RSA CRYPTOSYSTEM: AN ANALYSIS AND PYTHON SIMULATOR

by

Cescily Nicole Metzgar

Honors Thesis

Appalachian State University

Submitted to the Department of Mathematical Sciences
and The Honors College
in partial fulfillment of the requirements for the degree of

Bachelor of Science

May, 2017

Approved by:

Rick Klima, Ph.D., Thesis Director

Dee Parks, Ph.D., Second Reader

Vicky Klima, Ph.D., Honors Director, Department of Mathematical Sciences

Ted Zerucha, Ph.D., Interim Director, The Honors College

Abstract

This project involves an exploration of the RSA cryptosystem and the mathematical concepts embedded within it. The first goal is to explain what the cryptosystem consists of, and why it works. Additional goals include detailing some techniques for primality testing, discussing integer factorization, modular exponentiation, and digital signatures, and explaining the importance of these topics to the security and efficiency of the RSA cryptosystem. The final goal is to implement all of these components into a full simulation of the entire RSA cryptosystem using the Python programming language.

Contents

1 Background

The RSA cryptosystem was created by three MIT professors, Ron Rivest, Adi Shamir, and Len Adleman and published in an article named *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems* in 1978. While the cryptosystem is named for this trio of mathematicians, it is less widely known that a man named Clifford Cocks had actually discovered the algorithm first while working in a classified environment for the British cryptologic agency GCHQ. Clifford Cock's discovery of the RSA algorithm was revealed more than two decades after the publication by Rivest, Shamir and Adleman [Klima, Sigmon].

RSA was the world's first public-key cryptosystem, which is part of why the algorithm is so well-known and popular. Being a public-key cryptosystem stems from being asymmetric. This means that the encryption key is made public knowledge and does not in any way give clues to an outsider or even the person sending the message about how to obtain the decryption key. In a more formalized assertion, a public key cryptosystem is a system in which the encryption function f can be public knowledge without revealing f^{-1}. This inability to determine f^{-1} from the encryption function f comes from the practical difficulty of factoring large primes [Klima, Sigmon]. This factorization problem allows RSA to be an extremely secure cryptosystem. Because of its security it is most widely used today to provide privacy and ensure the authenticity of digital data. RSA is implemented by web servers and browsers to secure web traffic, it is used to ensure privacy and authenticity of email, and it is frequently used in electronic credit card payment systems. These are all applications in which security of digital data is of extreme importance, which exemplifies the high level of security RSA can provide. While RSA is extremely secure, the mathematics that underlie the system are fairly simple as will be shown.

2 How RSA Works

It is important to note that RSA depends on the numerical conversion of a message. To do this, one can map the letters of the alphabet to a corresponding element in the ring $\mathbb{Z}_{26}$. This mapping can be outlined as follows: A $\mapsto 0$, B $\mapsto 1$, C $\mapsto 2$, $\ldots$, Z $\mapsto 25$ [Klima, Sigmon]. The only setback to this method is that only capital letters can be used in creating messages to send. To improve upon this method, ASCII representations of letters, symbols and spaces can be used. With ASCII, lower and upper case letters have different numerical representations, so messages converted using ASCII are able to use upper and lower case letters as well as spaces and symbols. All of these elements within a message are preserved throughout the encryption and decryption process. In the Python program included as part of this thesis, this is the method used to convert strings of text to their numerical representations. Any way you do it, you must convert the string of characters that make up your messages into a numerical equivalent. We let this numerical message be x. To get started with the RSA encryption algorithm, we must first choose two distinct prime numbers p and q. We then must determine $n = pq$ as well as $m = (p-1)(q-1)$. Next we would need to determine an encryption exponent $a \in \mathbb{Z}_m^*$ which satisfies $\gcd(a, m) = 1$. A decryption exponent $b \in \mathbb{Z}_m^*$ will also need to be found that satisfies $ab = 1 \bmod m$. We cannot choose any number as our a because in order for RSA to work, a must be relatively prime to m. This in turn allows us to be able to find a value of b that when multiplied by a yields the value $1 \bmod m$. These two conditions on a and b are extremely important. When met, they force the following equation to be true: $x^{ab} = x \bmod n$. This equation shows that once we raise a plaintext message to the encryption exponent, we can raise the resulting ciphertext to the decryption exponent then reduce mod n and the same plaintext message x will be the end result [Klima, Sigmon].

Finding an encryption exponent a that is relatively prime to a chosen m is fairly straightforward. Once an a is found, finding the decryption exponent b requires the use of the Euclidean algorithm. The Euclidean algorithm can also be used to confirm that your choice of a is indeed relatively prime to m as well. So, knowing the Euclidean algorithm and how to implement it is fairly important when looking for parameters that will allow the RSA algorithm to work.

Suppose you want to receive RSA encrypted messages, so you must first generate the keys. For the sake of this first example very small numbers will be used to show how the Euclidean algorithm works to help choose the encryption and decryption exponents. Suppose you select primes $p = 47$ and $q = 61$. This would give $n = pq = 47 \cdot 61 = 2867$. You must then calculate $m = (p-1)(q-1)$, which is $(47-1)\cdot(61-1)$, or $46 \cdot 60 = 2760$. Now you would need to establish an a such that a and 2760 have no common divisors greater than 1. Suppose you choose $a = 67$. Now you can use the Euclidean algorithm to prove that this choice of a is actually relatively prime to m as follows. Begin by dividing m by a, while noting the quotient and remainder. This process will then be repeated, each time using the divisor from the previous step and then dividing that by the remainder from the previous step. The last nonzero remainder in this process is the gcd of a and m:

$$\begin{aligned} 2760 &= 67(41) + 13 \\ 67 &= 13(5) + 2 \\ 13 &= 2(6) + 1 \\ 2 &= 1(2) + 0. \end{aligned}$$

Because the last nonzero remainder is 1, the gcd of 67 and 2760 is 1, and so the two numbers are relatively prime, which justifies the choice of a. Now we can find a valid decryption exponent. We are looking for a value of b that satisfies $ab = 1 \bmod m$. In other words we need a value of b such that $67b = 1 \bmod 2760$. The Euclidean algorithm equations used to validate our choice of a can also be used to find b as well. The Euclidean algorithm equations can be rewritten in a way that allows us to work backwards to find the multiplicative inverse of our $a \bmod m$. This can be done as follows:

$$\begin{aligned} 1 &= 13 - 2(6) \\ &= 13 - (67 - 13(5))(6) \\ &= -67(6) + 13(31) \\ &= -67(6) + (2760 - 67(41))(31) \\ &= 2760(31) + 67(-1277). \end{aligned}$$

The last line gives $2760(31) + 67(-1277) = 1$, which can be reduced mod 2760, giving the result $67(-1277) = 1 \bmod 2760$. This tells us that -1277 is a multiplicative inverse of 67 mod 2760. However, we do not want to use a negative number for b, but since we are working mod 2760, we can see that $-1277 \bmod 2760 = 1483$. This means that 1483 can work as our decryption exponent b. Now that we have determined valid encryption and decryption exponents, we have obtained all the information we need to make the encryption exponent and modulus n public and begin receiving encrypted messages that only our decryption exponent b can decrypt. We can now do some examples using RSA with our parameters $n = 2867$, encryption exponent $a = 67$, and decryption exponent $b = 1483$.

Example 1: Encryption

Suppose someone wants to send us a message, specifically the message *URGENT: Meet at dusk!*. First, the sender must convert their message into its numerical equivalent. This can be done using the ASCII correspondences shown in Table 1. Using this method, the message *URGENT: Meet at dusk!* converts into the following string of numbers:

$$085082071069078084058032077101101116032097116032100117115107033.$$

At this point, the sender will use the encryption calculation to encrypt the message. Recall that x represents the message to be encrypted. The encryption calculation is: $x^a \bmod n$. Because the message is such a large number, we must split it into pieces and do separate calculations in order to complete the encryption process. The pieces of the message to be encrypted cannot be bigger than $n = 2867$ in this example, or else they could not be decrypted correctly, and so we will split the message into groups of three digits. In other words, we must encrypt one character at a time. This splits our plaintext into the groups 085, 082, 071, 069, 078, 084, 058, 032, 077, 101, 101, 116, 032, 097, 116, 032, 100, 117, 115, 107, 033. Now the sender can take these pieces of the plaintext and use the encryption calculation to find the ciphertext as follows:

$$\begin{aligned}
085^{67} \bmod 2867 &= 1533 \\
082^{67} \bmod 2867 &= 2663 \\
071^{67} \bmod 2867 &= 1978 \\
069^{67} \bmod 2867 &= 2595 \\
078^{67} \bmod 2867 &= 884 \\
084^{67} \bmod 2867 &= 525 \\
058^{67} \bmod 2867 &= 1168 \\
032^{67} \bmod 2867 &= 578
\end{aligned}$$

