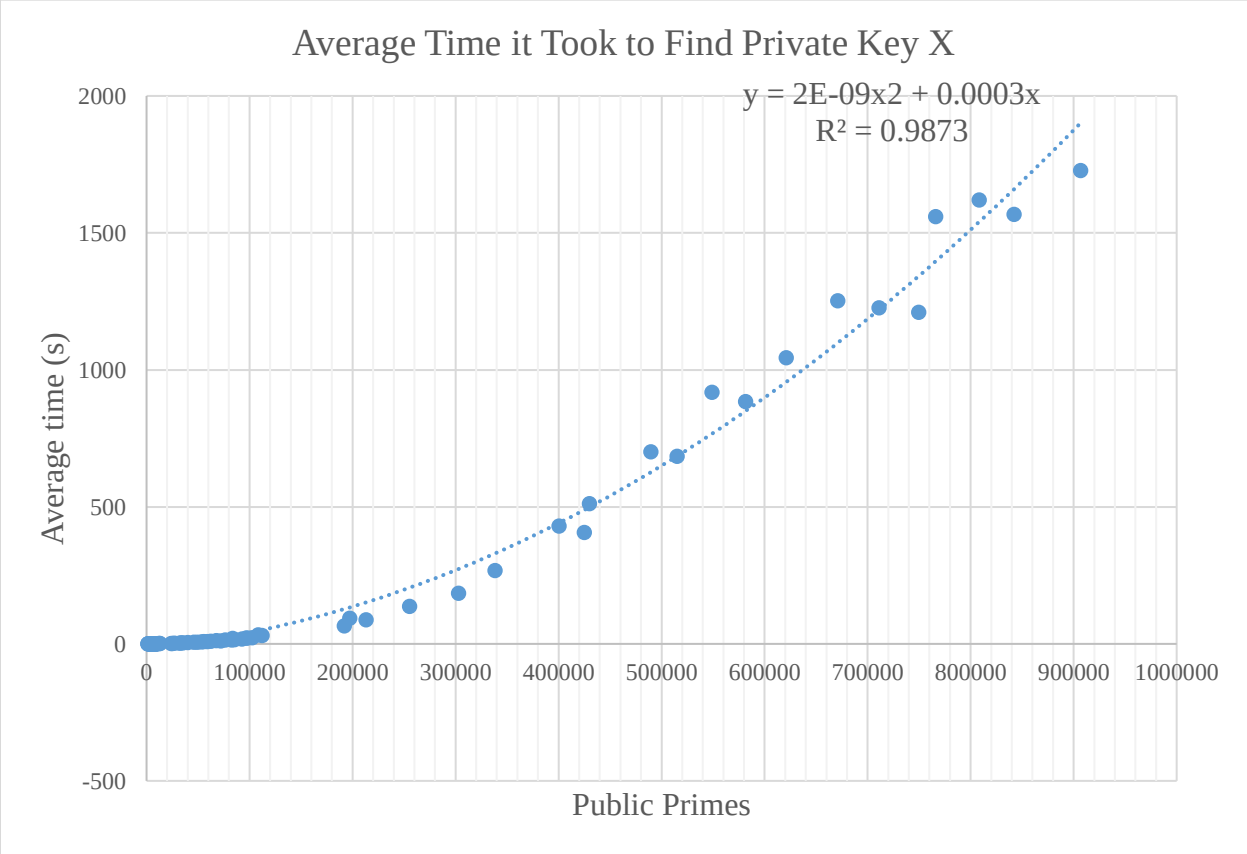
Simulated Brute Force Attack

In order to simulate a brute force attack, initial conditions need to be reviewed. The first is that the main prime number used was public. In order to get all the primitive roots of that initial prime, code was written to check every number and check whether it is primitive. The code was executed on a Red Hat Enterprise Linux Server release 6.9 for Embry-Riddle Aeronautical University for Prescott using SSH Secure Shell. That code used to check if a prime is a primitive root can be found in Figure 2 of the Appendix. The one drawback is that the higher primes get, the longer it took to get primitive roots. When it started to take over 20 min to get each primitive roots, it seemed more appropriate for the project to find the first primitive root and in order to save time the PrimitiveRoot function in Wolfram Mathematica 10 was used to quickly calculate the first primitive roots of each prime.

