An Overview of Asymmetric Cryptography

by Craig Belair

Athabasca University

Abstract

Asymmetric cryptography plays an important role in modern digital secu-rity. This paper aims to assess the security and efficiency of some common techniques used in asymmetric cryptography. The paper outlines the RSA pro-tocol and proves that it is valid. Through literature review, I then explore techniques used to optimize RSA implementations. Algorithms for some of these techniques are presented in pseudocode. Mathematical analysis is used in order to gauge the security and expected computational cost of those algo-rithms. Common applications of RSA are described. The relationship between RSA and the classical DHKE is explained. It is determined that many of the techniques used to optimize RSA may also be applied to the classical DHKE. I then compare RSA to newer form of asymmetric cryptography - ECC. I find that in many situations, ECC is able to provide comparable security to RSA while requiring less set up time.

Table of Contents

Motivation for Research

Over the past few decades, the Internet has become increasingly integral to everything we do. Whether we're aware of it or not, our new highly network-dependent world relies heavily on cryptography. Cryptography is a necessary part of any activity that requires the secure transfer of data over the Internet. This includes using email, withdrawing cash from an ATM, making online purchases or sales, using a debit or credit card, accessing an online portal for school or work, streaming online content, using social networking sites/apps, and nearly anything else one might do with their computer or phone. Cryptography can provide users with guarantees of:

Authenticity: Ensuring that the parties communicating with one another are indeed who they claim to be.

Confidentiality: Preventing information communicated from being disclosed to unauthorized parties.

Integrity: Preventing information communicated from being modified by unauthorized parties.

Before comparing asymmetric protocols, it is necessary to develop a foundational understanding of how asymmetric cryptography works. I begin by exploring the difference between asymmetric and symmetric methods of cryptography. I then present a criteria that must be fulfilled by the functions used when developing a public key cryptosystem. I then analyze the efficacy, cost, and uses of one of the most common asymmetric cryptographic protocols: Rivest-Shamir-Adleman (RSA). The paper also touches on the Diffie-Hellman Key Exchange (DHKE), and the use of Elliptic-Curve Cryptography (ECC).

Asymmetric vs Symmetric Methods

There are two main methods for designing cryptographic schemes - symmetric methods and asymmetric methods. When using a symmetric method, all parties communicating share the same secret key. This shared key is used to encrypt and decrypt messages. The word "symmetric" is used because the key used to encrypt messages is the same as the key used to decrypt messages. All cryptography from ancient times until 1976 was exclusively based on symmetric methods.[1] When asymmetric methods are used for encryption and decryption, two different keys are used: a private key and a public key. Anyone with the public key can encrypt data, but only the private key can be used to decrypt data that has been encrypted using the public key. When asymmetric methods are used to develop a system for encryption and decryption, that system is referred to as a public key cryptosystem.

Although modern symmetric algorithms are fast and secure, they do have a major short-coming. In order for a symmetric protocol to work, the secret key must first be exchanged between the parties that intend to communicate. To ensure the security of the symmetric scheme, this secret key must be communicated over a secure channel. This presents a serious logistics problem - the key must be shared in order to encrypt data, but the key itself must be encrypted in order for it to be shared. The problem of securely communicating a secret key is referred to as the key distribution problem..

Asymmetric methods can be used to solve the key distribution problem. A secret key (to be used for a symmetric method) can be encrypted and shared using asymmetric methods. The exact details of this key exchange depend on the implementation of the cryptosystem(s) used in the key exchange. I will provide examples of key distribution techniques throughout my paper. Another important use of asymmetric methods is the creation of digital signatures. Digital signatures provide a way to assure that the author of a message is authentic, and that the message has not been compromised. But if asymmetric methods can already be used to encrypt data and more, why do we bother using asymmetric methods for key exchange? Why not simply encrypt our messages using the public key cryptosystems themselves?

Unfortunately asymmetric methods are based on computationally heavy mathematical functions. As such, the actual execution of asymmetric algorithms tends to be much slower than the execution of their symmetric counterparts (slower by a factor of about $1,000$ according to Kumar et al (2011)).[3] Because of this short-coming, asymmetric methods are rarely used for the encryption of large amounts of data. Instead, asymmetric methods are typically integrated with symmetric methods - asymmetric methods solve the key distribution problem and provide other important security measures, while symmetric methods are used to encrypt any bulk data that must be communicated.[14]

Criteria for Public Key Cryptosystems

The importance of asymmetric methods is undeniable, but a way of designing such methods is not immediately obvious. Consider the problem of setting up a public key system. Those interested in creating such a system need a process for generating two functions, a decryption function, $D(m)$, and an encryption function $E(m)$. Their process must satisfy the following criteria:

Criterion 1) $D(E(m)) = m$ for any valid input message m.

If it were not the case that $D(E(m)) = m$, then the decryption function would not reverse the encryption function, and would therefore not reliably return the correct message.

Criterion 2) Both $D(m)$ and $E(m)$ must be computationally easy.

This criterion ensures that the cryptosystem is practical. If the process of encryption or decryption required too much time (or too many resources), then the system would not be useful.

Criterion 3) It must be computationally unfeasible to deduce $D(m)$ using $E(m)$.

It is necessary that $D(m)$ be kept secret from individuals knowing $E(m)$. If this were not the case, then anyone holding the public key could decrypt data.

Criterion 4) It must be computationally easy to generate many function pairs, $E(m)$ and $D(m)$

Each time the public key system is initialized, a new pair of public/private keys (hence a new pair of functions, $E(m)$ and $D(m)$) should be generated. If the same keys were used each session, then the system would not be secure. It follows that the process of generating a new function pair, $E(m)$ and $D(m)$, must be relatively computationally easy. The maximum acceptable cost of initializing the public key system tends to be much larger than the maximum acceptable cost of encrypting and decrypting messages. This is because the system must be initialized only once per each session.

The phrase *computationally easy* is somewhat ambiguous. The acceptable cost of computing $E(m)$ and $D(m)$ depends on the intended use of the cryptosystem, as well as the technology available to execute the functions. By nearly any standard, if functions are to be considered computationally easy, the functions must run in at most polynomial time.[15] That is, an upper cap on the runtime of $E(m)$ and $D(m)$ must be expressible as a polynomial function of the bit length of the input, m. Using asymptotic complexity notation, one can say that if a function, $F(m)$, is to be considered computationally easy, then it must be the case that $F(m) = O(p(m))$ where $p(m)$ is some polynomial function. Note that although $F(m) = O(p(m))$ is a necessary condition for $F(m)$ to be considered computationally easy, it is not a sufficient condition.