Decimal Representation	Character	Decimal Representation	Character
32	Space	80	P
33	!	81	Q
34	”	82	R
35	#	83	S
36	$	84	T
37	%	85	U
38	&	86	V
39	’	87	W
40	(	88	X
41	)	89	Y
42	*	90	Z
43	+	91	[
44	,	92	\
45	−	93	]
46	.	94	ˆ
47	/	95	_
48	0	96	‘
49	1	97	a
50	2	98	b
51	3	99	c
52	4	100	d
53	5	101	e
54	6	102	f
55	7	103	g
56	8	104	h
57	9	105	i
58	:	106	j
59	;	107	k
60	<	108	l
61	=	109	m
62	>	110	n
63	?	111	o
64	@	112	p
65	A	113	q
66	B	114	r
67	C	115	s
68	D	116	t
69	E	117	u
70	F	118	v
71	G	119	w
72	H	120	x
73	I	121	y
74	J	122	z
75	K	123	{
76	L	124	|
77	M	125	}
78	N	126	∼
79	O		

Table 1: ASCII Correspondences

$$\begin{aligned}
077^{67} \bmod 2867 &= 2699 \\
101^{67} \bmod 2867 &= 1058 \\
101^{67} \bmod 2867 &= 1058 \\
116^{67} \bmod 2867 &= 1091 \\
032^{67} \bmod 2867 &= 578 \\
097^{67} \bmod 2867 &= 2794 \\
116^{67} \bmod 2867 &= 1091 \\
032^{67} \bmod 2867 &= 578 \\
100^{67} \bmod 2867 &= 1286 \\
117^{67} \bmod 2867 &= 748 \\
115^{67} \bmod 2867 &= 1726 \\
107^{67} \bmod 2867 &= 710 \\
033^{67} \bmod 2867 &= 613.
\end{aligned}$$

The sender can then send the string of ciphertext numbers 1533, 2663, 1978, 2595, 884, 525, 1168, 578, 2699, 1058, 1058, 1091, 578, 2794, 1091, 578, 1286, 748, 1726, 710, 613 to us. It is important to note that with such a small value of n forcing such small groupings of plaintext digits, this particular encryption is insecure, as it is vulnerable to being broken by frequency analysis. In practice, the primes chosen to form n are extremely large and would help to eliminate this insecurity.

Example 1: Decryption

After we receive the encrypted message formed above, we can then decrypt the message using the decryption key. This is how the decryption process works. Suppose we refer to the encrypted message as c. The decryption calculation is $c^b \bmod 2867$. This must be done for each of the ciphertext numbers in order to reveal the message as follows:

$$\begin{aligned}
1533^{1483} \bmod 2867 &= 085 \\
2663^{1483} \bmod 2867 &= 082
\end{aligned}$$

$$\begin{aligned}
1978^{1483} \bmod 2867 &= 071 \\
2595^{1483} \bmod 2867 &= 069 \\
884^{1483} \bmod 2867 &= 078 \\
525^{1483} \bmod 2867 &= 084 \\
1168^{1483} \bmod 2867 &= 058 \\
578^{1483} \bmod 2867 &= 032 \\
2699^{1483} \bmod 2867 &= 077 \\
1058^{1483} \bmod 2867 &= 101 \\
1058^{1483} \bmod 2867 &= 101 \\
1091^{1483} \bmod 2867 &= 116 \\
578^{1483} \bmod 2867 &= 032 \\
2794^{1483} \bmod 2867 &= 097 \\
1091^{1483} \bmod 2867 &= 116 \\
578^{1483} \bmod 2867 &= 032 \\
1286^{1483} \bmod 2867 &= 100 \\
748^{1483} \bmod 2867 &= 117 \\
1726^{1483} \bmod 2867 &= 115 \\
710^{1483} \bmod 2867 &= 107 \\
613^{1483} \bmod 2867 &= 027.
\end{aligned}$$

If we convert these decrypted numbers back into characters according the ASCII correspondences in Table 1, we see that it converts back into the original plaintext *URGENT: Meet at dusk!*.

In Example 1 the size of our encrypted blocks was only three digits, meaning that each character was encrypted and decrypted separately. This is ineffective and insecure, since encrypting one character at a time yields a substitution cipher which can be easily broken through frequency analysis. To really take advantage of the security RSA can offer, we must encrypt larger groupings of digits. However, in order to be decrypted, the groupings encrypted cannot be larger than the value of n. So, in general, taking advantage of the security RSA can offer requires a large value for n. In the following we will do this same example, but with larger groupings in order to show the encryption and decryption process with more secure parameters.

Example 2: Encryption

We will use the same plaintext in this example, *URGENT: Meet at dusk!*, which converted under the ASCII correspondences into the following string of numbers:

$$085082071069078084058032077101101116032097116032100117115107033.$$

In this example we will use primes $p = 1009$ and $q = 1511$, which give $n = 1524599$ and $m = 1522080$. A possible value for a is 15221, which we can confirm is a valid encryption exponent as follows:

$$\begin{aligned}
1522080 &= 15221(99) + 15201 \\
15221 &= 15201(1) + 20 \\
15201 &= 20(760) + 1 \\
20 &= 1(20) + 0.
\end{aligned}$$

Since the last nonzero remainder is 1, the gcd of 15221 and 1522080 is 1, and so the two numbers are relatively prime, justifying our choice of a. We can rewrite these equations to find a valid value for b as follows:

$$\begin{aligned}
1 &= 15201 - 20(760) \\
&= 15201 - (15221 - 15201(1))(760) \\
&= -15221(760) + 15201(761) \\
&= -15221(760) + (1522080 - 15221(99))(761) \\
&= 1522080(761) + 15221(-76099).
\end{aligned}$$

The last line gives $1522080(761) + 15221(-76099) = 1$, which can be reduced mod 1522080, giving $15221(-76099) = 1 \bmod 1522080$. This tells us that a value that works for b is -76099 mod

$1522080 = 1445981$. Now we can begin the encryption process. With our value of $n = 1524599$, we can make plaintext groups of two characters for the encryption and decryption process, which will have equivalent numerical lengths of 6 digits. With two characters in each group, our plaintext numbers will be 085082, 071069, 078084, 058032, 077101, 101116, 032097, 116032, 100117, 115107, 033120. Since there are an odd number of characters in our message, our last group only had one character. To maintain consistency with the group size, we have added a single random character to the end of our message, namely x. We can now encrypt our message via the following calculations:

$$\begin{aligned}
085082^{15221} \bmod 1524599 &= 508128 \\
071069^{15221} \bmod 1524599 &= 259410 \\
078084^{15221} \bmod 1524599 &= 1505416 \\
058032^{15221} \bmod 1524599 &= 1516259 \\
077101^{15221} \bmod 1524599 &= 812195 \\
101116^{15221} \bmod 1524599 &= 552080 \\
032097^{15221} \bmod 1524599 &= 743297 \\
116032^{15221} \bmod 1524599 &= 1127324 \\
100117^{15221} \bmod 1524599 &= 978406 \\
115107^{15221} \bmod 1524599 &= 336239 \\
033120^{15221} \bmod 1524599 &= 1330515.
\end{aligned}$$

Thus the ciphertext is the list of numbers 1459719, 792327, 729432, 475592, 528380, 999910, 1080674, 55252, 943425, 255531, 633201.

Example 2: Decryption

After we receive the encrypted message formed above, we can then decrypt the message using the decryption key as follows:

$$508128^{1445981} \bmod 1524599 = 085082$$

$$\begin{aligned}
259410^{1445981} \bmod 1524599 &= 071069\\
1505416^{1445981} \bmod 1524599 &= 078084\\
1516259^{1445981} \bmod 1524599 &= 058032\\
812195^{1445981} \bmod 1524599 &= 077101\\
552080^{1445981} \bmod 1524599 &= 101116\\
743297^{1445981} \bmod 1524599 &= 032097\\
1127324^{1445981} \bmod 1524599 &= 116032\\
978406^{1445981} \bmod 1524599 &= 100117\\
336239^{1445981} \bmod 1524599 &= 115107\\
1330515^{1445981} \bmod 1524599 &= 033120.
\end{aligned}$$

If we convert these decrypted numbers back into characters according the ASCII correspondences in Table 1, we see that it converts back into the original plaintext *URGENT: Meet at dusk!*.

Example 2 eliminates the possibility for cryptanalysis using frequency analysis and is in general more secure than Example 1. Example 2 is, however, still on an extremely small scale compared to the parameters used in actual practice. While not obvious how, it would not take much time for an outsider to find the prime factors p and q that created our n, which would in turn expose the algorithm to outsiders. To obtain the high level of security RSA is known for, modern applications use extremely large primes that are practically impossible to find from n, even with immense computing power. As computing power is ever evolving and improving, so has the minimum key length to consider an RSA algorithm truly secure. Today, 2048, 3072 or 4096 bit keys are typically used. This corresponds to about 617, 925, and 1233 digits, respectively. This obviously makes our 7 digit n seem very minuscule.