To properly simulate the brute force attack, it is important to understand which part needed to be calculated in the ElGamal scheme that was used. User A has two key and the attacker would have access to the public key which includes the main prime, the primitive root, and the integer Y which would be calculated using the private key X with, "$Y_A = \alpha^X_A \bmod q$," where q is the prime number and α is the primitive root (Stallings, 2017). To forge/break the signature, the attacker would need to find X and in order to do that the attacker would need to find it using the Y. To generate the Ys the RANDBETWEEN() function in excel was used to get 100 random Y integers between 2 and below each of 74 prime numbers selected. Then using code in Figure 3, the time was recorded for how long it took for each of the calculations to solve for X. It is important to solve for X because X is need to get the values for the second part of the signature which can't be calculated without it.

Results and Countermeasures Recommended

The data gathered on how long it took to get each X showed a clear exponential increase of time with the slope equation of y = 2E-09x^2 + 0.0003x where x is the prime used and y is the average time, this is shown by the graph below. So this says that increasing the size of the primitive root delays the time that it takes to solve for it using a basic brute force code. Therefore, it is recommended that the main public primitive number that is used is high.

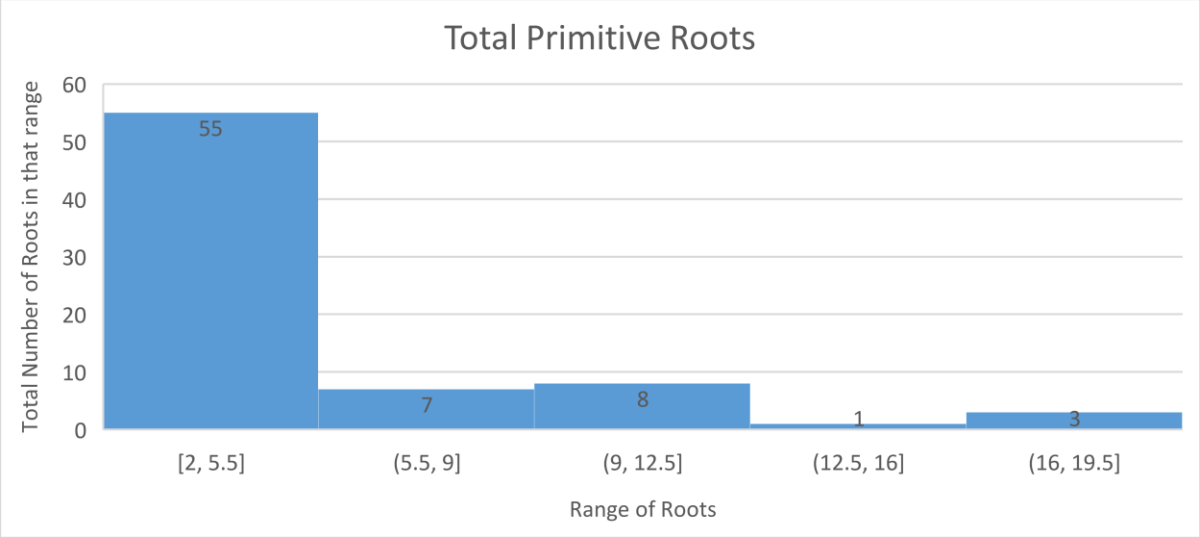


Graph 1: Average Time it Took to Find X for each of the randomly selected primes

Although increasing the prime number will increase the time it takes to solve for the X, the resulting times for this experiment do not reflect the exact time it might take for a different server, computing engine, or super computer to solve for the primes that were used here because

it might take them faster to calculate these numbers. For instance, Mathematica 10 might calculate for X in 10 seconds versus the code used here that took 330 seconds.

Additionally, there is a higher usage of initial primitive roots being integers under 6 as shown in Graph 2 below. This means that increasing the primitive root to a high number will also help delay the attack because it will be less predictable.



Graph 2: Total number of primitive roots used in each range

Although the primitive root was considered public in this case and there is code to solve for it, it is recommended that the primitive root that is used is not the first primitive root for that prime because not only are there more publicly available resources to help find the first primitive root for a prime but also this would delay the attack even more.

