Spring 5-7-2021

Simulations of an Attack on RSA

Makayla Ferrell

Simulations of an Attack on RSA

Makayla Ferrell

submitted in partial fulfillment of the requirements for Honors in Mathematics at the University of Mary Washington

Fredericksburg, Virginia

April 2021

This thesis by **Makayla Ferrell** is accepted in its present form as satisfying the thesis requirement for Honors in Mathematics.

Date	Approved
_____	_____
	Randall Helmstutler, Ph.D. thesis advisor
_____	_____
	Debra Hydorn, Ph.D. committee member
_____	_____
	Larry Lehman, Ph.D. committee member

Contents

Abstract

We introduce a variation of Shanks' baby-step/giant-step (BSGS) to carry out a chosen plaintext attack on RSA. The original usage of BSGS is to find a solution to the discrete log problem. Although BSGS is guaranteed to find a solution, it is not guaranteed to find the matching RSA private key. Our variation of BSGS finds the matching private key d by using the order of the plaintext message to generate a set of solutions. We then use the length of this set to determine the most likely candidate for $\phi(n)$. We use a large data set to test the efficiency and accuracy of our variation.

1 Introduction

1.1 RSA cryptosystem

RSA is a public-key cryptosystem that is widely used for secure data transmission that partially relies on the Discrete Log Problem (DLP) [1]. In this system each user has a public key (n, e) for encryption, a private key (d) for decryption, and all calculations are carried out modulo n. When a user wants to send an encrypted message, they encrypt the message with the recipient's public key using $c = m^e \bmod n$, where c is the ciphertext and m is the message. Only the recipient, with their matching private key, can decrypt and view this message with $m = c^d \bmod n$. To better explain this, let's walk through an example.

First, we need to generate a set of keys for Alice. The first step to generate a key is to compute n, which is a product of two distinct prime numbers, p and q. In a round of encryption and decryption, all calculations are carried out mod $n = pq$. This thesis assumes knowledge of modular arithmetic and basic group theory. For this example, we will use $p = 5$ and $q = 11$, thus $n = 55$. The next step is to compute $\phi(n)$, where $\phi(n)$ is the Euler totient function. Since n factors as pq, $\phi(n)$ is calculated by $\phi(n) = (p-1)(q-1)$. In our case, $\phi(n) = (5-1)(11-1) = 40$. We must now choose any integer e that satisfies $\gcd(e, \phi(n)) = 1$. For simplicity, we will choose the first e that satisfies this, which is $e = 3$. Note that $\gcd(3, 40)$ is in fact 1. Alice can now publish her public key (55,3) for others to use when sending her an encrypted message.

However, for Alice to decrypt the messages sent to her, she will need to find her private key using $d = e^{-1} \bmod \phi(n)$. In our example, $d = 3^{-1} = 27 \bmod 40$. Alice will keep $d = 27$ a secret so that she is the only one who can decrypt messages encrypted with her public key. This is the standard formulation for RSA. The proof is included in Appendix B.

Now let's look at how to send encrypted messages in RSA. Let's pretend Bob wants to send Alice a message. Bob obtains Alice's public key (55,3). He can select any message $m \in \mathbb{Z}_{55}$. Bob chooses $m = 24$ to send to Alice. To encrypt his message, he uses $c = 24^3 = 19 \bmod 55$. Bob now sends the encrypted message 19 to Alice.

Now we can look at how to decrypt messages. When Alice receives the message she uses her private key and the following equation to decrypt: $m = c^d \bmod n$. Thus, $m = 19^{27} = 24 \bmod 55$. Notice that Alice was able to recover the message that Bob sent using only the private key, which only she possesses.

RSA relies on the Integer Factorization Problem and the Discrete Log Problem for its security. However, we will only be focusing on the Discrete Log Problem, which stops an attacker from using a chosen plaintext attack to discover the private key d. For example, suppose an attacker knows only the public key. In a chosen plaintext attack, the attacker will chose a message m and encrypt it with this public key, producing the ciphertext c. The attacker will then try to solve the decryption

formula $c^d = m \bmod n$ for d. However, this decryption formula is a Discrete Log Problem, meaning it holds the form $g^x = h \bmod n$, so the attacker cannot recover d. Thus, RSA is protected from a chosen plaintext attack.

1.2 Diffie-Hellman Key Exchange and ElGamal cryptosystem

Another protocol that relies on the Discrete Log Problem is the ElGamal cryptosystem that uses Diffie-Hellman Key Exchange to generate keys. It is important to note that the Diffie-Hellman Key Exchange (DHKE) protocol is not an encryption system. DHKE is mostly used to securely exchange cryptographic keys over a public channel [1]. Instead of users trying to send encrypted messages, Alice and Bob use DHKE to agree on a common key to use for the ElGamal encryption system.

Let's walk through how to use DHKE to generate a key. The first step is for Bob and Alice to agree on a prime p and a fixed element $g \in \mathbb{Z}_p$. The fixed element g must be primitive, meaning it is a generator of the group of units $\mathbb{Z}_p^*$ to ensure the security of DHKE. Both p and g are made public. We will use $p = 103$ and $g = 5$. Next, is the private portion of DHKE. Alice chooses an x such that $1 \leq x \leq p-1$, then computes $A = g^x \bmod p$. Let's say Alice chooses $x = 9$, then $A = 5^9 = 39 \bmod 103$. Next, we will do the same for Bob. Bob chooses a y such that $1 \leq y \leq p-1$, then computes $B = g^y \bmod p$. Bob will choose $y = 12$, making $B = 5^{12} = 34 \bmod 103$. Although they keep x and y secret, Bob and Alice exchange A and B. Alice will now compute her key using the following equation: $k_{Alice} = B^x = 12^9 = 72 \bmod 103$. The key for Alice is 72. Bob will compute his key using the following equation: $k_{Bob} = A^y = 39^{12} = 72 \bmod 103$. Note that Bob's key is also 72. They have now used DHKE to create a shared key.