Similarly, the phrase *computationally unfeasible* is ill-defined. The more computationally difficult it is to deduce $D(m)$ using $E(m)$, the more secure the cryptosystem is. However, the level of computational difficulty that is needed

depends on both how secure the system is expected to be, and the technology available to attackers. Typically, in order for a problem to be considered computationally unfeasible, it must be estimated that with the best modern technology and the best known algorithms for solving the problem, the problem would still take several thousand years to solve.[15] Often times, the computational difficulty of a problem can be deceptive - advances in algorithms and hardware can quickly transform seemingly impossible problems into manageable ones. For example, in 1977, Rivest estimated that factoring a 125 digit number which is the product of two 63 digit primes would require at least 40 quadrillion years using the best factoring algorithm known.[38] However, by 1994, such numbers had already been factored. By 2005, an RSA modulus of over 200 digits had been factored. A chart of the records for the largest RSA modulus factored between the years 1964 and 2005 is contained in *Appendix III* (Figure 1).

An Introduction to RSA

RSA is the most widely used asymmetric encryption algorithm to date.[16] It was publicized in 1977 and it is named after its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman. The security of RSA relies on the difficulty of factoring large numbers.

The basic idea behind RSA is quite simple. During the initialization phase the public and private keys are generated. When the initialization phase begins, two distinct primes, p and q must be found. Let $n = pq$. The encryption power e is then chosen*: e must be a positive integer such that $gcf(e, (p-1)(q-1)) = 1$.**

*(Note that in this case, e does **not** refer to the base for the natural logarithm.)
**(Throughout this paper, the notation $gcf(x, y)$ denotes the greatest common factor of x and y.)

Using the *Extended Euclidean Algorithm*[6], one can then find $d \in \mathbb{Z}^+$ such that $ed = 1 + k(p-1)(q-1)$ for some $k \in \mathbb{Z}^+$. The integer d is used as the decryption power. The public key is then broadcast as the pair of values, (e, n). While the private key, d, is kept secret.

Once the initialization phase is complete, users are ready to encrypt plaintext messages m. In order for m to be encrypted, m must be an integer with $0 \leq m < n$. If the message m is not already a non-negative integer, some encoding technique can transform m into a non-negative integer. If $m \geq n$, then m can be broken up into several smaller blocks, and each block can be encrypted and sent individually. With an appropriate input m, the RSA en-

cryption function is defined as follows:

$E(m) = m^e \ \% \ (n)$

Throughout this paper, the notation $x \ \% \ (n)$ denotes reduction of x mod n. To be precise:

$x' = x \ \% \ (n)$ if and only if x' is the unique integer such that $x' \equiv x \bmod n$ **and** $x' \in \{0, 1, 2, ..., n-1\}$. A proof of the uniqueness of x' is given in *Appendix I* (Theorem A.) The set $\{0, 1, 2, ..., n-1\}$ is known as the canonical complete residue system modulo n.

The RSA decryption function is as follows:

$D(c) = c^d \ \% \ (n)$

Proof of the Correctness of RSA

Suppose RSA parameters were initialized in the manner described above. I will prove that $D(E(m)) = m$ for any valid input m. Throughout this paper I will take for granted certain properties of modular arithmetic. *Appendix I* presents some auxiliary proofs that support these assumptions.

I will first show that $m^{ed} \equiv m \bmod n$. I will then use this result to show that $D(E(m)) = m$.

Since n is the product of two primes, p and q, its only factors are 1, p, q, and pq. Further, since $m < n = pq$, it must be the case that $gcf(m, n) \neq pq$. The only remaining possibilities are: $gcf(m, n) = 1$, $gcf(m, n) = p$, or $gcf(m, n) = q$. Partition the possible choices for m into two cases:

Case A : Suppose $gcf(m, n) = 1$. Then:

$m^{ed} = m^{k(p-1)(q-1)+1} = (m^{(p-1)(q-1)})^k * m = (m^{\phi(n)})^k * m$

Where ϕ represents the Euler Totient Function. *Euler's Theorem*[9] shows that for any m such that $gcf(m, n) = 1$, it is the case that $m^{\phi(n)} \equiv 1 \bmod n$. Thus:

$(m^{\phi(n)})^k * m \equiv (1)^k * m \equiv m \bmod n$

Hence, if $gcf(m, n) = 1$, then $m^{ed} \equiv m \bmod n$.

Case B : Now consider cases where $gcf(m, n)$ is either p or q. Without loss of generality, one can suppose $gcf(m, n) = p$. Then:

$m^{ed} = m^{k(p-1)(q-1)+1} = (m^{q-1})^{k(p-1)} * m = (m^{\phi(q)})^{k(p-1)} * m$

Where ϕ again represents the *Euler Totient Function*. Using *Euler's Theorem*, this reduces to:

$(m^{\phi(q)})^{k(p-1)} * m \equiv (1)^{k(p-1)} * m \equiv m \bmod q$

Since $gcf(m,n) = p$, p divides m. Hence:

$m^{ed} \equiv 0^{ed} \equiv 0 \equiv m \bmod p$

Since $m^{ed} \equiv m \bmod p$, $m^{ed} \equiv m \bmod q$, and $gcf(p,q) = 1$, by *Theorem C* in *Appendix I*, it must be the case that $m^{ed} \equiv m \bmod pq$. Thus if $gcf(m,n) = p$ then $m^{ed} \equiv m \bmod n$.

An identical argument can be made to show that when $gcf(m,n) = q$ it is the case that $m^{ed} \equiv m \bmod n$. Thus for any valid selection of m, it must be the case that $m^{ed} \equiv m \bmod n$.

One more detail must be addressed to prove that $D(E(m)) = m$:

Let $D(E(m)) = m'$. Note that by construction of the decryption function (which involves reduction modulo n), m' is contained in the canonical complete residue system modulo n. I have already shown that $m' \equiv m \bmod n$. Hence, by *Theorem A* in *Appendix I*, it must be the case that $m' = m$. (m is the unique representative of m in the canonical complete residue system modulo n.)

Q.E.D.

Naive Modular Exponentiation

I have shown the correctness of the RSA decryption and encryption functions, but it must also be shown that these functions are not too costly. The encryption and decryption functions in RSA are very simple - all that must be done is exponentiation and reduction modulo n. So how does one perform this modular exponentiation?

Suppose you want to take m and raise it to the e'th power (where e is the encryption power in the RSA system). A naive approach would be to start with m, and then multiply m by itself $e-1$ times. Afterall - exponentiation is just repeated multiplication. Modern RSA exponents are typically between 2048 and 3072 bit numbers.[4] Straightforward multiplication would therefore take up-

wards of about 2^{2048} multiplications. Suppose a computer can multiply by m and perform a reduction modulo n in a microsecond. It would then require about 10^{600} millennia to encrypt our data. Unfortunately, most users are unwilling to wait even one millennium! Fortunately, mathematical techniques can be used to expedite the exponentiation. In the original paper in which RSA was published, the Repeated Square and Multiply Technique was recommended to speedup exponentiation.[5] I will describe this method and analyze the expected cost of exponentiation using this method.