3 Why RSA Works

Several fundamental theorems comprise the basic foundation for why the RSA cryptosystem works. Among these are Lagrange's Theorem and Fermat's Little Theorem.

Lagrange's Theorem: For a finite group G with identity element e, if $|G| = k$ and $g \in G$, then $g^k = e$.

Fermat's Little Theorem is a corollary to Lagrange's Theorem, in the case where G is the set of nonzero elements in $\mathbb{Z}_p$ for prime p under multiplication mod p. Since it is known that the nonzero elements in any finite field form a group under multiplication, and $\mathbb{Z}_p$ for prime p is a finite field, it follows that $G = \mathbb{Z}_p^*$ is a group under multiplication mod p.

Fermat's Little Theorem: Let p be a prime, and suppose $x \in \mathbb{Z}$ satisfies $\gcd(x, p) = 1$. Then $x^{p-1} = 1 \bmod p$.

Proof: Consider the group $\mathbb{Z}_p^*$ for prime p under multiplication mod p. Note that $|\mathbb{Z}_p^*| = p-1$, and $\mathbb{Z}_p^*$ has identity element $e = 1$. For any $x \in \mathbb{Z}$ that satisfies $\gcd(x, p) = 1$, it follows that x is not a multiple of p, and so $y = x \bmod p \in \mathbb{Z}_p^*$. But then by Lagrange's Theorem, we have $x^{p-1} = y^{p-1} \bmod p = 1 \bmod p$.

Finally, the following theorem verifies the mathematical fact that makes the RSA cryptosystem work.

Theorem: Let p and q be distinct primes, and suppose $n = pq$ and $m = (p-1)(q-1)$. If a and b are integers with $ab = 1 \bmod m$, then $x^{ab} = x \bmod n$ for all $x \in \mathbb{Z}_n$.

Proof: If $ab = 1 \bmod m$, then $ab = 1 + km$ for some $k \in \mathbb{Z}$, and for all $x \in \mathbb{Z}_n$ the following will hold:

$$x^{ab} = x^{1+km} = x(x^{km}) = x(x^{p-1})^{k(q-1)}.$$

If $\gcd(x, p) = 1$, then by Fermat's Little Theorem we know that $x^{p-1} = 1 \bmod p$. Thus, $x^{ab} = x(1)^{k(q-1)} \bmod p = x \bmod p$. Also, if $\gcd(x, p) \neq 1$, then $x = 0 \bmod p$, and certainly $x^{ab} = x \bmod p$. Similarly, $x^{ab} = x \bmod q$ for all $x \in \mathbb{Z}$. Thus, $p | x^{ab} - x$ and $q | x^{ab} - x$, and so $pq | x^{ab} - x$. That is, $n | x^{ab} - x$, or, equivalently, $x^{ab} = x \bmod n$ [Klima, Sigmon].

4 Why RSA is Public-Key

The fact that RSA is a public-key cryptosystem means users of an RSA cipher can assume that almost everything related to the cipher is public knowledge, including not just the form of the encryption calculations, but the fixed parameters in the encryption calculations as well. This means that if an outsider were to intercept an RSA encrypted ciphertext, they could know not only that each ciphertext integer was formed as $x^a \bmod n$ for some plaintext integer x and positive integers a and n, but they could actually know the values of a and n [Klima, Sigmon]. This may seem like an extreme threat to the security of the RSA cryptosystem. However, with sufficiently large choices for the primes p and q used to form $n = pq$, this is not the case.

While one may be skeptical of the security of a cryptosystem that offers so much information to the public, we must think about what it is that makes RSA as secure as it is in practice. Even in possession of a and n, a piece of information necessary for decryption that an outsider would be missing is the decryption exponent b. As we have seen, b is found as a multiplicative inverse of a mod m, i.e., $ab = 1 \bmod m$. The only way an outsider could crack the system is to somehow manage to find such a value of b from only the knowledge of a and n. But to find b, the outsider would have to first find m to know what modulus to use in the equation $ab = 1 \bmod m$. And to find $m = (p-1)(q-1)$, the outsider would need to know p and q. It is the difficulty of finding p and q from n that provides RSA with its extremely high level of security. The sad reality for outsiders is that with extremely large values for p and q, factoring $n = pq$ is essentially impossible. For example, if p and q were both hundreds of digits long, then the fastest known factoring algorithms would in general take millions of years to factor $n = pq$, even when programmed on a computer that could perform millions of operations per second [Klima, Sigmon]. So, even with a and n being public knowledge, an outsider should not be able to determine the decryption exponent b. This is precisely why the RSA cryptosystem is a public-key system. Factoring n is not a problem for the intended recipient of an RSA ciphertext though, since as in our Examples 1 and 2, the recipient starts the process by choosing p and q used to form n. And of course the tremendous benefit to RSA being public-key to the users of an RSA cipher is that they do not have to figure out a way to securely exchange an encryption exponent and modulus for the cipher. Rather, they can just make them public knowledge.

Since RSA's security relies on extremely large primes, it is important to note that by Euclid's Theorem we know that the primes are unbounded in size and number. This theorem tells us that even with greater and greater advances of modern computing power, there will always be larger and larger primes that we can use to continue ensuring the security of the RSA cryptosystem.

Theorem (Euclid's Theorem): There are infinitely many primes.

Proof: For the sake of contradiction, suppose there are finitely many primes, with the following being a complete list: $p_1, p_2, \ldots, p_n$. Assume without loss of generality that $p_1 < p_2 < \cdots < p_n$. Consider $M = p_1 p_2 \cdots p_n + 1$. Then M is certainly not prime, since it is larger than p_n. Since M is not prime, it must be divisible by at least one of the primes in our complete list, say p_j. That is, p_j divides M. But p_j certainly divides $M - 1$, and thus it must be the case that $p_j = 1$. This is a contradiction, however, since we are assuming that p_j is prime. $\rightarrow\leftarrow$ Therefore, there cannot be only finitely many primes, and thus there must be infinitely many primes.

5 Primality Testing

The security of RSA relies on the use of very large primes, however finding primes large enough to make RSA so secure is not particularly easy. Motivated in part by the development of public-key cryptosystems like RSA, much research has been done over the past few decades in the area of primality testing. Contrary to the name, primality testing usually focuses on criteria that prove a number is not prime rather than criteria that prove a number is prime [Klima, Sigmon]. In failing to find evidence that a number is not prime, we can then trust that it is prime. This is how most methods for primality testing work. The most direct and accurate method for testing the primality of an odd integer n is to find nontrivial factors of n by trial and error. This could be done systematically by checking if $m|n$ as m takes on odd integer values starting with $m = 3$ and ending when m reaches $\sqrt{n}$ [Klima, Sigmon]. This method, however, is extremely inefficient if n is a very large number. For instance, if we were testing $n = 1697835486935464783165465534684354687$ for primality, we would have to compute $1.30301016379 \times 10^{18}$ divisions. This is a problem because we need very large primes for RSA to be secure, but we also want to be as efficient as possible when developing the RSA algorithm.

One simple primality test is based on Fermat's Little Theorem. If n is a prime integer, then as a consequence of Fermat's Little Theorem it will be true that $a^{n-1} = 1 \bmod n$ for all $a \in \mathbb{Z}_n^*$. As a result, if $a^{n-1} \neq 1 \bmod n$ for any $a \in Z_n^*$, we can conclude that n is definitely not prime. Thus, we can test the primality of an integer n by checking if $a^{n-1} = 1 \bmod n$ for some values of a in $\mathbb{Z}_n^*$, with the power of the test increasing as we check more values of a [Klima, Sigmon].

This fairly simple primality test is the one used in the Python simulator created for this thesis. The simulator tests about 40 values of a, yielding results virtually certain to be accurate yet still very efficient. A drawback to this primality test is that there are some values of a for which $a^{n-1} = 1 \bmod n$ even when $\gcd(a, n) = 1$ and n is not prime. In such cases, n is called a pseudoprime to the base a. This may seem worrisome as to the accuracy of the Fermat primality test, but pseudoprimes are extremely scarce compared to primes. For example, there are only 245 pseudoprimes to the base 2 less than one million, while there are 78,498 primes less than one million. Also, most pseudoprimes to the base 2 are not pseudoprimes to many other bases [Klima, Sigmon]. This means that if a number is pseudoprime to the base 2 but not to another base a, the Fermat test would still identify the number as not prime as long as that a value is checked, hence the increased accuracy with the inclusion of more a values.