Although DHKE cannot be used for encryption, it can be used as the first step in the ElGamal cryptosystem. In the ElGamal cryptosystem each user has their own individual key, just like in RSA. Alice's public key is (p, g, A). In our example, Alice would publish $(103, 5, 39)$. Her private decryption key is x. Recall that for Alice, $x = 9$.

Let's suppose Bob wants to send a message to Alice. Suppose he chooses a message $m \in \mathbb{Z}_p$, he chooses $m = 70$. To encrypt, Bob uses $c = \alpha m$, where α is the shared DHKE key we computed above ($\alpha = A^y = B^x \bmod p$). Thus, the ciphertext is $c = \alpha m = 72 \cdot 70 = 96$. Bob sends $(B, c) = (34, 96)$ to Alice. Alice will now decrypt by computing $m = \alpha^{-1} c = 72^{-1} \cdot 96 = 70$, where α is the DHKE key shared between Alice and Bob. Alice has now successfully decrypted Bob's message. Similarly to RSA, ElGamal is secure because the public information is not enough to solve $g^x = A$ for the private key x, which is a Discrete Log Problem.

1.3 Shanks' baby-step/giant-step

The security of ElGamal and RSA both rely on the Discrete Log Problem (DLP). The Discrete Log Problem can be generically stated with a modulus n and all calculations occurring in $\mathbb{Z}_n$, where we have two fixed elements $g, h \in \mathbb{Z}_n$, and an unknown integer x satisfying

$$g^x = h \bmod n.$$

The goal is to solve this expression for x, the private key, much quicker than brute force, which is trying every possible solution. Although the DLP may look unfamiliar in generic terms, we have already seen it twice: in RSA, where $c^d = m \bmod n$ and in DHKE/ElGamal where $g^x = A \bmod p$. The Discrete Log Problem is considered very difficult to solve since there is not a known algorithm considerably faster than brute force that will always produce a solution. However, when certain

hypotheses are met, Shanks' baby-step/giant-step (BSGS) algorithm can find a solution to the DLP faster than trying every solution [1].

To better understand Shanks' baby-step/giant-step, we will first walk through the algorithm and then show an example with a small modulus. Suppose we wish to solve for x in $g^x = h \bmod n$. The first step is to set $N = \text{ord}(g)$. As a reminder, the order of an element a is the smallest positive integer m with $a^m = e$, where G is a finite group and $a \in G$. The next step in BSGS is to set $m = 1 + \lfloor \sqrt{N} \rfloor$. Now we create List B and List G. List B consists of terms g^i, while List G consists of terms hg^{-im}. In these lists, $i = 0, 1, 2, ..., m$.

i	0	1	2	...	m
List B	1	g	g^2	...	g^m
List G	h	hg^{-m}	hg^{-2m}	...	hg^{-m^2}

Once we have created the lists, we search for a collision, which is just a match between both lists. Due to the way that N and m are computed, we are guaranteed to find a collision between the lists. In this case, a collision is where $g^i = hg^{-jm}$. After discovering the collision, the solution can be found using $x = i + jm$, where i is the index of the collision in List B and j is the index of the collision in List G. We have now found x, the private key, and solved this particular DLP. Let's show that x is correct. If we substitute $x = i + jm$ into g^x, we get $g^x = g^{i+jm} = g^i g^{jm} = hg^{-jm} g^{jm} = h$. Thus, the solution is correct.

To better illustrate this, let's use BSGS on our ElGamal example from earlier. Recall that for Alice, $g = 5$, $A = 39$, and $p = 103$. The DLP for this problem is

$$5^x = 39 \bmod 103.$$

To match the notation in this section, we will use generic notation where $g = 5$, $h = 39$, and $n = 103$. To solve for Alice's private key x, we set $N = \text{ord}(g) = 102$. Next, we set $m = 1 + \lfloor \sqrt{N} \rfloor = 1 + \lfloor \sqrt{102} \rfloor = 11$. Now we can generate two lists. List B starts with 1, then g, and each subsequent term is multiplied by $g \bmod n$. For our example, list B starts with 1, then $g = 5$, and the next term is $g^2 = 25$. The rest of the list will be generated by multiplying by g. List G starts with h, then hg^{-m}, and each subsequent term is multiplied by $g^{-m} \bmod n$. The first three terms of list G are $h = 39$, $hg^{-m} = 33$, and $hg^{-2m} = 20$. The lists are continued below.

i	0	1	2	3	4	5	6	7	8	9
List B	1	5	25	22	7	35	72	51	49	39
List G	39	33	20	9	71	68	10	56	87	

Notice that there is a collision between the lists so we do not have to finish both lists to $i = 11$. The collision (39) is at $i = 9$ for list B and $j = 0$ for list G. We can now compute $x = i + jm = 9 + 0(11) = 9$. This is Alice's private key that was supposed to be secure and secret. We can now decrypt all messages sent to Alice. We can even take it a step further and compute the secret key Bob and Alice made. Bob's public $B = 12$ and we now know $x = 9$, so the key is $B^x = 12^9 = 72$. To show this solution works, let's go back to the original DLP. As a reminder, the original DLP was $5^x = 39 \bmod 103$. We can plug in $x = 72$ and get $5^{72} = 39 \bmod 103$. Thus, we have used BSGS to find a solution to this DLP.