The Repeated Square and Multiply Technique:

The objective of the *Repeated Square and Multiply Technique* is to take an integer m, and compute $m^e \ \% \ (n)$. For the sake of simplicity, I will not explicitly address the reduction modulo n in my algorithm. In nearly any real-world implementation, reduction modulo n takes place after each squaring or multiplication.

Suppose e is an k bit integer. The description of this method will make use of the binary representation of the integer e. Binary values will be denoted by parentheses and a subscripted 2. Let $e = (a_1 a_2 a_3 ... a_k)_2$ where $a_1 = 1$ and $a_i \in \{0, 1\}$ for $2 \leq i \leq k$. Then the algorithm proceeds as follows:

Repeated Square and Multiply Algorithm

Step 1:
Initialize $m_1 = m$. Move to the next step.

Step i (for $1 < i \leq k$):
If $a_i = 0$ let $m_i = (m_{i-1})^2$.

If instead $a_i = 1$, let $m_i = (m_{i-1})^2 * m$. Move to the next step.

Step $k + 1$:
Return m_k as the output. Terminate the algorithm.

Correctness of the Repeated Square and Multiply Technique

Recall that the exponent e can be represented in binary as $e = (a_1 a_2 ... a_k)_2$. I will make the inductive assumption that $m_i = m^{(a_1 a_2 ... a_i)_2}$ for any $1 \leq i \leq k$. Below the validity of this induction is proven:

BASIS STEP ($i = 1$):

The algorithm starts by setting $m_1 = m$. Hence:

$m_1 = m = m^{(1)_2} = m^{(a_1)_2}$

Thus for $i = 1$ it is the case that $m_i = m^{(a_i)_2}$.

INDUCTION ($1 < i \leq k$) (part A):

Suppose $a_i = 0$. By the inductive assumption, $m_{i-1} = m^{(a_1 a_2 \ldots a_{i-1})_2}$. Since $a_i = 0$, the i'th step computes $m_i = (m_{i-1})^2$. Using the inductive assumption and binary notation, it follows that:

$m_i = (m_{i-1})^{(10)_2} = (m^{(a_1 a_2 \ldots a_{i-1})_2})^{(10)_2} = m^{(a_1 a_2 \ldots a_{i-1})_2 * (10)_2} = m^{(a_1 a_2 \ldots a_{i-1} 0)_2} = m^{(a_1 a_2 \ldots a_{i-1} a_i)_2}$

Thus if $a_i = 0$, the inductive assumption holds.

INDUCTION ($1 < i \leq k$) (part B):

Suppose instead that $a_i = 1$. Then on the i'th step, the algorithm begins by squaring m_{i-1}. As shown above, $(m_{i-1})^2 = m^{(a_1 a_2 \ldots a_{i-1} 0)_2}$. The algorithm then multiplies the result by m, yielding:

$m_i = (m_{i-1})^2 * m = m^{(a_1 a_2 \ldots a_{i-1} 0)_2} * m^{(1)_2} = m^{(a_1 a_2 \ldots a_{i-1} 1)_2} = m^{(a_1 a_2 \ldots a_{i-1} a_i)_2}$

Again the inductive assumption holds. This shows that for all $1 \leq i \leq k$ it is the case that $m_i = m^{(a_1 a_2 \ldots a_i)_2}$. It follows that $m_k = m^{(a_1 a_2 a_3 \ldots a_k)_2} = m^e$. Since $m_k = m^e$, the algorithm must provide the desired output.

Q.E.D.

The Cost of the Repeated Square and Multiply Technique

How does the cost of the *Repeated Square and Multiply Technique* compare to the naive approach? If e is a k bit integer, the algorithm must perform $k - 1$ squarings, as each time it "add a bit" it must square the current stored value. The algorithm must also perform $H(e) - 1$ multiplications by m, where $H(e)$ is the number of 1's in the binary representation of e. ($H(e)$ is known as the Hamming weight of e.) The number of bits needed to represent e is $\lfloor log_2(e) \rfloor + 1$. If you assume that the digits of the bits of e are evenly distributed, then $H(e) \cong \frac{\lfloor log_2(e) \rfloor + 1}{2}$.

(Here and throughout this paper, $\cong$ is used to denote approximately equal to.)

Thus the number of squarings required in computing m^e is $\lfloor log_2(e) \rfloor$. The expected number of multiplications by m to compute m^e is about $\frac{\lfloor log_2(e) \rfloor + 1}{2} - 1$. Altogether, this is about $\frac{3}{2}\lfloor log_2(e) \rfloor$ operations of either squaring or multiplication by m. Squaring and multiplication by m are operations of similar cost, as in both cases, the algorithm is taking the product of two integers, both of which are less than n. (They are less than n because reduction modulo n occurs after each squaring or multiplication.)

For the 2048 bit keys mentioned earlier, this results in a total of about $\frac{3}{2}\lfloor log_2(2^{2048}) \rfloor = 3072$ multiplication/squaring operations. For each operation, a reduction modulo n must also occur. Assuming that a modern computer can perform a multiplication and reduction modulo n in a microsecond, then computing m^e using the *Repeated Square and Multiply Technique* would take approximately 3 milliseconds. Much more acceptable than the many, many millennia required when using the naive approach. The *Repeated Square and Multiply Technique* does not require a significant amount of memory. In the algorithm presented above, only m_i, for the current step i, must be stored. Once m_{i+1} has been determined, m_i can be discarded.

Since the publication of RSA in 1977, many other techniques for speeding up exponentiation have been developed.

Other Methods for Fast Exponentiation

The *Repeated Square and Multiply Technique* can be extended into a family of techniques for modular exponentiation known as M-ary Techniques. I outline *M-ary Techniques* below.

Suppose one wishes to compute $x^e \ \% \ (n)$. A power of two, M, is first chosen. The exponent e is then represented in base M: $e = (a_1 a_2 ... a_k)_M$. The set S (shown below) is precomputed and stored for future reference:

$S = \{x \ \% \ (n), \ x^2 \ \% \ (n), \ x^3 \ \% \ (n), \ ... \ x^{M-1} \ \% \ (n) \ \}$

The algorithm then continues in a manner similar to the *Repeated Square*

and Multiply Technique. For step 1, it initializes x_1 as x^{a_1}. For steps $1 < i \leq k$, it lets $x_i = (x_{i-1})^M * x^{a_i}$. Multiplication by x^{a_i} can be quickly performed by looking up the value for x^{a_i} in the stored set S.