However, there are nonprime integers n that are pseudoprime to every positive base $a < n$ with $\gcd(a, n) = 1$. These numbers are called Carmichael numbers. There are 2163 Carmichael numbers less than 25 billion [Klima, Sigmon]. As can be seen, these numbers are fairly rare. Given a randomly chosen odd integer n less than 10^{17}, the probability that n is a Carmichael number is only a little over $\frac{1}{10^{11}}$ (about one in one hundred billion) [Rabin-Miller]. The smallest Carmichael number is 561. Using 561 as a quick example, we can see that it is pseudoprime to all of the following choices of a which satisfy $\gcd(a, n) = 1$:

$$\begin{aligned} a = 2 &\quad\Longrightarrow\quad 2^{560} = 1 \bmod 561 \\ a = 13 &\quad\Longrightarrow\quad 13^{560} = 1 \bmod 561 \\ a = 40 &\quad\Longrightarrow\quad 40^{560} = 1 \bmod 561 \\ a = 65 &\quad\Longrightarrow\quad 65^{560} = 1 \bmod 561. \end{aligned}$$

Note that this is only 4 choices for a, but the same result will hold with any choice of $a < n$ with $\gcd(a, n) = 1$.

Another well-known primality test is the Euler Test, which is based on the fact that if n is an odd prime, an integer can have at most two square roots mod n. In particular, the only square roots of 1 mod n are ± 1. Thus, if $a = 0 \bmod n$, then $a^{(n-1)/2}$ is a square root of $a^{(n-1)} = 1 \bmod n$, and $a^{(n-1)/2} = \pm 1 \bmod n$. So, Euler's test tells us that if $a^{(n-1)/2} \neq \pm 1 \bmod n$ for some a with $a \neq 0 \bmod n$, then n is composite. The Euler test improves upon the Fermat test. It is true that if the Fermat test finds that n is composite, the Euler test will as well. However, the Euler test may find that n is composite even when the Fermat test fails to do so. This can happen in the case of certain n values if n is an odd composite integer (other than a prime power), because 1 will have at least 4 square roots mod n. In this case we can have $a^{(n-1)/2} = \beta \bmod n$, where $\beta \neq \pm 1$ is a square root of 1. Then $a^{n-1} = 1 \bmod n$. In this situation, the Fermat Test (incorrectly) declares n a probable prime, but the Euler test (correctly) declares n composite [Rabin-Miller]. Some of the Carmichael numbers that were mentioned as issues for the Fermat test can actually be labeled as composite with the Euler test. The following table shows a comparison of the Fermat and Euler tests with the seven Carmichael numbers under 10000.

n	$\phi(n)$	Number of a with $a^{n-1} = 1 \bmod n$	Number of a with $a^{(n-1)\ 2} = \pm 1 \bmod n$
561	320	320	160
1105	768	768	364
1729	1296	1296	1296
2465	1792	1792	1792
2881	2160	2160	1080
6601	5280	5280	2640
8911	7128	7128	1782

In each case, the Fermat test will falsely identify the Carmichael number as prime because $a^{n-1} = 1 \bmod n$ for every a with $\gcd(a, n) = 1$, the number of which is given by $\phi(n)$. The Euler test, however, identifies five of the seven Carmichael numbers as composite, if the right values of a are tested. The two Carmichael numbers that cause the Euler test to fail, 1729 and 2465, are called absolute Euler pseudoprimes. There are fewer absolute Euler pseudoprimes than Carmichael numbers, so the Euler test is considered more accurate than Fermat's [Rabin-Miller].

Another primality test that is widely known is the Rabin-Miller test, that improves even on Euler's test. The limitation of the Euler test is that it does not go to any special effort to find square roots of 1 different from ± 1. The Rabin-Miller test does do this. In the Rabin-Miller test, we use $n-1 = 2^s m$, with m odd and $s \geq 1$. To start the Rabin-Miller test, we compute $a^m \bmod n$. If $a^m = \pm 1 \bmod n$, we declare n a probable prime, and stop. This is because we know that $a^{n-1} = (a^m)^{2^s} = 1 \bmod n$, and we will not find a square root of 1, other than ± 1, in repeated squaring of a^m to get a^{n-1}. So, if $a^m \neq \pm 1 \bmod n$, we square $a^m \bmod n$ to obtain a^{2m}, unless $s = 1$. If $a^{2m} = 1 \bmod n$, we declare n composite, and stop. This is due to a^m being a square root of $a^{2m} = 1 \bmod n$, different from ± 1. If $a^{2m} = -1 \bmod n$, we declare n a probable prime, and stop. This is because, similarly to the previous, we know that $a^{n-1} = 1 \bmod n$, and we will not find a square root of 1, other than ± 1. If neither of these are the case, unless $s = 2$, we square $a^{2m} \bmod n$ to obtain $a^{2^2 m}$. If $a^{2^2 m} = 1 \bmod n$, we declare n composite, and stop. If $a^{2^2 m} = -1 \bmod n$, we declare n a probable prime, and stop. Otherwise we continue in this manner until we either we stop the test, or we have computed $a^{2^{s-1} m}$, and stopped if $a^{2^{s-1} m} = a^{(n-1)/2} = \pm 1 \bmod n$ [Rabin-Miller].

If we take the Euler absolute pseudoprime 1729, with $a = 671$, the Rabin-Miller test proceeds as follows. Since $1729 - 1 = 1728 = 2^6(27)$, then $s = 6$ and $m = 27$. Then we have the following:

$$\begin{aligned} 671^{27} &= 1084 \bmod 1729 \\ 671^{27(2)} &= 1084^2 \bmod 1729 \\ &= 1065 \bmod 1729 \\ 671^{27(2^2)} &= 1065^2 \bmod 1729 \\ &= 1 \bmod 1729. \end{aligned}$$

The test will then declare n composite and terminate.

If we test a number that is in fact prime, say $n = 104513$, with $a = 3$, the Rabin-Miller test proceeds as follows. Since $n - 1 = 104512 = 2^6(1633)$, then $s = 6$ and $m = 1633$. Then:

$$\begin{aligned} 3^{1633} &= 88958 \bmod n \\ 3^{1633(2)} &= 88958^2 \bmod n \\ &= 10430 \bmod n \end{aligned}$$

$$\begin{aligned}
3^{1633(2^2)} &= 10430^2 \bmod n \\
&= 91380 \bmod n \\
3^{1633(2^3)} &= 91380^2 \bmod n \\
&= 29239 \bmod n \\
3^{1633(2^4)} &= 29239^2 \bmod n \\
&= 2781 \bmod n \\
3^{1633(2^5)} &= 2781^2 \bmod n \\
&= -1 \bmod n.
\end{aligned}$$

We could then conclude that n is a probable prime, but we might perform a few more tests before we are truly convinced that n is actually prime.

Like the Fermat and Euler tests, the Rabin-Miller test has pseudoprimes, with the choices of a with which the test declares a composite integer to be a probable prime. Rabin-Miller pseudoprimes are called strong pseudoprimes. There are fewer strong pseudoprimes than Fermat or Euler pseudoprimes. More importantly, there are no Rabin-Miller absolute pseudoprimes, which are those pseudoprimes that pass an integer off as prime for every a value that can be chosen [Rabin-Miller]. This is what makes the Rabin-Miller test so strong and one of the most commonly used.

6 Integer Factorization

The security of RSA relies on the difficulty of factoring n, which should be the product of two very large distinct primes p and q. This relation between factoring and cryptography is one reason why interest in evaluating the practical difficulty of the integer factorization problem in the mathematical community has increased in recent years. Currently the limits of our factoring capabilities lie around 130 decimal digits [Lenstra]. This is why having a very large n, ideally larger than this upper bound of 130, makes RSA practically unbreakable even with modern computing power. Earlier we saw the sizes of n typically used in practice for the RSA algorithm, namely 2048, 3072, or 4096 bit keys. These bit sizes correspond to about 617, 925, and 1233 decimal digits, respectively, which are obviously all significantly longer than 130 decimal digits. This is precisely why RSA is such a secure cryptosystem. On the other hand, although integer factorization is considered very difficult, especially under the constraints of computing power, there are a few integer factorization methods that we should consider.

The most obvious method for integer factorization involves trial and error. Trial division consists of systematically testing whether n is divisible by any smaller number. However, for efficiency's sake, it would only make sense to see if n were divisible by any prime numbers smaller than it. This is because if we test some number x and find it is not a factor, than any multiple of x will also not be a factor of n. Furthermore, the factors tried need go no higher than $\sqrt{n}$, since if n were divisible by some number r, then $n = r \times s$, and if s were smaller than r, n would have earlier been detected as being divisible by s or a prime factor of s. There are several issues with this method, specifically when working with n values as large as the ones employed in the RSA algorithm. To try and make the trial and error method more efficient we would have to only test prime numbers, but when we are looking at larger and larger sets of possible factors we have to be able to determine which numbers are actually prime before we test them. This brings us back to the difficulty and efficiency issues associated with primality testing. Time is also an issue, as we add more and more digits to n, the time to carry out trial divisions increases exponentially. For these reasons, the trial divisions method is considered extremely inefficient and an insufficient method when dealing with such large n values.