Now that we understand Shanks' baby-step/giant-step (BSGS) algorithm, there is a clear contradiction. RSA is protected from a chosen plaintext attack by a DLP, and similarly the security of DHKE and ElGamal relies on a DLP. Hoewver, BSGS appears to solve discrete log problems. Can BSGS be used to defeat these systems?

2 BSGS Analysis

2.1 BSGS Complexity

Shanks' baby-step/giant-step (BSGS) does find a solution to the DLP, but how efficient is this algorithm in finding a solution? We define efficiency of an algorithm based on time, memory, and its relation to brute force. In the BSGS algorithm, the first step is to compute $N = \text{ord}(g)$, where g is the base term in our DLP. In terms of efficiency, BSGS only requires $2\sqrt{N}$ tries to guarantee a solution is found, as opposed to trying every possible solution. However, BSGS is not the most efficient in terms of time or memory. Recall that in BSGS we make two lists that could be of length m, where $m = 1 + \lfloor\sqrt{N}\rfloor$. The memory of storing these two lists and doing the calculations to produce these lists with a 2048-bit moduli is exhaustive. Additionally, to generate the lists used in BSGS, we multiply each subsequent term by g in List B and by g^{-m} in List G. All calculations are done in modulus n which is time exhaustive with such large moduli. It is due to this complexity that BSGS is not seen as an efficient solution to solving discrete log problems.

2.2 Prime versus Semi-prime Moduli

In order to highlight a key aspect of BSGS, we will follow two examples. The first example uses $n = 31$, $g = 3$, $h = 5$. The resulting DLP is $3^x = 5 \bmod 31$. The matching decryption key and expected solution for this problem is $x = 20$. Let's use BSGS to get that solution. First, we find N and m, and in doing so we get

$$N = \text{ord}(3) = 30$$
$$m = 1 + \lfloor\sqrt{31}\rfloor = 6$$

Next, we produce the following table to find a collision.

i	0	1	2	3	4	5	6
List B	1	3	9	27	19	26	16
List G	5	10	20	9	18	5	10

The collision occurs at $i = 2, j = 3$. Thus, $x = 2 + 3(6) = 20$. Notice we were able to use BSGS to get the intended solution to the DLP.

Next, let's use the example from the RSA section, where $n = 55$. We will use $h = 53$ and $g = 47$ The resulting DLP is $47^x = 53 \bmod 55$. We will use BSGS to try to find the private key x. From the earlier section, we know $x = 27$. To start, we find N and m, and in doing so we get

$$N = \text{ord}(47) = 20$$
$$m = 1 + \lfloor\sqrt{20}\rfloor = 5$$

Next we produce the following table to find a collision.

i	0	1	2	3	4	5
List B	1	47	9	38	26	12
List G	53	9	42	31	53	9

The collision occurs at $i = 2, j = 1$. Thus, $x = 2 + 1(5) = 7$. Notice our solution x is not 27. However, this x does does serve as a valid decryption exponent for all messages. Let's plug $x = 7$ into the DLP. That produces $47^7 \bmod 55$ which equals 53. Thus, $x = 7$ is a solution. This is an important finding: BSGS does not find the solution but a solution. The solution BSGS finds will solve the DLP but will not always be the matching private key.

The key difference in the two examples are the moduli, n. In the first example, $n = 31$ is a prime number. In the second example, $n = 55$ is a semi-prime, meaning it is a product of two prime numbers. When the modulus is a semi-prime, BSGS will find a solution that decrypts but is not always the private key. This is because, generally, there are multiple solutions to an RSA DLP. In the next section, we will explore this finding.

2.3 Multiple Solutions to the DLP

In the last section, we discovered that there exists more than one solution to the DLP and that BSGS finds only one such solution. Let's look at our RSA example again. In this example, $n = 55$ and the DLP was $47^x = 53 \bmod 55$. In the previous section we ran BSGS on this problem and stopped the lists after the first collision. If we continue to generate the lists for BSGS until the next collision, we can see that BSGS will get us to another solution to this DLP.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
List B	1	47	9	38	26	12	14	53	16	37	34	3	31
List G	53	9	42	31	53	9	42	31	53	9	42	31	

Notice that the second collision is at $i = 7$, $j = 4$. For this collision, $x = 7 + 4(5) = 27$. This is also a solution to the DLP. Notice that $x = 27$ decrypts, $47^{27} = 53 \bmod 55$. The logical next question is how many solutions exist? Recall that the first solution we found with BSGS was 7, and this solution was 27. The difference is 20, which is the order of our plaintext message. This is actually the case for all RSA discrete log problems. We can produce more solutions to RSA discrete log problems by adding the order of the plaintext message to a previous solution.

The proof for this is quite simple. We will denote M_n as the maximum possible order across all invertible elements in n. We know that the ciphertext c raised to the private key d equals the plaintext message m, in other words $c^d = m$. We can prove that c^{d+M_n} also equals m. This is because $c^{d+M_n} = c^d c^{M_n}$. It is clear that $\text{ord}(m) = \text{ord}(c)$, and therefore $c^{M_n} = 1$. Therefore, $c^{d+M_n} = c^d \cdot 1 = c^d = m$. We have now proven that we can find more solutions to these discrete log probelms by adding the order of the plaintext message to a previous solution. Thus, we know that there are multiple keys that can decrypt a RSA discrete log problem.