A detailed outline of the *M-ary Techniques* is available in the paper *Efficient Modular Exponentiation Methods for RSA*(Güner et al (2017)).[17] There, authors also provided a detailed description of the *Modified M-ary Technique*. When using the *Modified M-ary Technique*, during the precomputation phase, only the odd powers of x are precomputed to form the set S. When an even power of x is needed, the algorithm instead squares the appropriate stored odd power of x. *Güner*'s paper presented a theoretical comparison of the *M-ary technique*, the *Modified M-ary Technique*, and the *Repeated Square and Multiply Technique*. The authors also implemented algorithms using each of these techniques, and compared the run-times of those implementations using a range of different RSA modulus sizes. In their implementations, for all moduli, the encryption exponent, e, was fixed at 65537. The decryption power, d, of course was dependent on e and the modulus. In all cases, the decryption power d was larger than the encryption power, e. As will be shown in the next section of this paper, the RSA decryption exponent, d, is almost always much larger than the encryption exponent, e.

The *M-ary Techniques* did not appear to provide significant improvements over the *Repeated Square and Multiply Technique* during encryption. However, during decryption (where larger exponents were used), an improvement of over 23% was found for each modulus size. In order to achieve the improvement, values for M had to be strategically chosen. In their implementations, the *Modified M-ary Technique* performed slightly better than the *M-ary Technique*. Both the *M-ary Techniques* and the *Modified M-ary Techniques* require a small amount of memory usage. This memory is used to store the precomputed sets. A table of values comparing the speed of the *Modified M-ary Technique* to the *Repeated Square and Multiply Technique* was provided in their paper. For convenience, the table is also present in *Appendix III* (Figure 2).

Another family of techniques for fast modular exponentiation are the Sliding Window Techniques. Suppose again one has an integer x, and wishes to compute $x^e \ \% \ (n)$. The algorithm represents e using base 2, $e = (a_1 a_2 ... a_k)_2$. The k bits of e are then partitioned into smaller windows. The exact manner in which these k bits are partitioned depend on the particulars of the *Sliding Window Technique* used. Let $|$ denote concatenation. Then $e = (a_1 a_2 ... a_k)_2 = (w_1|w_2|...w_i)_2$ where each w_i is a string of bits. Let z_i represent the number of bits in e which are to the right of the rightmost bit of the window w_i. The algorithm can then compute $x^e = \Pi_{t=1}^{t=i}(x^{w_t})^{2^{z_t}}$. For further clarity, I present an example below.

Suppose $e_p = (1100100)_2$. One possible way to partition e_p is: $e_p = (11|00|1|00)_2$.

In this case $w_1 = (11)_2$, $w_2 = (00)_2$, $w_3 = (1)_2$, and $w_4 = (00)_2$. While $z_1 = 5$, $z_2 = 3$, $z_3 = 2$, and $z_4 = 0$.

One can then compute:

$(x^{w_1})^{2^{z_1}} * (x^{w_2})^{2^{z_2}} * (x^{w_3})^{2^{z_3}} * (x^{w_4})^{2^{z_4}} =$

$(x^{(11)_2})^{(100000)_2} * (x^{(00)_2})^{(1000)_2} * (x^{(1)_2})^{(100)_2} * (x^{(00)_2})^{(1)_2} =$

$x^{(1100000)_2} * x^{(0)_2} * x^{(100)_2} * x^{(0)_2} = x^{(1100100)_2}$

One benefit of the *Sliding Window Techniques* can be seen through this example. Any windows which contains only the digit 0 need not be computed. This is true because for any $x \in \mathbb{Z}^+$ it is the case that $(x^0)^{2^{z_i}} = 1$.

When using *Sliding Window* methods, precomputation can be used to store x^i for all $2 \leq i < 2^t$ where t is the bitlength of the largest window. This reduces the problem of computing x^{w_i} to looking up a value in the precomputed set. However, for large windows this requires a significant amount of precomputation. Storing larger precomputed sets also requires the use of more memory.

In some *Sliding Window Techniques*, variable window lengths (which were used in my example) are not permitted. The paper *Efficient Pre-Processing for Large Window-Based Modular Exponentiation Using Ant Colony* (Nedjah 2005) gives a detailed description of different types of *Sliding Window* methods.[18] In the paper, author Nedjah notes that *Sliding Window Techniques* can be further optimized by strategically selecting what values to precompute. She developed an algorithm that uses an artificial intelligence system to minimize the number of operations required during precomputation. When implemented, she found that her system (the *ant system*) performed favorably to *M-ary Techniques*.

Efficient modular exponentiation remains a very active research topic. As such, my discussion of the topic has certainly not been exhaustive - I have presented only the foundational ideas behind efficient modular exponentiation. Another way of improving the performance of RSA implementations is to strategically select the keys used to initialize the RSA system.

Key Selection for Fast Encryption in RSA

During the initialization phase of RSA, there is some freedom in selecting the public key (e, n). Since 2015, NIST (the National Institute of Standards and Technology) has recommended a minimum bit length of 2048 for the modulus n when implementing RSA.[7] The decryption key, d must also be reasonably large - if d were too small, then the system would be vulnerable to brute force attacks. In order for the system to be secure, d should have a bit length of at least $0.3t$ where t is the bit length of the modulus n.[22] In practice, the bit length of d is often very close to the bit length of n. However, the public exponent, e, need not be particularly large. Needless to say, by choosing small values for e, one can expedite the process of computing m^e. The most commonly chosen value for e is $2^{16} + 1$; a key of only 17 bits.[21]

Due to the difference between the size of e and the size of d, there is a difference between the speed of encryption and the speed of decryption when using RSA. RSA encryption tends to be significantly faster than RSA decryption. In fact, RSA encryption is, in almost all practical cases, faster than encryption using any other public key scheme.[20] In addition to selecting small values for e, values for e can be selected such that their Hamming weight, $H(e)$, is low. This is why values of the form $e_p = 2^k + 1$ are commonly used. For such values, $H(e_p) = 2$. Recall from my analysis that $H(e) - 1$ determines the number of multiplications required when computing m^e when using the *Repeated Square and Multiply Technique.*

The Chinese Remainder Theorem for Fast Decryption in RSA

Selection of the decryption key, d, does not allow for the same freedoms as does the selection of the encryption key e. d must be kept secret, and thus cannot be too small. d also cannot be selected based on a predictable pattern. Further, d depends on the selection of e and n, since it must be the case that $ed \equiv 1 \bmod (p-1)(q-1)$.* However, the primes p and q that are used to create n do not need to be kept secret from those who hold the private key d. As such, the values p and q can be used to help speedup the process of decryption. Suppose an RSA system is initialized with valid parameters, and that one wishes to compute $x^d \ \% \ (n)$. Further, suppose that the values p and q that were used to construct the system are known. I will describe an alternative method for computing $x^d \ \% \ (n)$ which makes use of the Chinese Remainder Theorem . I will then show that this alternative method is typically less costly than the conventional approach to computing $x^d \ \% \ (n)$.

*(It is sufficient that $ed \equiv 1 \bmod lcm((p-1),(q-1))$ where $lcm(x, y)$ represents the lowest common multiple of x and y. Some outlines of RSA however

specify that $ed \equiv 1 \bmod (p-1)(q-1)$; I have used this as the convention in my paper.)