Some other ways to protect the data include encrypting the document with a secure algorithm which would make it difficult to read when it is being transferred across the network. Additionally, taking a hash of the file that requires the signature will help see if it has been tampered with.

Conclusion

The ElGamal algorithm is not the securest algorithm out there because there exist far more complex algorithms that may not have been commercialized. Based on the experiment that was done, for faster data collection, it may be wise, depending on the experiment, to invest in a server from well-known companies because then several process can be run without interruption of other processes. For instance, because the server that was used to collect the data was a public server that was available for the students at ERAU Prescott to use, the processes that were used were competing with the processes that were created by other students so it took much longer to collect the data. Additionally, the larger that the primitive number, the longer it also took to find the private key X. Moreover, because no algorithm is truly secure (especially because of the development of super computers), it is recommended that when the ElGamal algorithm is used for the digital signature the primitive number should be high and the primitive root chosen is not within the first five integers to delay a brute force attack.

Appendix A

1. elgamal.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int checkIfPrime (int newPrime);
int checkRoot (int newPrime, int newRoot);
int computeYa (int newRoot, int newXa, int newPrime);
int computeInverseK (int userK, int newPrime);
int computeS2 (int inverseK, int userHash, int userXa, int userS1, int newPrime);
int computeV2 (int Ya, int S1, int S2, int newPrime);

int main ()
{
 int p = 0, q = 0, randXa = 0, UserAYa = 0, userS1 = 0, userS2 = 0;
 int primeCheck = 0, primitiveCheck = 0, userInt = 0;
 int hashValue = -1, randK = 0, inverseK = 0;
 char userChoice;
 char userMessage[256];
 int gcd;
 int V1 = 0, V2 = 0;
 printf  ("Welcome to the ElGamal digital signature this works with primes up to 17\n\nThe global elements of this signature are a prime q\nand p, which is the primitive root of q\n");

 while (primeCheck == 0)
  {
   printf ("\nEnter a prime number q: ");
   scanf ("%d", &q);
   primeCheck = checkIfPrime (q);
   if (primeCheck == 1) printf (" This is in fact prime\n");
   else
    {
     printf (" Whoops, not prime\n");
     primeCheck = 0;
    }
  }
```

```
while (primitiveCheck == 0)
 {
   printf ("\nEnter a primitive root of p: ");
   scanf ("%d", &p);
   primitiveCheck = checkRoot (q, p);
   if (primitiveCheck == 1) printf ("\nThis is in fact a primitive root\n");
   else
    {
      printf ("\nWhoops, not a primitive root\n");
      primeCheck = 0;
    }
 }

printf ("\n\nNext User A generates a private/public key pair. First your random integer is generated");
printf ("\nWould you like to enter a number greater than 1 but less than %d, y for yes or n for no: ", q - 1);
scanf (" %c", &userChoice);

if (userChoice == 'n')
 {
   while ((randXa <= 1) || (randXa >= q - 1))
    {
      randXa = rand ();
    }
 }
else
 {
   while (randXa == 0)
    {
      printf ("Enter your random integer: ");
      scanf ("%d", &randXa);
      if ((randXa <= 1) || (randXa >= q - 1))
        randXa = 0;
    }
 }

printf ("\n\nNext we computer Ya = p^Xa mod q");
UserAYa = computeYa (p, randXa, q);
printf ("\nUser A's Ya is %d", UserAYa);
```

```
printf ("\nA's private key Xa is %d, A's public key is {q, p, Ya} {%d,%d,%d}\n",
  randXa, q, p, UserAYa);

while (getchar () != '\n');   //Clears the buffer
printf ("\n\nPlease enter your message:");
scanf ("%[^\n]", userMessage);

printf ("\nNext we are gonna give the user Message M a hash value m.");
printf ("\nWould you like to enter one? y/n: ");
scanf (" %c", &userChoice);

if (userChoice == 'n')
 {
   while ((hashValue < 0) || (hashValue > q - 1))
    {
      hashValue = rand ();
    }
 }
else
 {
   while (hashValue == -1)
    {
      printf ("Enter your hash value: ");
      scanf ("%d", &hashValue);
      if ((hashValue < 0) || (hashValue > q - 1)) hashValue = -1;
    }
 }
printf ("\nThe hash value is %d", hashValue);
printf ("\n\nA then forms a digital signature as follows");

while (randK < 1)
 {
   while ((randK < 1) || (randK > q - 1))
    {
      randK = rand ();
    }
   for (int i = 1; i <= randK && i <= (q - 1); i++)
    {
      if ((randK % i == 0) && ((q - 1) % i == 0))
```