Another well-known method that is fairly simple is Fermat's factorization method. This particular method is extremely useful for factoring integers that are the product of two very large distinct primes that are close together. Let $n = pq$ be the product of two distinct primes, and suppose that we would like to determine the values of p and q from n. This is the heart of the security of RSA. If someone were to find the p and q used in the algorithm, the security would instantly collapse. If p and q were relatively close together, then even if they were both very large, we could determine them fairly quickly using Fermat's factorization as follows. Let $x = \frac{p+q}{2}$ and $y = \frac{p-q}{2}$. Then $n = pq = x^2 - y^2 = (x+y)(x-y)$. Since n has prime factors p and q, it follows that p and q would have to be $x+y$ and $x-y$. To determine p and q, we would only have to find the values of x and y. In order to do this, we could begin by assuming that x is the smallest integer larger than $\sqrt{n}$. Since $n = x^2 - y^2$, if we have assumed the correct value of x, then it will follow that $x^2 - n$ will be the perfect square y^2. If this is not the case, then we would know that we had assumed an incorrect value for x, and we could simply increase x by one and repeat the process until the correct x is found. If p and q are relatively close together, then the number of times that this process would have to be repeated would be relatively small [Klima, Sigmon]. This is exactly why this method is useful in this case.

As an example of Fermat's factorization method, suppose we wanted to find the two prime factors of $n = pq = 108371$. The smallest integer larger than $\sqrt{108371}$ is 330, so this would be our first x. But then $330^2 - n = 529 = 23^2$, and so we have found the correct values for x and y on the first trial, $x = 330$ and $y = 23$. The prime factors of n are then $x + y = 353$ and $x - y = 307$. Because these factors are so close to each other, we were able to find them very quickly.

If we use an n created with primes separated by a little more distance, say $n = 69841$, then we will most likely have to repeat the process more than once. The smallest integer larger than $\sqrt{69841}$ is 265, which will be our first x. Then $265^2 - n = 384$, which is not a perfect square. This means $x = 265$ is not correct, and so now we try $x = 266$. Then $266^2 - n = 915$, which is also not a perfect square. With $x = 267$, we have $267^2 - n = 1448$, which is also not a perfect square. With $x = 268$, we have $268^2 - n = 1983$ which is also not a perfect square. With $x = 269$, we have $269^2 - n = 2520$ which is also not a perfect square. With $x = 270$, we have $270^2 - n = 3059$, which is also not a perfect square. With $x = 271$, we have $271^2 - n = 3600 = 60^2$, and so we have finally found the right values for x and y, 271 and 60, respectively. Our prime factors for 69841 are then $x + y = 331$ and $x - y = 211$. You can see that the farther apart p and q are, the more repetitions of the factorization method are needed to find the right x and y values.

Another well-known method for integer factorization is Pollard's rho method, which is basically a modification of the trial division method that increases the odds of finding a factor of n. The trial division method essentially chooses one number at a time and tests to see if that number is indeed a factor of n. Pollard's rho method chooses k numbers, $\{x_1, \ldots, x_k\}$, and tests whether $\gcd(|x_i - x_j|, n) > 1$. In other words, we ask if $x_i - x_j$ and n have a non-trivial factor in common. This at once increases the number of chances for successes. For example, if we ask how many numbers divide $n = pq$, we have just two: p and q. But if we ask how many numbers satisfy $\gcd(x, n) > 1$, we have many more: p, $2p$, $\ldots$, $(q-1)p$, q, $2q$, $\ldots$, $(p-1)q$, pq. So, for Pollard's rho algorithm, we generate random numbers one by one and check two consecutive numbers. This process is repeated until a factor is found. A function f is used that will generate pseudorandom numbers. In other words, we will keep applying f to generate numbers that seem random for the purpose of this algorithm. One such function that has the pseudorandom property is $f(x) = x^2 + a \bmod n$. We start with $x_1 = 2$ or some other number. We then find $x_2 = f(x_1)$, $x_3 = f(x_2)$, etc., following the general rule $x_{n+1} = f(x_n)$ [Pollard's Rho]. If we use $n = 55$ as an example, Pollard's rho method can be carried out as follows with $f(x) = x^2 + 2 \bmod 55$:

x_n	x_{n+1}	$\gcd(\lvert x_n - x_{n+1}\rvert,\ n)$
2	6	1
6	38	1
38	16	11

The last line in this table tells us that 11 is a factor of 55. From that, we can determine the other factor by calculating $\frac{55}{11} = 5$.

The number $n = 55$ is obviously very small with small p and q, so we can show the Pollard's rho method with a slightly larger n, say $n = 707$ with $f(x) = x^2 + 1 \bmod 707$:

x_n	x_{n+1}	$\gcd(\lvert x_n - x_{n+1}\rvert,\ n)$
2	5	1
5	26	7

The last line in this table tells us that 7 is a factor of 707. From that, we can determine the other factor by calculating $\frac{707}{7} = 101$.

Even though the value of n in our second example of Pollard's rho method was larger than in the first example, the method found the factor just as quickly. This is because Pollard's rho method is very efficient for factoring fairly small numbers or large n values with one factor being significantly smaller than the other. This is in contrast to Fermat's factorization method, which was optimized when the factors p and q were relatively close together. Still, this method is quite inefficient when dealing with the massive n values used in practice with the RSA cryptosystem. One of the problems with Pollard's rho method is that it can generate sequences that cycle and hence never produce a factor. Floyd's cycle finding algorithm, and later Brent's cycle finding algorithm have been integrated into Pollard's rho method to prevent these issues [Pollard's Rho].

As a final note regarding integer factorization, in comparison of the problems of primality testing and integer factorization, factoring a known non-prime integer is in general significantly more time-consuming than finding a prime of approximately the same size. This is really what makes RSA useful in practice. As we have mentioned, the security of RSA is based on the apparent difficulty of factoring a number that is the product of two very large distinct primes. To be more precise, the security of RSA is based on the fact that it would apparently be much more time-consuming for an outsider to factor the publicly known value of $n = pq$ than for the intended recipient of the message to choose p and q [Klima, Sigmon].

7 Modular Exponentiation

Encrypting and decrypting messages using the RSA cryptosystem securely requires modular exponentiation with extremely large bases and exponents. Say we needed to raise the following ciphertext:

$$397056677510513368122841363348174734852 89$$

to the following power:

$$542993009508418269900718536789979854000 35$$

and reduce the result mod the following modulus:

$$2000336999557142833451725215840084689896 39.$$

Even if we used the fastest computer on the planet, completing this calculation by actually multiplying the ciphertext by itself repeatedly with a total number of factors equal to the power would essentially take forever [Klima, Sigmon]. When implementing RSA, there is a necessity to encrypt and decrypt messages efficiently so that the processes can be carried out in a reasonable length of time rather than essentially taking forever. There are much more efficient ways to do these kinds of modular exponentiation calculations that can be done very quickly and allow us to avoid efficiency issues. For example, consider the first decryption calculation $508128^{1445981} \bmod 1524599$ in Example 2 of Section 2. This calculation can be done in a more efficient manner than actually multiplying 508128 by itself repeatedly with a total of 1445981 factors. To do this calculation efficiently, we first find the values of $508128^{2^i} \bmod 1524599$ for $i = 1, 2, \ldots, 20$. So for $P = 508128$ and $M = 1524599$, we begin by computing $P^2, P^4, P^8, ..., P^{2^{20}}$, and reducing each result modulo M. Note that each $P^{2^i} \bmod M$ can be found by squaring $P^{2^{i-1}} \bmod M$, so finding these values requires a total of only 20 multiplications. The modular exponentiation for our example could then be completed by calculating the following:

$$\begin{aligned} P^{1445981} \bmod M &= P^{1048576+262144+131072+4096+64+16+8+4+1} \bmod M \\ &= P^{2^{20}+2^{18}+2^{17}+2^{12}+2^{6}+2^{4}+2^{3}+2^{2}+2^{0}} \bmod M \\ &= P^{2^{20}} P^{2^{18}} P^{2^{17}} P^{2^{12}} P^{2^{6}} P^{2^{4}} P^{2^{3}} P^{2^{2}} P^{2^{0}} \bmod M. \end{aligned}$$

This would only require 8 additional multiplications, so this technique could be used to perform the entire modular exponentiation with a total of only $20 + 8 = 28$ multiplications. This is obviously a vast improvement over the number of multiplications required to find the result by multiplying P by itself repeatedly with a total of 1445981 factors.

The technique for efficiently calculating $P^a \bmod M$ described in the previous paragraph will in general require at most $2\log_2(a)$ multiplications. So even for the massive modular exponentiation described at the beginning of this section, the technique would require at most the following number of multiplications [Klima, Sigmon]:

$$2 \times \log_2(54299300950841826990071853678997985400035) \approx 270.$$

Again, this is obviously a tremendous improvement over multiplying the base by itself repeatedly with a total of the following number of factors:

$$54299300950841826990071853678997985400035.$$

For the Python simulator created for this thesis, a technique for efficient modular exponentiation that is predefined in Python was employed. This technique is almost certainly the technique described in this section.