3 Alternative BSGS Usage

3.1 Example

In this section we will walk through an RSA example to illustrate an alternative way to use BSGS to get the most likely $\phi(n)$ and the matching private key d. We will start with only publicly available information. In RSA, the only thing that is made public is the public key (n, e). In this example we will use $n = 1189, e = 3$. As discussed in the earlier section, the best way to use BSGS on an RSA DLP is a chosen plaintext attack. In a chosen plaintext attack, we select and encrypt a plaintext message so that we can craft our own discrete log problem. We will choose plaintext message m to be 35. To encrypt m, we use $m^e = c \bmod n$. In our example $c = 35^3 = 71 \bmod 1189$. Now that we have chosen and encrypted our plaintext, we can set up the DLP. As a reminder the DLP for RSA looks like $c^d = m \bmod n$. Thus the DLP for our example is

$$71^d = 35 \bmod 1189.$$

We can now use BSGS to find a solution to this problem. For simplicity, we will use generic DLP notation when doing BSGS: $g = 71$, $h = 35$, $n = 1189$. The first step in BSGS is to set

$N = \text{ord}(g) = \text{ord}(71) = 280$. Next, we set $m = 1 + \lfloor\sqrt{N}\rfloor = 1 + \lfloor\sqrt{280}\rfloor = 18$. Now we can create List B and List G.

i	0	1	2	3	4	5	6	7	8	9	10
List B	1	71 (g)	285 (g^2)	22	373	325	484	1072	16	1136	993
List G	35 (h)	767 (hg^{-m})	468 (hg^{-2m})	608	109	792	846	93	883	700	1072

Notice the collision at $i = 7$ and $j = 10$. Using these values for i and j, we can calculate the solution $x = i + jm = 7 + 10(18) = 187$. This value should decrypt our problem, and it does. By plugging in $x = 7$ into the decryption formula we get $71^{187} = 35 \bmod 1189$.

To test our solution even more, let's use this key to encrypt and decrypt another message. We will use $m = 68$. This means that the ciphertext $c = 68^3 = 536 \bmod 1189$. In order for our newly discovered solution to be a true solution, it must decrypt 536 and give us 68. This means 536^{187} mod 1189 must equal 65, and it does. That means that this solution can act as a private key for RSA and decrypt messages in $\mathbb{Z}_n$. However, this is not the matching private key for $(1189, 3)$. Recall that to create the matching private key we used $d = e^{-1} \bmod \phi(n)$. Sadly, $d = 187$ does not make this expression true. However, there is a way to use BSGS to continue to find $\phi(n)$ and then the matching private key.

First we must run BSGS again, but this time we will let BSGS run until the second collision is found. Here are portions from list B and list G if they were extended. In order to save space, i starts at 23.

i	23	24	25	26	27	28	29	30	31	32	33	34	35
List B	962	529	700	951	937	1132	709	401	1124	141	499	948	724
List G	526	724	341	67	585	760	1028	990	463	1008	212	875	

Notice the second collision happens at $i = 35$ and $j = 24$. Thus, we can calculate our second solution: $x = 35 + 24(18) = 467$. As with our last solution, this solution can decrypt messages but is not the matching private key. However, we can use our two solutions to obtain a most likely candidate for $\phi(n)$.

First, we calculate the difference between the two solutions: $s_1 - s_0 = 467 - 187 = 280$. This number is the order of our plain text message, and can be used to generate a set of solutions by adding 280 (M_n) to a previously found solution. For example, the next solution we can create is 747, which is $467 + 280$. We can continue this and create more solutions, $1027(747 + 280)$ and $1307(1027 + 280)$ that also decrypt our Discrete Log Problem. However, we need to limit this set of solutions so we will use this set of solutions mod $\phi(n)$. If we mod $\phi(n)$ the length of the set of solutions is $\frac{\phi(n)}{\text{ord}(m)}$. This will be explained in a later section. Recall that $\phi(n) = (p-1)(q-1)$, but we do not know p or q. This means we cannot calculate $\phi(n)$ or length at this time. However, we can use an estimate of $\phi(n)$ to discover the most likely $\phi(n)$.

To do this we will estimate $\phi(n)$ using $\phi(n)_{est} = n - 2\lfloor\sqrt{n}\rfloor$ [3]. This estimate is larger than the actual $\phi(n)$ but close enough to allow us to discover the most likely $\phi(n)$. To differentiate between $\phi(n)$ and our estimate, we will denote the estimated $\phi(n)$ as $\phi(n)_{est}$. For this example, $\phi(n)_{est} = n - 2\lfloor\sqrt{n}\rfloor = 1189 - 2\lfloor\sqrt{1189}\rfloor = 1121$. Now we can calculate the approximate length of the solution set using $\phi(n)_{est}$ for $\phi(n)$ in our length expression. This means $L = \frac{\phi(n)_{est}}{\text{ord}(m)} = \frac{1121}{280} = 4.0036$. We know that the length can't be a decimal, and we also know that our estimated $\phi(n)$ was too large so to get the actual length we need to floor this number, $\lfloor 4.0036\rfloor = 4$. We now know the length of our solution set mod $\phi(n)$ is 4. We can calculate the most likely $\phi(n)$ by rearranging our length formula to produce $\phi(n) = \text{ord}(m) \cdot L$. Plugging in our example variables

gives $\phi(n) = \text{ord}(m) \cdot L = 280 \cdot 4 = 1120$. We have done it. The most likely candidate for $\phi(n)$ is 1120. Additionally, the correct $\phi(n)$ is 1120.