I begin by defining some variables:

$x_p = x \ \% \ (p)$
$x_q = x \ \% \ (q)$
$d_p = d \ \% \ (p-1)$
$d_q = d \ \% \ (q-1)$
$y_p = x_p^{d_p} \ \% \ (p)$
$y_q = x_q^{d_q} \ \% \ (q)$
c_p is the unique value such that $qc_p \equiv 1 \bmod p$ and $c_p \in \{1, 2, ..., p-1\}$
c_q is the unique value such that $pc_q \equiv 1 \bmod q$ and $c_q \in \{1, 2, ..., q-1\}$

The values c_p and c_q can be computed using the *Extended Euclidean Algorithm*. They are guaranteed to exist because $gcf(p, q) = 1$. Further, they are guaranteed uniqueness by *Theorem A* in *Appendix I*.

Then let $x_{pq} = [(qc_p)y_p + (pc_q)y_q] \ \% \ (n)$.

I aim to prove that $x_{pq} = x^d \ \% \ (n)$.

Proof of the Correctness of Chinese Remainder Theorem Exponentiation:

I will first show that $x_{pq} \equiv x^d \bmod p$. Using *Euler's Theorem*:

$x_{pq} \equiv (qc_p)y_p + (pc_q)y_q \equiv qc_py_p + (0 * c_q)y_q \equiv (qc_p)y_p \equiv$

$1y_p \equiv x_p^{d_p} \equiv x^{d_p} \equiv x^{d+k(p-1)} \equiv x^d * (x^{p-1})^k \equiv x^d(1)^k \equiv x^d \bmod p$

Thus $x_{pq} \equiv x^d \bmod p$

A symmetric argument shows that $x_{pq} \equiv x^d \bmod q$

By *Theorem C* (in *Appendix I*), since $gcf(p, q) = 1$, it follows that $x_{pq} \equiv x^d \bmod pq$. Since x_{pq} is a member of the canonical complete residue system modulo n and $x_{pq} \equiv x^d \bmod n$, it must be the case that $x_{pq} = x^d \ \% \ (n)$. Thus x_{pq} is the same as the value found using the conventional approach to computing $x^d \ \% \ (n)$.

Q.E.D.

Making use of the *Chinese Remainder Theorem* (CRT) gives the correct result for $x^d \ \% \ (n)$, but is it any faster than conventional modular exponentiation?

I will make some approximations in order to compare the cost of decryption using the CRT technique to the cost of decryption without using the CRT technique.

In all RSA implementations conforming to the first version of the Public-Key Cryptography Standards (PKCS #1), d is chosen such that $d \leq n$. Typically, d is of similar bit-length to n. The reason this is true can be seen by analyzing the way RSA is implemented. By construction, $ed = 1+k(p-1)(q-1)$. Often $k = 1$; key selections such that $k = 1$ allow for faster computation than comparable selections where $k > 1$. If one assumes $k = 1$, then it follows that:

$$ed = 1 + (p-1)(q-1) \cong 1 + pq = 1 + n \cong n$$

Since $ed \cong n$, it follows that:

$$log_2(e) + log_2(d) = log_2(ed) \cong log_2(n)$$

As noted in the previous section, e is typically chosen to be of small bit-length. Hence $log_2(e)$ is negligible and:

$$log_2(d) \cong log_2(n)$$

Using the *Repeated Square and Multiply Technique*, my analysis found that the number of squaring/multiplication operations that must be performed in computing x^d for some exponent d was approximately $\frac{3}{2}log_2(d)$. So if one computes x^d without the use of the CRT technique, the expected number of squaring/multiplications is about:

$$\tfrac{3}{2}log_2(d) \cong \tfrac{3}{2}log_2(n)$$

How does this compare to computing x^d while using the CRT technique? The steps required to calculate x_p, x_q, d_p and d_q are relatively trivial - they each require a singular modular reduction. Computing c_q and c_p does not take a significant amount of time either - each requires a single application of the *Extended Euclidean Algorithm*. The bulk of the computation that must be done using the CRT technique takes place when computing y_p and y_q:

$$y_p = x_p^{d_p} \ \% \ (p)$$

$$y_q = x_q^{d_q} \ \% \ (q)$$

Note that $d_p < p$ and $d_q < q$ by construction. Here I again use the approximate expression determined during the analysis of the *Repeated Square and Multiply*

Technique. The number of multiplications/squarings required to compute y_p is approximately $\frac{3}{2}log_2(d_p)$ while the number of squarings/multiplications used to compute y_q is approximately $\frac{3}{2}log_2(d_q)$. Thus to compute both y_q and y_p, the expected number of operations is:

$\frac{3}{2}log_2(d_q) + \frac{3}{2}log_2(d_p) = \frac{3}{2}log_2(d_q d_p)$

Note that

$\frac{3}{2}log_2(d_q d_p) < \frac{3}{2}log_2(pq) = \frac{3}{2}log_2(n)$

Thus one can expect to perform no more than about $\frac{3}{2}log_2(n)$ operations of squaring or multiplication when using the CRT technique. $\frac{3}{2}log_2(n)$ is the same number of operations I approximated would be needed without using the CRT technique. However, note that x_p and d_p are bound above by p, and x_q and d_q are bound above by q. As such, although the number of multiplication/squaring operations are similar when using either technique, the sizes of the integers being dealt with are smaller when using the CRT technique. How much of a benefit does one expect to gain from dealing with smaller integers?

When p and q are chosen during initialization, they are typically chosen to be of similar size. One reason for this choice is that when p and q are of similar size, the CRT exponentiation method provides the greatest speed benefit. Another reason for the choice is that it avoids having an unnecessarily small factor of n. When n has a small factor, n is vulnerable to certain factorization techniques (such as the Elliptic Curve Factorization Method).[23] If p and q are chosen to be of similar size, it is reasonable to assume that if n is a k bit integer, then p and q are each approximately $\frac{k}{2}$ bit integers. So by using the CRT technique, one bounds the size of integers they're working with to be about half the bit length of the integers they would be working with if they did not use the CRT technique. The complexity of multiplication decreases quadratically with bit length.[24] It follows that operating with integers bound by $\frac{k}{2}$ bits rather than integers bound by k bits results in decryption using the CRT technique being up to 4 times as fast as decryption without the use of the CRT technique.

This concludes my analysis of encryption and decryption using RSA. I will now further explore how RSA parameters can be initialized.