8 Digital Signatures

RSA being public-key can lead to issues in the verification of the sender of an encrypted message. If we assume that our encryption exponent a and value for n are made public, we typically assume that this information could be accessed by anyone, and not just the intended point of correspondence. If we suppose that there is a group of people who wish to communicate over insecure lines of communication using RSA, and each person has their own secret values of p and q, then each person could make public their a and n. Upon receiving an encrypted message however, the issue comes in determining if the message came from the person claiming to have sent it. This is where the concept of digital signatures comes in.

Suppose we want to send a secret message, P, to a colleague across an insecure line of communication. Assume that our personal modulus n_1 and encryption exponent a_1 are public and our decryption exponent b_1 is kept secret while our colleague has made public their personal modulus n_2 and encryption exponent a_2 while their decryption exponent b_2 is kept secret. Assume also that $n_1 < n_2$. Normally, to encrypt the plaintext P to send to our colleague we would calculate $P^{a_2} \bmod n_2$. To incorporate a digital signature, we could instead apply our own decryption exponent and modulus first by calculating $P_1 = P^{b_1} \bmod n_1$, and then send to our colleague the ciphertext C_1 formed by $C_1 = P_1^{a_2} \bmod n_2$. Our colleague could then easily decrypt this ciphertext by first applying their decryption exponent and modulus to obtain $P_1 = C_1^{b_2} \bmod n_2$, and then using our public encryption exponent and modulus to obtain $P = P_1^{a_1} \bmod n_1$. Since the decryption exponent b_1 used before the encryption is only known to us, our colleague would then know that the message could have only come from us. Because it has the effect of authenticating the message, applying our own decryption exponent and modulus in the encryption of a message is sometimes called signing the message [Klima, Sigmon].

Example: Encryption with a Digital Signature

To send our colleague the message *Meeting time changed to 12!* we would first convert it to its numerical equivalent under the ASCII correspondences, resulting in the following numerical plaintext P:

077101101116105110103032116105109101032099104097110103101100032116111032049050033.

Now, say that we have chosen $p = 12517$ and $q = 154897$, and formed $n_1 = 1938845749$, $a_1 = 19386785$, and $b_1 = 1595863265$. Suppose our colleague has chosen $p = 18013$ and $q = 200003$, and formed $n_2 = 3602654039$, $a_2 = 36024365$, and $b_2 = 3292142165$. Note that $n_1 < n_2$. The first step will be to split P into blocks smaller than n_1, and apply our own decryption exponent b_1 and modulus n_1 to these blocks to form blocks for P_1. Note that since n_2 is 10 digits long, we can group three plaintext characters at a time:

$$\begin{aligned}
077101101^{1595863265} \bmod 1938845749 &= 1562102271 \\
116105110^{1595863265} \bmod 1938845749 &= 1920418988 \\
103032116^{1595863265} \bmod 1938845749 &= 1353617224 \\
105109101^{1595863265} \bmod 1938845749 &= 158917195 \\
032099104^{1595863265} \bmod 1938845749 &= 1315419167 \\
097110103^{1595863265} \bmod 1938845749 &= 529371812 \\
101100032^{1595863265} \bmod 1938845749 &= 1346893900 \\
116111032^{1595863265} \bmod 1938845749 &= 1268387943 \\
049050033^{1595863265} \bmod 1938845749 &= 1402501352.
\end{aligned}$$

Now we can form blocks for the ciphertext C_1 by raising these P_1 blocks to the a_2 mod n_2:

$$\begin{aligned}
1562102271^{36024365} \bmod 3602654039 &= 2536616313 \\
1920418988^{36024365} \bmod 3602654039 &= 1413314080 \\
1353617224^{36024365} \bmod 3602654039 &= 2529883408 \\
158917195^{36024365} \bmod 3602654039 &= 3493306487 \\
1315419167^{36024365} \bmod 3602654039 &= 3175216239 \\
529371812^{36024365} \bmod 3602654039 &= 1443314018 \\
1346893900^{36024365} \bmod 3602654039 &= 1688039043
\end{aligned}$$

$$\begin{aligned}
1268387943^{36024365} \bmod 3602654039 &= 1613024412 \\
1402501352^{36024365} \bmod 3602654039 &= 2381003738.
\end{aligned}$$

Our colleague can start the decryption process by raising these C_1 blocks to the b_2 mod n_2. Note that the results are the P_1 blocks:

$$\begin{aligned}
2536616313^{3292142165} \bmod 3602654039 &= 1562102271 \\
1413314080^{3292142165} \bmod 3602654039 &= 1920418988 \\
2529883408^{3292142165} \bmod 3602654039 &= 1353617224 \\
3493306487^{3292142165} \bmod 3602654039 &= 158917195 \\
3175216239^{3292142165} \bmod 3602654039 &= 1315419167 \\
1443314018^{3292142165} \bmod 3602654039 &= 529371812 \\
1688039043^{3292142165} \bmod 3602654039 &= 1346893900 \\
1613024412^{3292142165} \bmod 3602654039 &= 1268387943 \\
2381003738^{3292142165} \bmod 3602654039 &= 1402501352.
\end{aligned}$$

Our colleague can then complete the decryption process by raising these P_1 blocks to the a_1 mod n_1. Note that the results are the P blocks:

$$\begin{aligned}
1562102271^{19386785} \bmod 1938845749 &= 077101101 \\
1920418988^{19386785} \bmod 1938845749 &= 116105110 \\
1353617224^{19386785} \bmod 1938845749 &= 103032116 \\
158917195^{19386785} \bmod 1938845749 &= 105109101 \\
1315419167^{19386785} \bmod 1938845749 &= 032099104 \\
529371812^{19386785} \bmod 1938845749 &= 097110103 \\
1346893900^{19386785} \bmod 1938845749 &= 101100032 \\
1268387943^{19386785} \bmod 1938845749 &= 116111032 \\
1402501352^{19386785} \bmod 1938845749 &= 049050033.
\end{aligned}$$

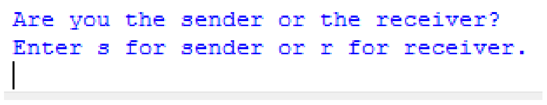
Since, the original plaintext was recovered, our identity was authenticated to our colleague. If the original plaintext was unable to be recovered using our public encryption exponent, our colleague would know that we were not the actual originator of the message.

Recall that we assumed the condition for this digital signature scheme that our modulus n_1 was less than our colleague's modulus n_2. This is because if $n_2 < n_1$, the P_1 blocks could potentially be larger than n_2, and thus not be recoverable in the first decryption step. To avoid this potential problem, if $n_2 < n_1$, we could just reverse the order of the encryption and signing calculations. That is, in encryption, instead of using b_1 and n_1 first and then a_2 and n_2 second, we could use a_2 and n_2 first and then b_1 and n_1 second. This would guarantee that the calculations could be reversed correctly.

9 Python Simulator Description

Some of the features included in the RSA Python simulator created for this project are outlined throughout this thesis. This section synthesizes everything into an overview of what exactly the simulator does and how it simulates the RSA cryptosystem. The core of the program lives in the RSA method. This is where most of the actual RSA simulations are taking place. To supplement this main method, five additional methods are included. These are a toNumber method, a toLetter method, a euclidean method, an isprime method, and a gcd method. The toNumber method takes a string as a parameter and uses Python's predefined "ord" function to convert characters into their ASCII representations. For ASCII representations that are less than 100, a leading 0 is added to make sure all ASCII representations are three digits in length. Similarly, the toLetter method takes in a string of numbers which are then examined three digits at a time and converted to the character representation using Python's predefined "chr" function. The euclidean method is a coded version of the Euclidean algorithm that is used in the RSA process to find the decryption exponent b that satisfies $ab = 1 \bmod m$. In other words the euclidean method finds the multiplicative inverse of a given a when working with a given mod m. Both of these values are parameters to the euclidean method. The next method is the primality test I chose to implement, isprime. As mentioned earlier I used the Fermat primality test and tested about 40 bases. The method will return false if the number given as a parameter is found to be composite, and true otherwise. The gcd method is a method to compute the gcd of two numbers which are given as parameters.