Now to find our matching private key, we can use $d = e^{-1} \bmod \phi(n)$. Plugging in our example variable produces $3^{-1} = 747 \bmod 1120$. This is the correct private key. We have now been able to use BSGS to find a solution to this DLP, the most likely $\phi(n)$, and the matching RSA private key while only having access to the RSA public key. This approach has potential, but we did leave out some key details explaining how we are able to do these steps. In the next sections, we will generalize this algorithm and explain the mathematics that allows us to use this algorithm to solve an RSA Discrete Log Problem.

3.2 Algorithm

Now that we have seen an example of our alternative BSGS usage, we can walk through this algorithm in general terms.

We will start with having only an RSA public key (n, e).

1. Choose a message h and encrypt with $h^e = g \bmod n$ to get ciphertext g.
2. Set up the DLP for this problem using $g^x = h \bmod n$.
3. Run Shanks' baby-step/giant-step on this DLP until two collisions are found. Use each collision to compute a solution. We will denote the first solution as s_1 and the second solution as s_2.
4. Compute the maximal order of the plaintext message using $M_n = s_2 - s_1$.
5. Estimate $\phi(n)$ using $\phi(n)_{est} = n - 2\lfloor\sqrt{n}\rfloor$
6. Calculate the approximate length of the solution set mod $\phi(n)$ using $L = \frac{\phi(n)_{est}}{M_n}$
7. Find the most likely $\phi(n)$ using $\phi(n)_{rev} = \lfloor L \rfloor \cdot M_n$.
8. Find the matching private key d using $d = e^{-1} \bmod \phi(n)$

Now that we have enumerated the steps to the algorithm, let's discuss each step in more detail. The first two steps allow us to carry out a chosen plaintext attack. As a reminder, a chosen plaintext attack is where we select a message and encrypt it ourselves to create a DLP. To do the next step, we must first define two conjectures.

Conjecture 3.1. BSGS will always produce a solution that is a valid decryptor for all messages $m \in \mathbb{Z}_n$.

Conjecture 3.2. Running BSGS twice will always produce two sequential solutions, where the difference is M_n.

Using these conjectures, the third step allows us to strategically use BSGS to find the maximal order of our plaintext message (M_n). Although the first step of BSGS is to find the order of the plaintext message, $N = \text{ord}(g)$, this order is of an element and cannot be used for generating a set of solutions. However, the two solutions produced by BSGS are separated by the true maximal order of the plaintext message, which is why we compute the difference of the solutions as step four. Step five uses a published upper bound for $\phi(n)$ to give us an adequate estimated $\phi(n)$.

Step six requires a lengthier explanation. It is a known fact that M_n divides $\phi(n)$. That means there exists some L where $L = \frac{\phi(n)}{M_n}$. However, this doesn't explain why this is the length of our solution set. Suppose we have solution s_1. To generate more solutions we can use $s = s_1 + rM_n$, where $r \in \mathbb{Z}$. Eventually we will reach $r = L$, where $s = s_1 + LM_n$ or in other words, $s = s_1 + \phi(n)$. However, we are generating solutions mod $\phi(n)$, meaning $s = s_1 + \phi(n) = s_1$. We have now circled back to the beginning of our set. Thus, there are L solutions in our set mod $\phi(n)$.

In step six of our algorithm, we use an estimated $\phi(n)$, denoted $\phi(n)_{est}$, to find an approximate length. As seen in our example, $\phi(n)_{est}$ is larger than $\phi(n)$ and L is a decimal number. Thus we need to floor L before using it to calculate the most likely $\phi(n)$, denoted $\phi(n)_{rev}$. In step seven, we rearrange the length equation to solve for $\phi(n)$ and use the floor of L to calculate $\phi(n)_{rev}$. Finally, in step eight, we use the RSA algorithm for creating private keys to find our matching private key d.

In this algorithm, there are a few spots where we have skipped proofs. For example, we did not prove that the solutions produced by BSGS always decyrpt, or that $\phi_{est} - \phi(n) < M_n$ which is required to prove we can do the last step of the algorithm to find the most likely $\phi(n)$. We do not have proofs for these statements. However we have run 250 trials of varying length moduli and all trials support these conjectures. The summary and discussion of these trials will be discussed in the next subsection.

3.3 Analysis

In this section, we will discuss the findings of the 250 test cases that were run against our new algorithm. All test cases can be found in Appendix A. To produce the test cases, random primes p and q were selected. The modulus n was then calculated and an e that satisfies $\gcd(e, \phi(n)) = 1$ was selected. Every test case used $m = 53$ or $m = 54$. In an ideal chosen plaintext attack, a message with a easily computed order would have been selected for each modulus. However, since we are running a large data set, a generic message was used. Finally our new algorithm was run against the public key (n, e). A small subset of these cases are included below for discussion.

n	e	$\phi(n)_{rev}$	$\phi(n)$	$\phi(n)_{rev} - \phi(n)$	Runtime (s)
55	3	40	40	0	1.8358e-05
6767	7	6600	6600	0	0.00012708
196481	7	193200	193200	0	0.00081587
9069271	3	9061000	9061000	0	0.009043
141701759	7	141676704	141676704	0	3.155
450933727	3	450890776	450890776	0	15.015
571110181	7	571061664	571061664	0	6.6785
628834919	7	628784520	628784520	0	0.63923
802889363	7	802832688	802832688	0	55.013