RSA Initialization

Recall that when initializing an RSA protocol, the protocol must find two

distinct primes, p and q. If p and q are chosen to be of similar size, then for a k bit modulus, the protocol must find two primes, each of which is approximately $\frac{k}{2}$ bits in length. One method for finding these $\frac{k}{2}$ bit primes (as outlined in the *Handbook of Applied Cryptography*[25]) involves selecting random odd $\frac{k}{2}$ bit integers, and then performing primality tests on those integers. Any integers that are determined to be composite are discarded. Integers that are determined to be very likely prime are used in the initialization. The process of generating random integers itself is a surprisingly nuanced problem.

Computer functions are by definition deterministic - so when a standard computer function is used to generate "random" numbers, that function does so in a predictable pattern. When it comes to cryptographic keys, it is important that the randomly generated numbers not be predictable. Predictably generated keys are not secure. To address the problem of generating unpredictable random integers, cryptographers often draw from hard to predict real-world phenomena. Data from chaotic real-world systems can be tracked and translated into seeds that are then used to generate strings of random integers. For example, the movement of fluids inside a lava lamp is sometimes monitored and recorded. The data gathered from the motion of those fluids can then be transformed into numeric sequences, which are in turn used to generate the desired random integers.[10] Since the movement of the fluids in the lava lamp are nearly impossible to predict, it's also nearly impossible to replicate the seed used to generate random integers. This helps to ensure the security of the cryptosystem. Once a method for obtaining random $\frac{k}{2}$ bit odd integers is in place, the protocol must also determine whether or not those integers are prime. Below I present the *Miller-Rabin Composite Criteria*. This criteria plays an important role in one of the most frequently used primality tests.

The Miller-Rabin Composite Criteria:

Let b be any odd integer greater than 1. Denote $b-1 = 2^u r$ where r is the greatest odd factor of $b-1$.

If there exists an integer a that satisfies all of the following conditions, then b is **not** prime:

Condition 1: $a^r \not\equiv 1 \bmod b$.

Condition 2: $a^{r2^j} \not\equiv -1 \bmod b$ for all $j \in \{0, 1, ..., u-1\}$.

Condition 3: b does not divide a.

Proof of the Miller-Rabin Composite Criteria:

Assume b is prime and that conditions 2 and 3 above hold. I will show that under these assumptions, condition 1 must not hold.

Note that by assumption b is a prime and, since b does not divide a, it follows that $gcf(a, b) = 1$. Thus by *Euler's Theorem*, it can be argued that:

$a^{b-1} = a^{r2^u} \equiv 1 \bmod b$.

Now suppose that $a^{r2^{j+1}} \equiv 1 \bmod b$ for some $0 \leq j < u$. I will show that if this is the case, then it must also be the case that $a^{r2^j} \equiv 1 \bmod b$:

By assumption b is prime. Thus by *Theorem E* in *Appendix I*:

$a^{2r^{j+1}} = (a^{r2^j})^2 \equiv 1 \bmod b \implies$ either $a^{r2^j} \equiv 1 \bmod b$ **or** $a^{r2^j} \equiv -1 \bmod b$.

It is assumed that condition 2 of the *Miller-Rabin Composite Criteria* holds, hence $a^{r2^j} \not\equiv -1 \bmod b$.

Thus it must be the case that if $a^{r2^{j+1}} \equiv 1 \bmod b$, then $a^{r2^j} \equiv 1 \bmod b$.

Since $a^{r2^u} \equiv 1 \bmod b$, the above result proves that $\forall j \in \{0, 1, 2, ..., u-1\}$ it must be the case that $a^{r2^j} \equiv 1 \bmod b$.

But then it must be the case that $a^{r2^0} = a^r \equiv 1 \bmod b$. Hence condition 1 does not hold.

Thus if b is prime and conditions 2 and 3 of the *Miller-Rabin Composite Criteria* hold, then condition 1 of the criteria must not hold. It follows that if conditions 1, 2, and 3 of the *Miller-Rabin Composite Criteria* concurrently hold, then b must not be prime, and since b is an odd integer greater than 1, it follows that in such a scenario, b must be composite.

Q.E.D.

For any positive odd integer b, if an integer, a, is found satisfying all three conditions of the *Miller-Rabin Composite Criteria*, then b is certainly composite. However, even if there exists an integer a that does not satisfy all three conditions, one cannot say for certain that b is prime. The integer, a, described in the *Miller-Rabin Composite Criteria* is known as a base element. It has been shown[11] that if g distinct base elements are tested, each of which satisfy $1 < a < b - 1$, and all g of those base elements fail to prove that b is composite, then the probability that b is composite is less than 2^{-2g}. For larger

values of b, the likelihood of b being a composite that is falsely identified as a prime becomes even smaller.

When probabilistic primality algorithms are used to find candidate primes for RSA, the tests are typically implemented as so that the likelihood of selecting a "false prime" is less than 2^{-80}. This is virtually a guarantee that the chosen integer is a prime - which is important, because the proof of the correctness of RSA depended on p and q being prime. The *Miller-Rabin Composite Criteria* can be used to determine whether or not a randomly generated $\frac{k}{2}$ bit integer is likely prime. First, a security parameter s must be chosen. s indicates the number of distinct base elements one must test before feeling confident that a candidate prime, b, is indeed prime. As mentioned, in practice s is typically chosen as so that the probability of falsely classifying b as prime is less than 2^{-80}. If one uses the error bound mentioned above, letting $s = 40$ provides a guarantee that the probability of falsely classifying an integer as prime is less than 2^{-80}. In practice, the integers to be selected for p and q in an RSA modulus are large integers. As such, the error bound is even smaller, and one can opt for security parameters much smaller than $s = 40$. A chart relating bit length of b to the necessary security parameter when using the *Miller-Rabin Composite Criteria* to test for primality can be found in *Appendix III* (Figure 3).

Below is an outline of an algorithm for a probabilistic primality test using the *Miller-Rabin Composite Criteria*:

Miller-Rabin Primality Test

Step 0: This algorithm takes in an odd integer b such that $b > 1$ and a security parameter $s \in \mathbb{Z}^+$.

Step 1: The algorithm initializes the following variables and the set S:
$r = b - 1$
$u = 0$
$t = 0$
$S = \{\}$

Step 2:

While $(r \ \% \ (2) = 0)$: {
−Assign $u = u + 1$.
−Assign $r = \frac{r}{2}$. }

Step 3:

If $(t < s)$: {
−Assign $t = t + 1$.
−Generate a random integer a such that $a \in \{2, 3, ..., b - 2\}$ but $a \notin S$.
−Assign $S = S \cup \{a\}$.

−**If** $(a^r \ \% \ (b) = 1)$\{
−−Return to the beginning of Step 3. }

−**If** $(a^r \ \% \ (b) \neq 1)$\{
−−Assign $i = 0$.
−−Assign $j_0 = a^r \ \% \ (b)$.