```
            gcd = i;
          }
        if (gcd != 1)
          randK = 0;
      }
    printf ("\n  The a random integer K is generated %d\n", randK);

    printf ("\n  Next we compute S1 = p^K mod q\n");
    userS1 = computeYa (p, randK, q);
    printf ("  S1 is %d\n", userS1);

    printf ("\n  Next we compute K^-1 mod(q-1)");
    inverseK = computeInverseK (randK, q);
    printf ("\n  K^-1 is %d because %d*%d mod %d = 1", inverseK, inverseK,
          randK, q - 1);

    printf ("\n\n  Next we compute S2 = K^-1(m-XaS1)mod(q-1)");
    userS2 = computeS2 (inverseK, hashValue, randXa, userS1, q);
    printf ("\n  S2 is %d because it is %d*(%d - %d*%d)mod(%d) ", userS2,
          inverseK, hashValue, randXa, userS1, q - 1);

    printf ("\n\nThe signature consists of (S1, S2) which are %d , %d", userS1,
          userS2);

  //Verification
    printf ("\nWould you like to verify? y/n: ");
    scanf (" %c", &userChoice);
    if (userChoice == 'y')
      {
        printf ("\n\nTo check first we calculate V1 = p^m mod q");
        V1 = computeYa (p, hashValue, q);

        printf ("\n\nThen we calculate V2 = (Ya^S1 * S1^S2)mod q");
        V2 = computeV2 (UserAYa, userS1, userS2, q);

        printf ("\n\nFinally, we check whether they equal each other");
        if (V1 == V2)   printf ("\nThe signature is valid V1 %d == V2 %d\n\n", V1, V2);
        else  printf ("\nSignature is invalid V1 %d != V2 %d\n\n", V1, V2);
      }
```

```
 return 0;
}

int computeV2 (int Ya, int S1, int S2, int newPrime)
{
 int newV2 = 0;
 long long int exYa = 1, exS1 = 1;

 while (S2 != 0)
  {
   exS1 *= S1;
   --S2;
  }
 while (S1 != 0)
  {
   exYa *= Ya;
   --S1;
  }
 exYa = exYa % newPrime;
 exS1 = exS1 % newPrime;
 newV2 = exYa * exS1;
 newV2 = newV2 % newPrime;

 return newV2;
}

int computeS2 (int inverseK, int userHash, int userXa, int userS1, int newPrime)
{
 int newS2 = 0;
 newS2 = inverseK * (userHash - (userXa * userS1));
 newS2 = newS2 % (newPrime - 1);
 if (newS2 < 0)
  newS2 = newS2 + (newPrime - 1);
 return newS2;
}

int computeInverseK (int userK, int newPrime)
{
 int inverseK = 0;
```

```
  int testValue, posValue = 1;

  while (inverseK == 0)
   {
     testValue = (posValue * userK) % (newPrime - 1);

     if (testValue == 1)
      {
        inverseK = posValue;
      }
     else
      {
        posValue++;
      }
   }
  return inverseK;
}

int computeYa (int newRoot, int newXa, int newPrime)
{
  long long int newYa = 1;
  while (newXa != 0)
    {
      newYa *= newRoot;
      --newXa;
    }
  newYa = newYa % newPrime;

  return (int) newYa;
}


// Checks if number is prime
int checkIfPrime (int newPrime)
{
  if (newPrime <= 1)
    return 0;
  if (newPrime > 2)
    for (int factor = 2; factor < newPrime; factor++)
```