The first column n is the modulus used for that test case and the second column is the matching public exponent e. The moduli used for the test cases were created by taking the product of randomly selected prime numbers. The moduli used in these cases ranged from two digits to ten digits. The third column shows the most likely $\phi(n)$ found by our algorithm, denoted $\phi(n)_{rev}$ for revised estimate. The fourth column shows the true $\phi(n)$. To further enforce the accuracy of our algorithm, the fifth column shows the difference between our candidate for $\phi(n)$ and the true $\phi(n)$. This value was zero for all test cases, meaning our algorithm found the correct $\phi(n)$ every time. The last column shows runtime of each test case in seconds. The average runtime for all 250 test

cases was 4.2457 seconds. This is faster than the average runtime of trying to factor n into $n = pq$ which was 8.3212 seconds for all 250 test cases.

Let's discuss the efficiency of this new algorithm. Since the algorithm relies on running BSGS twice, it is twice as exhaustive as BSGS on memory. However, even with this extensive memory usage, this algorithm is faster than trying to factor n to find pq on average. Although the accuracy of the algorithm has not been proven, the test cases support that the algorithm will always be able to produce a solution and find the matching RSA private key.

A Test Cases

n	e	$\phi(n)_{rev}$	$\phi(n)$	$\phi(n)_{rev} - \phi(n)$	Runtime (s)
55	3	40	40	0	1.8358e-05
69	3	44	44	0	1.3113e-05
85	3	64	64	0	1.4067e-05
323	7	288	288	0	2.5988e-05
973	7	828	828	0	7.391e-05
1241	7	1152	1152	0	3.4094e-05
1387	7	1296	1296	0	3.1948e-05
3749	7	3564	3564	0	6.4135e-05
3827	7	3696	3696	0	0.00031114
4009	7	3780	3780	0	0.00017095
5767	7	5616	5616	0	0.00013304
6767	7	6600	6600	0	0.00012708
7313	7	7140	7140	0	0.00042391
8479	7	8280	8280	0	0.00016904
8977	3	8740	8740	0	0.00049186
9143	7	8880	8880	0	0.00060272
12193	3	11968	11968	0	0.00037217
15853	3	15580	15580	0	0.001384
23533	3	23200	23200	0	0.0022011
24503	7	24168	24168	0	0.00026584
37627	3	37240	37240	0	0.002625
196481	7	193200	193200	0	0.00081587
654289	3	646324	646324	0	0.035557
784639	7	782856	782856	0	0.010215
859043	7	854400	854400	0	0.054813
2646823	3	2640088	2640088	0	0.077633
2722121	7	2708244	2708244	0	0.15295
3526577	7	3522420	3522420	0	0.068986
4255127	7	4245360	4245360	0	0.019329
4325003	7	4318440	4318440	0	0.029
4330891	7	4321512	4321512	0	0.021102
4640033	7	4613040	4613040	0	0.030334
4754831	7	4734192	4734192	0	0.25225
5823613	7	5817600	5817600	0	0.062107
6087559	3	6063616	6063616	0	0.32129
6847933	3	6822208	6822208	0	0.03035
6856961	7	6841812	6841812	0	0.37865
8300791	3	8294584	8294584	0	0.42453
9069271	3	9061000	9061000	0	0.009043
9873781	3	9852172	9852172	0	0.27026
9890449	7	9884160	9884160	0	0.0032811
10608361	3	10599424	10599424	0	0.14711
11637229	7	11625984	11625984	0	0.00348
12218021	7	12208812	12208812	0	0.31516

n	e	$\phi(n)_{rev}$	$\phi(n)$	$\phi(n)_{rev} - \phi(n)$	Runtime (s)
12904427	7	12896880	12896880	0	0.021717
13726877	7	13714716	13714716	0	0.36001
15048613	3	15040300	15040300	0	0.77451
15897289	3	15879556	15879556	0	0.08427
16636709	7	16610880	16610880	0	0.01205
16731059	7	16720704	16720704	0	0.016105
18410999	7	18392400	18392400	0	0.035305
23266769	7	23248740	23248740	0	0.42889
23307829	3	23279776	23279776	0	0.32106
23632717	7	23617980	23617980	0	0.23157
24172723	3	24162880	24162880	0	0.19786
24465611	7	24447672	24447672	0	0.007551
25150187	7	25140120	25140120	0	0.28527
25258231	7	25236000	25236000	0	0.039919
25380617	7	25368960	25368960	0	0.079527
26339431	7	26329032	26329032	0	0.49804
26597357	7	26585280	26585280	0	0.15976
27346127	7	27334776	27334776	0	0.76538
27695527	7	27669096	27669096	0	0.51714
28440011	7	28422912	28422912	0	1.5919
28972051	3	28957552	28957552	0	1.6474
29684131	3	29668912	29668912	0	0.24891
30769927	7	30752496	30752496	0	0.012291
30801269	7	30778860	30778860	0	0.89935
31534259	7	31518000	31518000	0	0.31422
33918499	7	33895440	33895440	0	0.09387
34068869	7	34050540	34050540	0	2.0576
34826443	7	34796520	34796520	0	0.032951
34837643	7	34817520	34817520	0	0.70316
36912149	7	36899136	36899136	0	0.10205
36937183	3	36918280	36918280	0	1.1535
38202767	7	38182560	38182560	0	0.59846
39376451	7	39362232	39362232	0	2.3942
39556787	7	39544176	39544176	0	0.15396
43074833	7	43046640	43046640	0	0.038461
44890717	7	44876700	44876700	0	0.98519
48107471	7	48093600	48093600	0	0.61834
49600409	7	49586304	49586304	0	0.7867
52097321	7	52078992	52078992	0	0.0081179
53319317	7	53299980	53299980	0	3.5668
56461079	7	56445984	56445984	0	0.59945
58222667	7	58207296	58207296	0	3.6917
59724109	7	59696316	59696316	0	0.63324
60494047	3	60478360	60478360	0	0.97379
60614227	7	60598656	60598656	0	0.1841
61000603	7	60978960	60978960	0	0.33276
61139791	3	61112632	61112632	0	3.9956