− − **While** $(i < u)$: {

− − −**If** $(j_i \ \% \ (b) = b - 1)$: {
− − −−Return to the beginning of Step 3. }

− − −**Else**: {
− − −−Assign $i = i + 1$.
− − −−Assign $j_i = (j_i)^2$.}}

− − **If** $(i = u)$: {
− − −Terminate the entire algorithm and output: "b is composite
− − −with a as a witness." }}}

Terminate the algorithm and output: "b is probably prime with a security parameter of s."

This algorithm works by first computing u and r such that $b-1 = 2^u r$ where r is the greatest odd factor of $b-1$. It then generates random $a \in \{2, 3, ..., b-2\}$ and uses each of those base elements, a, to test the *Miller-Rabin Composite Criteria*. If a base element serves as a witness of b's compositeness, the algorithm terminates and tells the system that b is composite. If a fails to indicate that b is composite, it is added to the set S. Once S has been filled with s distinct base elements, the algorithm terminates and tells the system that b is likely prime. Note that the random numbers used as witnesses need not be generated in a cryptographically secure manner. The values for a are not used as parts of a cryptographic key, only to test for primality. As such, one may use simple predictable random number generation techniques to generate base elements in this algorithm.

How computationally costly is the *Miller-Rabin Primality Test*? Step 1 is of negligible cost - it simply involves initializing variables. Step 2 is not costly either - it involves repeatedly dividing r by 2 until it is odd. Since r is initially set at $b-1$, the number of divisions by 2 is bound above by $\lfloor log_2(b) \rfloor$. The majority of the cost of the algorithm is incurred in step 3, when using base elements to test the primality of b.

For each base element a, it is possible that the algorithm will need to compute $a^{r2^j} \ \% \ (b)$ for $0 \leq j < u$. For most base elements, not all values of j must be tested - if $a^{r2^j} \ \% \ (b) = -1$ for even one value of j, then the program knows that a cannot serve as a witness of b's compositeness. Shoof (2004)[19] proves testing an individual base element a, requires no more than $(log_2(b))^3$ elementary bitwise operations. It follows that for a security parameter s, the complexity of the algorithm (described using asymptotic complexity notation) is $O(s(log_2(b))^3)$. So if one wants to initialize an RSA protocol with a k bit modulus, what is the ultimate cost of initialization?

By the *Prime Number Theorem*[26], it is known that for a given natural number, n, the probability that a randomly selected natural number no greater than n is prime is approximately $\frac{1}{ln(n)}$. Since the algorithm restricts the randomly generated integers to odd numbers, the probability that a random odd integer, n, is prime is approximately $\frac{2}{ln(n)}$. Thus, the expected number of $\frac{k}{2}$ bit integers that must be tested in order to find a $\frac{k}{2}$ bit prime is approximately $\frac{ln(2^{\frac{k}{2}})}{2}$. As determined above, the cost of performing a primality test on each of these candidate primes, expressed in asymptotic complexity notation, is $O(s(log_2(2^{\frac{k}{2}}))^3)$. Further, the process of finding a $\frac{k}{2}$ bit prime must be done twice, since the protocol needs to determine both a p value and a q value. By compiling these

findings, I have expressed the expected cost of initializing an RSA system using asymptotic complexity notation (below, **e** represents Euler's number):

$$2 * \frac{ln(2^{\frac{k}{2}})}{2} * O(s(log_2(2^{\frac{k}{2}}))^3) =$$

$$\frac{log_2(2^{\frac{k}{2}})}{log_2(\mathbf{e})} * O(s(\frac{k}{2})^3) =$$

$$(\frac{1}{2log_2(\mathbf{e})}) * k * O(\frac{1}{8} * (sk^3)) =$$

$$c_1 k * O(c_2(sk^3)) = O(sk^4)$$

(Properties of complexity analysis are used to simplify the expression in the last line. These properties are proven in *Introduction to Algorithms*[2].)

This gives an approximate bound on the number of bitwise operations that must be performed in order to find appropriate primes for a k bit RSA modulus using a security parameter of s: $O(sk^4)$. Once the values p and q have been found, the process of determining e and d is usually quite simple. As noted, e is typically chosen to be of small size and such that $H(e)$ is relatively small. Note that the most common choice for e, 65537 is itself a prime number. This means the likelihood that $gcf(65537, (p-1)(q-1)) = 1$ is very high. In fact $gcf(65537, (p-1)(q-1)) \neq 1$ only if $65537|(p-1)(q-1)$; this is only expected to be the case about $\frac{1}{65537}$ of the time. In such cases, other values for e must be used. Once e, p, and q are determined, the *Extended Euclidean Algorithm* can be used to determine the decryption exponent, d. The number of elementary bitwise operations that must be performed when using the *Extended Euclidean Algorithm* to determine d is $O(k)$ where k is the bit length of the RSA modulus.[29]

I have shown how RSA can be initialized and how encryption and decryption occurs. There is one more important aspect of RSA that arises in real world applications.

Padding in RSA

The basic RSA protocol provides a good level of security - if a malicious party (who does not hold the private key) intercepts an encrypted message, it's extremely difficult for that party to decrypt the message. However, there are a number of weaknesses in the protocol that have not yet been addressed:

Encryption using RSA alone is referred to as malleable - which means that it is easy for attackers to transform one ciphertext message into another. To illustrate how this is a problem when using RSA alone, consider an example:

Suppose an honest party, Bob, sets up an RSA protocol using the public key (e, n) and the private key d. He publicly broadcasts (e, n). Now suppose another honest user, Alice, wishes to send Bob the message x. Alice would begin by encrypting her message, x, into a ciphertext (we'll call her ciphertext s) using the encryption function: $s = x^e \ \% \ (n)$. Suppose a malicious party, Trouble, then intercepts Alice's encrypted message, s. Using the public key that's available to him, Trouble could compute $p^e s \ \% \ (n)$. Trouble could then, posing as Alice, relay this new message to Bob. When Bob uses his decryption function on the compromised ciphertext, he would find:

$$D(p^e s \% (n)) = (p^e s)^d \ \% \ (n) = px$$

Trouble would have effectively transformed Alice's message into something else, without ever having needed to decrypt her message. For obvious reasons, this is an undesirable trait for a cryptosystem.

Other issues involving using the RSA protocol alone include:

-The plaintext values $0, 1, -1$ always yield a ciphertext of $0, 1, -1$. This is true because for any positive integer e chosen as the encryption power, $0^e = 0$, $1^e = 1$, and $(-1)^e \in \{1, -1\}$.

-When small plaintext values and a small public exponent are used, ciphertext may not be secure. For example, suppose one has a 256 bit modulus, n, a plaintext value, $m = 6$ and that the public exponent is 17. Then the ciphertext is $c = 6^{17} \ \% \ (n) = 6^{17}$. Hence anyone who intercepts the ciphertext can simply compute $p = \sqrt[17]{c}$ and find the associated plaintext value.

-RSA alone is deterministic: given a message x and a public key pair (e, n), the ciphertext of x is always the same, it's $x^e \ \% \ (n)$. This can make it easy for attackers to identify certain recurring messages or patterns.