```
    {
     if (newPrime % factor == 0)
      return 0;
    }
  return 1;
}

int checkRoot (int newPrime, int newRoot)
{
  long long int primeCombo = 0, modCheck = 1;
  int power;
  char result;

  if (newRoot >= newPrime) return 0;
 for (int factor = 1; factor < newPrime; factor++) primeCombo = primeCombo + factor;
  printf ("\n Starting PrimeCombo %d", primeCombo);

  for (int exponent = 1; exponent < newPrime; exponent++)
   {power = exponent;
    while (power != 0){modCheck *= (long long int) newRoot; --power;}
    modCheck = modCheck % (long long int) newPrime;
    printf ("\n%d^%d mod %d is %lli", newRoot, exponent, newPrime, modCheck);
    modCheck = 1;}

  printf ("\nDid any value repeat?(y for yes or n for no) ");
  scanf (" %c", &result);

  if (result == 'y')
   return 0;
  else
   return 1;
}
```

2. Checks if Primitive Root

```
int checkRoot (int newPrime, int newRoot)
{
 int power;
 int totalNumbers[newPrime-1];

 if (newRoot >= newPrime)
  return 0;
```

```
 for(int loop = 1; loop < newPrime; loop++)
      {
      power = 1;
            for(int repeats = 1; repeats <= loop; repeats++)
            {
            power *= newRoot;
            power %= newPrime;
            }
      totalNumbers[loop-1] = power;
      }

// Checks for duplicates
for (int duplicate = 0; duplicate < newPrime - 1; duplicate++)
{
   for (int check = duplicate + 1; check < newPrime - 1; check++)
{
      if (totalNumbers[duplicate] == totalNumbers[check])
         return 0;   // Ends if duplicate found
   }
}
   return newRoot;  // Returns if it is a Primitive Root
}
```

3. Brute force for data collection

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>

int computeYa (int newRoot, int newXa, int prime);

int main ()
{
 int compute;
 time_t start, end;
 double dif;

 int hundredPrimes[100] =
int hundredRoots[100] ;

int foundXi[100];

 printf ("\nWelcome to part of Brute force operation where you find Xi and see how long it takes to find that.\n");

time(&start);
for(int value = 50; value < 100; value++)
{            printf("\n For %d :", value);
             time(&start);

             for(int check = 2; check < 100000000; check++)
             {
                        //if(check % 50000 == 0)
                        //printf("\nChecking %d", check);
                        compute = computeYa(hundredRoots[75], check, hundredPrimes[75]);

                        if(RandPublicKeys[value] == (compute))
                        {
```

```
                        time(&end);
                        dif = difftime(end,start);
                        foundXi[value] = check;
                        check = 100000000;
                printf("It took %.2lf seconds to find Xi = %d with primitive root %d, public key Yi = %d, and Prime p = %d", dif,
foundXi[value], hundredRoots[75], RandPublicKeys[value], hundredPrimes[75]);
                        }
            }
}
 return 0;
}

int
computeYa (int newRoot, int newXa, int prime)
{
 int newYa = 1;
 for(int power = 1; power <= newXa; power++)
 {
  newYa *= newRoot;
  newYa = newYa % prime;
 }

 return newYa;
}
```

References

Boomija, M. D. & Kasmir Raja, S. V. (2016). Secure data sharing through Additive Similarity based ElGamal like Encryption. *International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB16).* Retrieved from http://ieeexplore.ieee.org.ezproxy.libproxy.db.erau.edu/stamp/stamp.jsp?arnumber=7538370

Cui, Y., Du, Y., Ning, H., & Yang P. (2014). Improvement of Kerberos Protocol Based on Dynamic Password and "One-time Public Key". *2014 10th International Conference on Natural Computations.* Retrieved from http://ieeexplore.ieee.org.ezproxy.libproxy.db.erau.edu/document/6975980/

Satvika Iswari, N. M. (2016). Key Generation Algorithm Design Combination of RSA and ElGamal Algorithm. *2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE), Yogyakarta, Indonesia.* Retrieved from http://ieeexplore.ieee.org.ezproxy.libproxy.db.erau.edu/document/6975980/

Stallings, W. (2017). *Cryptography and Network Security: Principles and Practices* (7th ed.). Hoboken, NJ: Pearson Education, Inc.

Wolfram (2017). [Software]. Mathematica (Version 10). Available from https://www.wolfram.com/mathematica/