n	e	$\phi(n)_{rev}$	$\phi(n)$	$\phi(n)_{rev} - \phi(n)$	Runtime (s)
63075163	7	63043200	63043200	0	0.23214
68244251	7	68223504	68223504	0	2.2285
73835327	7	73817760	73817760	0	4.8599
75744733	3	75723040	75723040	0	0.26538
76619957	7	76599936	76599936	0	0.12937
79998983	7	79978200	79978200	0	2.6393
80079919	3	80056960	80056960	0	1.3746
82467703	3	82448608	82448608	0	2.6665
82816081	7	82792512	82792512	0	0.15839
84188497	3	84169396	84169396	0	2.7315
84409147	7	84384576	84384576	0	1.8283
85966163	7	85944408	85944408	0	1.8768
90100961	7	90078912	90078912	0	0.25038
91582823	7	91563648	91563648	0	0.50146
97696771	7	97674552	97674552	0	0.72244
101133407	7	101111136	101111136	0	1.1031
103149379	7	103117176	103117176	0	0.74879
103704893	7	103683900	103683900	0	2.2695
107441651	7	107415000	107415000	0	0.15898
109113247	3	109085080	109085080	0	7.2908
110483773	7	110459808	110459808	0	1.2249
111775507	3	111754240	111754240	0	0.29079
111938129	7	111913584	111913584	0	1.841
113825797	7	113804460	113804460	0	0.13069
121973923	7	121944168	121944168	0	2.7146
124184393	7	124154640	124154640	0	0.16151
126739741	3	126717184	126717184	0	2.0874
129785783	7	129761160	129761160	0	2.8618
136511539	3	136487824	136487824	0	4.5273
138314117	7	138290316	138290316	0	4.7832
138447733	3	138422368	138422368	0	4.6893
139025093	7	139000620	139000620	0	9.2772
139738139	7	139707024	139707024	0	3.0905
141014233	7	140987700	140987700	0	3.094
141701759	7	141676704	141676704	0	3.155
144827611	3	144802672	144802672	0	2.3838
146255999	7	146231760	146231760	0	9.6838
153115153	7	153085680	153085680	0	1.7067
154995811	3	154970800	154970800	0	2.1016
156653089	7	156619440	156619440	0	0.027266
158888269	3	158862460	158862460	0	5.3302
160367173	3	160341280	160341280	0	1.3634
165550361	7	165524352	165524352	0	0.70164
168481351	7	168455232	168455232	0	0.17721
169739593	3	169711360	169711360	0	1.4388
170636503	7	170609760	170609760	0	3.9999
172207699	3	172177864	172177864	0	11.424

n	e	$\phi(n)_{rev}$	$\phi(n)$	$\phi(n)_{rev} - \phi(n)$	Runtime (s)
173966131	3	173937064	173937064	0	11.835
174306749	7	174280320	174280320	0	5.9489
176096941	3	176070400	176070400	0	1.1747
178455401	7	178424532	178424532	0	0.37067
182739127	7	182710440	182710440	0	1.3701
182766319	7	182737944	182737944	0	2.0466
182867513	7	182840448	182840448	0	0.52069
183244433	7	183211248	183211248	0	3.0322
184543937	7	184511316	184511316	0	12.506
187799429	7	187766976	187766976	0	3.4021
190534919	7	190499040	190499040	0	3.2051
195217963	7	195186240	195186240	0	0.49844
198632059	3	198597904	198597904	0	6.6442
214269259	7	214239600	214239600	0	0.79801
219730289	7	219697380	219697380	0	0.26326
222569047	3	222538576	222538576	0	0.29209
222592537	7	222558516	222558516	0	1.6599
225387007	3	225354040	225354040	0	14.967
225889201	3	225859072	225859072	0	0.9363
233979631	7	233948232	233948232	0	5.2195
234903083	7	234871728	234871728	0	8.0473
236455987	3	236424376	236424376	0	16.357
237022991	7	236988432	236988432	0	0.0016582
237362183	7	237324840	237324840	0	5.3126
237915547	3	237881896	237881896	0	7.8839
240048101	7	240016992	240016992	0	7.9556
248677969	3	248645440	248645440	0	4.2835
253900613	7	253867020	253867020	0	5.5963
255590407	3	255557920	255557920	0	17.068
268419007	3	268385800	268385800	0	8.9029
270480233	7	270447060	270447060	0	18.007
273349669	3	273312640	273312640	0	9.147
277188547	7	277155216	277155216	0	0.18124
277418213	7	277384800	277384800	0	9.2195
280617823	7	280582128	280582128	0	1.5836
285017417	7	284981760	284981760	0	1.1936
290060471	7	290021592	290021592	0	2.7767
290458079	7	290419800	290419800	0	19.228
291692963	7	291658800	291658800	0	0.017815
296912501	7	296877504	296877504	0	3.2972
298053169	7	298016496	298016496	0	3.2853
310189357	7	310153536	310153536	0	1.1879
324842003	7	324803448	324803448	0	21.906
325690663	7	325654560	325654560	0	0.00262
335380819	3	335344096	335344096	0	22.6
336277171	7	336236472	336236472	0	0.45205
343117351	3	343079104	343079104	0	5.6948