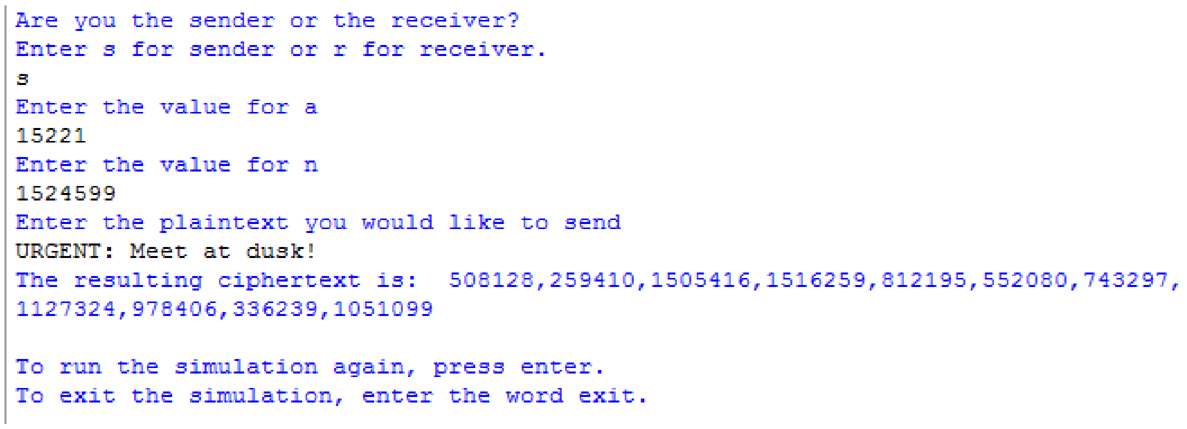
Within the RSA method itself, the simulator begins by asking the user if they are the sender or the receiver in the given scenario.



```
Are you the sender or the receiver?
Enter s for sender or r for receiver.
```

Figure 1: Simulation Prompt

This is essentially asking whether users want to encrypt or decrypt a message. If the user indicates they are the sender, then they will be prompted to enter the public encryption exponent a and the public modulus n of the person they are sending a message to, and the plaintext message they want to send.

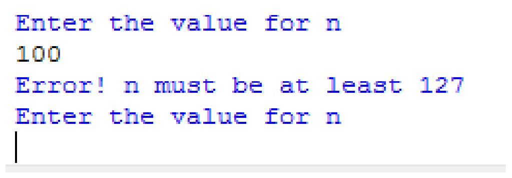


```
Are you the sender or the receiver?
Enter s for sender or r for receiver.
s
Enter the value for a
15221
Enter the value for n
1524599
Enter the plaintext you would like to send
URGENT: Meet at dusk!
The resulting ciphertext is:  508128,259410,1505416,1516259,812195,552080,743297,
1127324,978406,336239,1051099

To run the simulation again, press enter.
To exit the simulation, enter the word exit.
```

Figure 2: Simulation Encryption with Example 2

The simulator also requires $n > 126$ so that the ASCII representations of plaintext characters could be used.

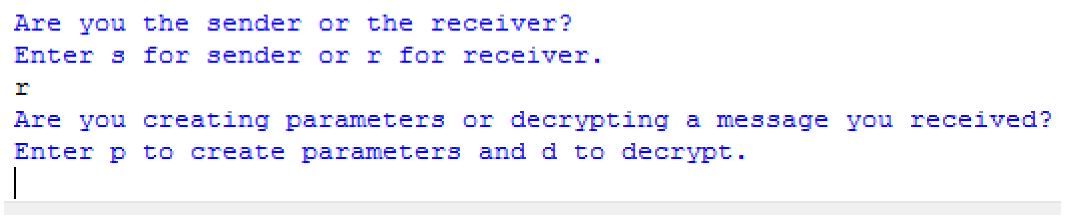


```
Enter the value for n
100
Error! n must be at least 127
Enter the value for n
```

Figure 3: Simulation Requirement for n

The simulator is designed to go through the process of finding the right number of groupings for the plaintext and then encrypting these groupings using the information provided by the user. The resulting ciphertext is then returned to the user instantly.

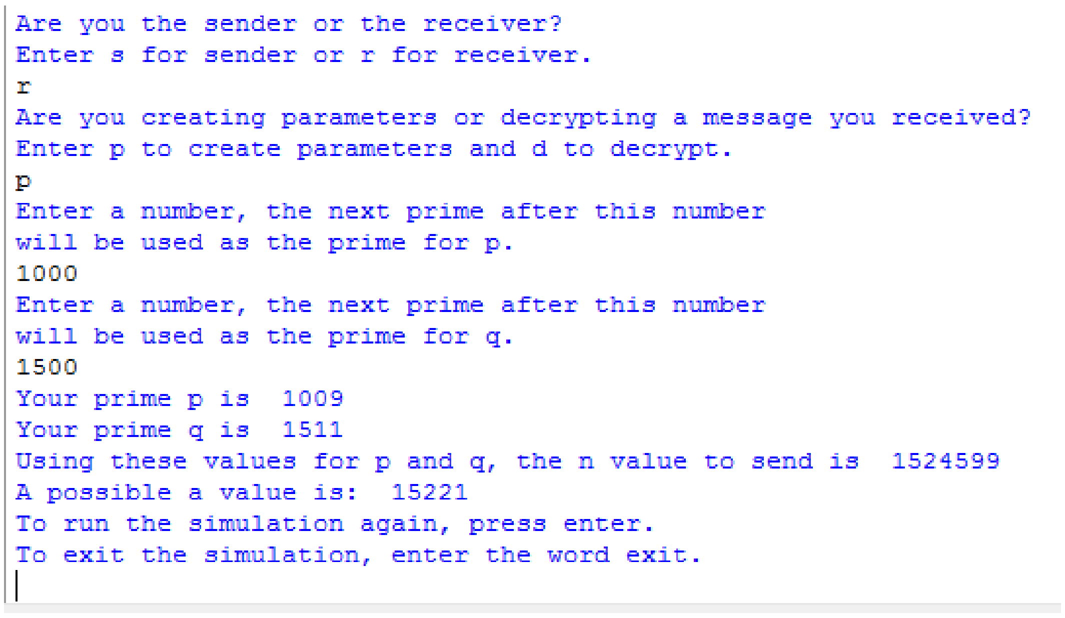
If the user had indicated that they were the receiver, the simulator asks whether they want to decrypt a message or create parameters.



```
Are you the sender or the receiver?
Enter s for sender or r for receiver.
r
Are you creating parameters or decrypting a message you received?
Enter p to create parameters and d to decrypt.
```

Figure 4: Simulation Receiver Options

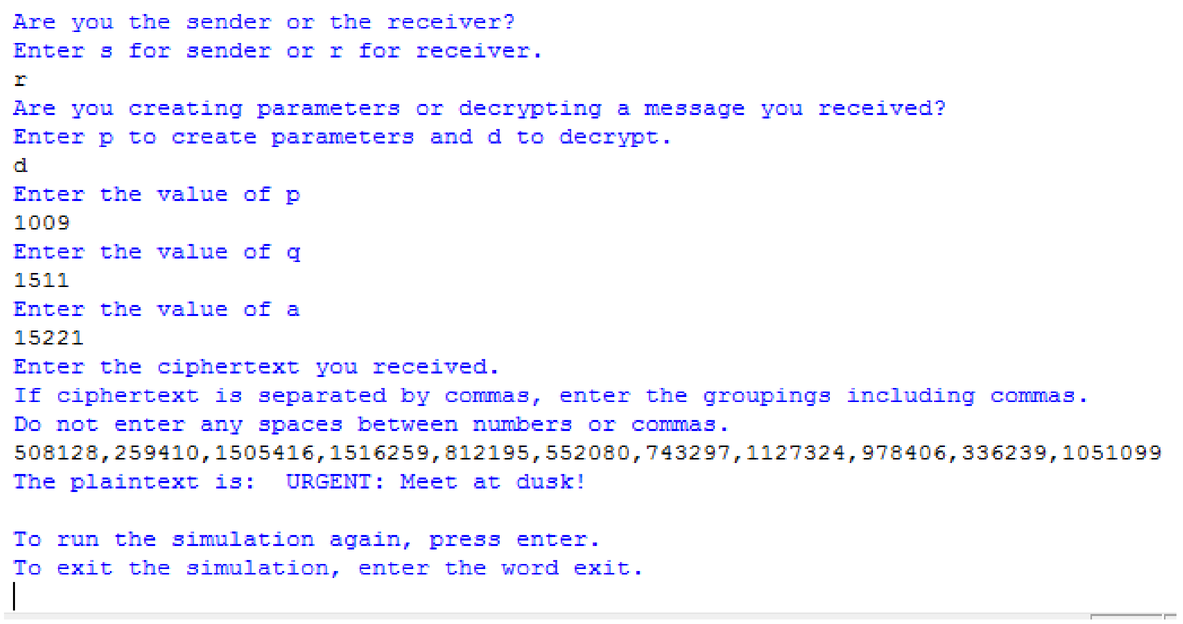
If the user wishes to create parameters, the user is asked for numbers close to what they would like their values of p and q to be. With these inputs, the isprime method is used to find the smallest prime numbers larger than the given inputs. These primes are shown to the user as their p and q. The resulting value of n is also shown as well as a possible value for a to use as their public encryption exponent. This creates all of the parameters needed in order for someone to begin receiving messages.



```
Are you the sender or the receiver?
Enter s for sender or r for receiver.
r
Are you creating parameters or decrypting a message you received?
Enter p to create parameters and d to decrypt.
p
Enter a number, the next prime after this number
will be used as the prime for p.
1000
Enter a number, the next prime after this number
will be used as the prime for q.
1500
Your prime p is  1009
Your prime q is  1511
Using these values for p and q, the n value to send is  1524599
A possible a value is:  15221
To run the simulation again, press enter.
To exit the simulation, enter the word exit.
```