n	e	$\phi(n)_{rev}$	$\phi(n)$	$\phi(n)_{rev} - \phi(n)$	Runtime (s)
356274917	7	356236320	356236320	0	5.9276
359696171	7	359658144	359658144	0	1.4918
363709987	7	363671640	363671640	0	4.0597
372878453	7	372836640	372836640	0	1.2625
387062051	7	387022704	387022704	0	4.2928
399917993	7	399877968	399877968	0	4.4239
402817097	7	402773700	402773700	0	8.8959
402865937	7	402825780	402825780	0	0.86695
411245309	7	411203100	411203100	0	9.1242
416281483	7	416239200	416239200	0	9.1915
417595847	7	417552696	417552696	0	14.072
420303691	3	420262672	420262672	0	7.0016
437545349	7	437502636	437502636	0	29.171
450933727	3	450890776	450890776	0	15.015
455193419	7	455149944	455149944	0	31.549
463958983	3	463913680	463913680	0	4.1178
482505559	7	482460840	482460840	0	10.914
500448787	3	500403976	500403976	0	35.843
504222679	7	504176400	504176400	0	1.3722
533741213	7	533695008	533695008	0	9.7099
559614493	3	559566940	559566940	0	20.32
560385809	7	560337204	560337204	0	0.34047
571110181	7	571061664	571061664	0	6.6785
575645573	7	575597568	575597568	0	19.93
603092363	7	603042720	603042720	0	10.226
614123129	7	614072676	614072676	0	14.179
628834919	7	628784520	628784520	0	0.63923
660824441	7	660772752	660772752	0	7.4881
662630249	7	662578500	662578500	0	7.5057
687999493	3	687947008	687947008	0	23.486
689052583	3	689000080	689000080	0	24.003
700822051	3	700769104	700769104	0	47.435
723309439	7	723255624	723255624	0	16.596
794768321	7	794711892	794711892	0	27.003
802889363	7	802832688	802832688	0	55.013
847628293	3	847570060	847570060	0	57.529

B RSA Proof

Theorem (RSA). Suppose that $n = pq$ where p and q are distinct primes. Suppose that e is selected so that $\gcd(e, \phi(n)) = 1$ and that $d = e^{-1} \bmod \phi(n)$. Then for any element $m \in \mathbb{Z}_n$ we have

$$m^{ed} \equiv m \pmod{n}$$

Proof. To get started, note that since $d = e^{-1} \bmod \phi(n)$ we know that $ed = 1 \bmod \phi(n)$. Thus,

$$ed = 1 + k\phi(n)$$

for some integer k. We will now use this to prove that $m^{ed} = m \bmod n$, no matter what m is. The proof falls into several cases, depending on whether or not p or q happen to divide your message m. (Note that if *both* p and q happen to divide m then $m = 0 \bmod n$, in which case it is obvious that $m^{ed} = m \bmod n$: any power of 0 is still 0.)

Case 1: Neither p nor q is a factor of m.

In this case m can have no common factors with $n = pq$. Euler's theroem tells us that $m^{\phi(n)} = 1 \bmod n$. Hence we may compute $m^{ed} \bmod n$ as follows:

$$\begin{aligned} m^{ed} &= m^{1+k\phi(n)} \\ &= m \cdot m^{k\phi(n)} \\ &= m \cdot \left(m^{\phi(n)}\right)^k \\ &= m \cdot 1^k \\ &= m. \end{aligned}$$

Case 2: q is a factor of m but p is not.

Since q is a factor of m we have $m = 0 \bmod q$. Then

$$m^{ed} = 0^{ed} = 0 = m \bmod q.$$

Since p is *not* a factor of m, Euler's Theorem also tells us that $m^{p-1} = 1 \bmod p$. We may compute $m^{ed} \bmod p$ as follows:

$$\begin{aligned} m^{ed} &= m^{1+k\phi(n)} \\ &= m^{1+k(p-1)(q-1)} \\ &= m \cdot m^{k(p-1)(q-1)} \\ &= m \cdot \left(m^{p-1}\right)^{k(q-1)} \\ &= m \cdot 1^{k(q-1)} \\ &= m. \end{aligned}$$

We have just argued that $m^{ed} = m \bmod p$ and $m^{ed} = m \bmod q$, which is what we wanted.

Case 3: p is a factor of m but q is not.

The argument for this case is just like the previous, but with the roles of p and q reversed.

Thus in all possible cases $m^{ed} \equiv m \pmod n$, which is what we wanted to prove. □

References

[1] Andysah Putera Utama Siahaan, E Elviwani, and Boni Oktaviana, *Comparative analysis of rsa and elgamal cryptographic public-key algorithms*, Proceedings of the Joint Workshop KO2PI and the 1st International Conference on Advance amp; Scientific Innovation (Brussels, BEL), ICASI'18, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2018, p. 163–172.

[2] Wen WAN Xingbo WANG, Zhikui DUAN1, *Some new inequalities with proofs and comments on applications*, vol. 10, Canadian Center of Science and Education, 2018.