Figure 5: Simulation Parameters

If the user wishes to decrypt a message, the user is prompted to enter their values for p, q, and a. Notice they are not prompted for the decryption exponent because this is calculated within the program using the euclidean method. This makes it so that the user never has to calculate any of these values on their own, a valid a can be given if they create parameters, and they do not have to worry about calculating the value of b in order to decrypt a message. After these prompts, the user can enter the ciphertext. The way encrypted messages are returned by the simulator is in blocks of numbers separated by commas, so this is required of ciphertexts to be decrypted as well. Once all of this information is gathered from the user, the ciphertext is decrypted using the decryption algorithm. The plaintext is then returned to the user.



```
Are you the sender or the receiver?
Enter s for sender or r for receiver.
r
Are you creating parameters or decrypting a message you received?
Enter p to create parameters and d to decrypt.
d
Enter the value of p
1009
Enter the value of q
1511
Enter the value of a
15221
Enter the ciphertext you received.
If ciphertext is separated by commas, enter the groupings including commas.
Do not enter any spaces between numbers or commas.
508128,259410,1505416,1516259,812195,552080,743297,1127324,978406,336239,1051099
The plaintext is:  URGENT: Meet at dusk!

To run the simulation again, press enter.
To exit the simulation, enter the word exit.
```

Figure 6: Simulation Decryption with Example 2

The entire simulator runs on an infinite loop, so as long as the user does not enter the word "exit" after they have completed an interaction, the program will start again from the beginning.

10 Python Simulator Code

```
def RSA():
    end = ""
    while(end != "exit"):
        print "Are you the sender or the receiver?"
        print "Enter s for sender or r for receiver."
        ans = raw_input()

        ctext = ""
        while(ans != "s" and ans != "r"):
            print "Error! Enter s for sender or r for receiver."
            ans = raw_input()
```

```
if(ans == "s"):
    print "Enter the value for a"
    a = int(raw_input())
    print "Enter the value for n"
    n = int(raw_input())
    while(n < 127):
        print "Error! n must be at least 127"
        print "Enter the value for n"
        n = int(raw_input())
    print "Enter the plaintext you would like to send"
    ptext = raw_input()
    ptext = toNumber(ptext)

    #determine length of groupings to encrypt
    if (len(str(n)) % 3 == 0):
        if(len(str(n)) == 3):
            length = 3
        else:
            length = len(str(n)) - 3
    else:
        i = 3
        while(i + 3 < len(str(n))):
            i = i + 3
        length = i

    plength = len(ptext)
    numgroups = plength / length
    leftover = plength % length
```

```
for i in range(0, numgroups * length, length):
    encrypt = ""
    for j in range(i, i + length):
        encrypt += ptext[j]
    encrypt = int(encrypt)
    encrypt = str(pow(encrypt, a, n))
    while(len(encrypt) < length):
        encrypt = "0" + encrypt
    encrypt += ","
    ctext += encrypt

encrypt = ""
if(leftover != 0):
    for i in range(leftover):
        encrypt += ptext[(numgroups*length) + i]

    pad = length - leftover
    for i in range(pad):
        encrypt += "0"

    encrypt = int(encrypt)
    encrypt = str(pow(encrypt, a, n))
    while(len(encrypt) < length):
        encrypt = "0" + encrypt
    ctext += encrypt

print "The resulting ciphertext is: " , ctext
print
```

```
else:

    print "Are you creating parameters or
    decrypting a message you received?"
    print "Enter p to create parameters and d to decrypt."
    choice = raw_input()
    while(choice != "p" and choice != "d"):
        print "Error! Enter p to create parameters or d to decrypt."
        choice = raw_input()

    if(choice == "p"):
        #Help create primes p and q
        print "Enter a number, the next prime after this number"
        print "will be used as the prime for p."
        pp = int(raw_input()) + 1
        while(isprime(pp) != True):
            pp += 1

        print "Enter a number, the next prime after this number"
        print "will be used as the prime for q."
        qq = int(raw_input()) + 1
        while(isprime(qq) != True):
            qq += 1
        print "Your prime p is " , pp
        print "Your prime q is " , qq
        print "Using these values for p and q,
        the n value to send is " , pp*qq

        #Help find an a s.t. gcd(a,m) = 1
        mm = (pp - 1) * (qq - 1)
        aa = max(int(mm * .01), 10)
        while(gcd(aa, mm) != 1 or (aa <= pp or aa <= qq)):
            aa += 1
        print "A possible a value is: ", aa

    if(choice == "d"):

        print "Enter the value of p"
        p = int(raw_input())
        while(isprime(p) != True):
            print "Error! p value is not prime."
            print "Enter another value for p."
            p = int(raw_input())
        print "Enter the value of q"
        q = int(raw_input())
        while(isprime(q) != True):
            print "Error! q value is not prime."
            print "Enter another value for q."
            q = int(raw_input())
        m = (p-1) * (q-1)
        print "Enter the value of a"
        a = int(raw_input())
        while(gcd(a,m) != 1):
            print "Error! a must be relatively prime to n."
            print "Enter a valid a."
            a = int(raw_input())

        print "Enter the ciphertext you received."
```

```
print "If ciphertext is separated by commas,
enter the groupings including commas."
print "Do not enter any spaces between numbers or commas."
c = raw_input()
ctext = c.split(",")
n = p*q
m = (p-1)*(q-1)
ptext = ""
b = euclidean(a, m)
if(b < 0):
    b = b + m

plaintext = ["" for i in range(len(ctext))]
block = 0
for i in range(len(ctext)):
    plaintext[i] = str(pow(int(ctext[i]), b, n))
    if(len(plaintext[i]) > block):
        block = len(plaintext[i])
temp = ""
for i in range(len(ctext)):
    if(len(plaintext[i]) < block):
        temp = ""
        if(len(plaintext[i]) % 3 != 1):
            temp += "0" + plaintext[i]
            if(len(temp) < block):
                j = block - len(temp)
                for k in range(j):
                    temp += "0"
        ptext += temp
```

```
                    else:
                        ptext += plaintext[i]

                #print ptext
                print "The plaintext is: " , toLetter(ptext)
                print ""

        print "To run the simulation again, press enter."
        print "To exit the simulation, enter the word exit."
        end = raw_input()

def toNumber(s):
    num = ""
    hold = 0
    for i in range(len(s)):
        hold = ord(s[i])
        if(len(str(hold)) < 3):
            num += "0" + str(hold)
        else:
            num += str(hold)
    return num

def toLetter(s):
    letter = ""
    for i in range(0, len(s) - 2, 3):
        current = s[i] + s[i+1] + s[i+2]
        letter += chr(int(current))
    return letter
```

```
def euclidean(a, m):
    q = [0,0]
    r = [m, a]
    u = [1, 0]
    v = [0, 1]
    i = 2
    rem = 0
    while(rem != 1):
        rem = m % a
        r.insert(i, rem)
        quo = int(m/a)
        q.insert(i, quo)
        u.insert(i, u[i-2] - u[i-1]*q[i])
        v.insert(i,v[i-2] - v[i-1]*q[i])
        m = a
        a = rem
        i = i+1

    b = v[len(v) - 1]
    return b


def isprime(n):
    a = 2
    for i in range(40):
        while(gcd(a, n) != 1):
            a += 1
        if(pow(a, n-1, n) != 1):
            return False
        a += 1
    return True


def gcd(x,y):
    while(x != 0):
        rem = y % x
        fac = y / x
        y = x
        x = rem
    return y


RSA()
```

References

[Rabin-Miller] “The Rabin-Miller Primality Test.” Web. 19 Mar. 2017.
http://home.sandiego.edu/ dhoffoss/teaching/cryptography/10-Rabin-Miller.pdf

[Lenstra] “Integer Factoring.” Web. 19 Mar. 2017.
https://www.fdi.ucm.es/profesor/m_alonso/Documentos/factorizacion/arjlensfac.pdf

[Klima, Sigmon] Klima, Richard E., and Neil Sigmon. Cryptology: Classical and Modern with Maplets. Boca Raton, FL: CRC Press, 2012. Print.

[Pollard’s Rho] “A Quick Tutorial on Pollard’s Rho Algorithm.” Web. 19 Mar. 2017.
https://www.cs.colorado.edu/ srirams/courses/csci2824-spr14/pollardsRho.html