A solution to these problems is the introduction of padding. Padding is the process of embedding a random structure into plaintext before encrypting it. When RSA is combined with an appropriate padding algorithm, the resulting protocol is no longer malleable, short plaintext messages are no longer vulnerable, and the protocol is non-deterministic. There exist many algorithms for padding which are compatible with RSA. In the document *Public-Key Cryptography Standards* (PKCS)[31], a specific method for padding, known as *Optimal Asymmetric Encryption Padding* (OAEP) is recommended for use with RSA. Although the particulars of padding algorithms vary, the general idea remains the same:

Suppose an RSA protocol is set up using an n bit modulus, and by convention, users agree to use the protocol to encrypt messages of length no greater than m bits, with $m < n - 2$. Let p be a plaintext message to be encrypted by user Alice. If p is less than m bits, Alice begins by concatenating a string of 0's to the left side of p, creating a string p' such that p' has a bit length of exactly m. Alice then generates a random $n - m - 2$ bit binary string, which will be denoted R. She uses p' and R to form a new n bit string as such:

$B = 01|R|p'$

(Here | denotes concatenation.)

B is Alice's padded plaintext, which she can now encrypt using a regular RSA scheme. Once the reciever, Bob, decrypts Alice's ciphertext, c, he knows to look at only the last m bits of the associated plaintext in order to recover Alices original message. Although only the last m bits of the plaintext are meaningful, the same does not hold for the ciphertext. A change to any bits in the ciphertext may have an effect on the last m bits in the associated plaintext. If an attacker were to intercept $c = B^e \ \% \ (n)$ and replace it with $c' = s^e c \ \% \ (n)$, the plaintext message found from decrypting c' would almost certainly not be sp. Infact, the message would likely be entirely garbled and nonsensical. As such, padded RSA no longer suffers from malleability. Since the random structure R is generated each time Alice encrypts her message, identical plaintext messages of the form p can result in many different ciphertexts. This resolves the problem of determinism that exists in RSA.

Mathematical Vulnerabilities of Padded RSA

With the widespread use of RSA, it should come as no surprise that RSA is regarded as a fairly secure cryptographic protocol. The most effective attacks on systems using padded RSA tend to have more to do with the organization responsible for using the RSA protocol than the protocol itself. My analysis, however, will focus on the mathematical vulnerabilities of RSA.

The security of RSA is dependent on the difficulty of factoring large integers. Suppose an RSA protocol were initialized with a public key, (e, n) with $n = pq$. If a malicious party were able to factor n, they would obtain the primes, p and q. They could then use the *Extended Euclidean Algorithm*, along with the public exponent e, to find d such that $ed \equiv 1 \bmod (p-1)(q-1)$. This shows that if the public key is known and n can be factored, then the private key, d, can be easily obtained.

It is widely believed that factoring n is the most efficient mathematical attack on RSA. This is known in literature as The RSA Assumption.[32] As the word *assumption* suggests, there is no known proof for this conjecture. For large integers (integers greater than $10^{100} \cong 2^{332}$ or approximately 332 bits), the most efficient known general factoring technique is the general number field sieve.[33] The computational cost of the general number field sieve (as performed on an integer of size n) is (as expressed in asymptotic complexity notation):

$$exp((\sqrt[3]{\frac{64}{9}} + o(1))(ln(n))^{\frac{1}{3}}(ln(ln(n)))^{\frac{2}{3}}))$$

Common Applications of RSA

Asymmetric protocols (including RSA) are typically used to exchange keys for use in a symmetric protocol. Another important application of RSA is the use of RSA to create digital signatures. Similar to a written signature, digital signatures provide a way of *proving* that a message was created by a particular individual. The RSA protocol can easily be adapted in order to create digital signatures:

Suppose a user, Bob, had a message, m, that he wished to send to another user, Alice. Further, suppose Bob wanted to digitally sign the message - to provide a guarantee to Alice (and anyone else who receives the message) that he was indeed responsible for writing that message. Bob would first generate a public key (e, n) and a private key, d, using the RSA methods described above.

Bob would then compute $s = m^d \ \% \ (n)$. s would be the signature, and would be sent with his message, m. Alice then receives the message m, and the signature s. Using Bob's public key, (e, n) Alice computes $s^e \ \% \ (n)$. If she finds that $s^e \ \% \ (n) = m$, she knows that the message was indeed sent by Bob. If $s^e \ \% \ (n) \neq m$, then the signature is not valid. I will first show that Bob's signatures will indeed be found to be valid:

Note that $s^e = (m^d)^e = m^{ed}$. As per the construction of RSA keys, $m^{ed} \ \% \ (n) = m$. Hence Bob's signature will be valid.

If Bob's private key is kept private, forgery is extremely difficult. Given a message m, a potential forger would need to compute $m^d \ \% \ (n)$. In order to compute this, the forger must first determine the secret key, d. Since the secu-

rity of the RSA protocol as a whole is predicated on attackers being unable to determine d, forging RSA digital signatures is as difficult as breaking a conventional RSA protocol.

The use of RSA to exchange keys for symmetric protocols is quite straight forward. Suppose there exists a client, Alice, and a server, Bob. Bob has already published a public key available to everyone (including Alice). Alice will begin by sending Bob a message, indicating she wishes to share a symmetric key for communication. Bob will then reply, acknowledging Alice's message and digitally signing his acknowledgement. Due to the digital signature, Alice is guaranteed that Bob is indeed who he claims to be. Alice can then take her symmetric key, encrypt it using Bob's public key, and send it to Bob. Bob will use his private key to decrypt the symmetric key. The symmetric key should then be held by Alice and Bob, and no-one else. At this point, Alice and Bob can use the symmetric key for a symmetric cryptography protocol. In real-world implementations, the details of this procedure vary - but the underlying idea behind RSA key exchange is generally the same.

RSA remains a popular method for symmetric key exchange. However, there exists another equally popular family of asymmetric protocols, designed explicitly for key exchange. The next section introduces the *Diffie-Hellman Key Exchange*.

The Diffie-Hellman Key Exchange

The *Diffie-Hellman Key Exchange* is a method of securely exchanging keys over a public channel. The DHKE was first published in 1976 by Whitfield Diffie and Martin Hellman.[27] The classical DHKE is an asymmetric protocol that makes use of large prime numbers. Suppose two parties, Alice and Bob, wish to generate and share a symmetric key with one another. The procedure below outlines how this could be done using the *Diffie-Hellman Key Exchange*.

The Classical Diffie-Hellman Key Exchange

Initialization:

1) Bob generates a random, large prime number, p.

2) Bob selects a random integer $a \in \{2, 3, ..., p-2\}$.

3) p and a are published.

Key Exchange:

1) Bob generates a random integer $b \in \{2, 3, ..., p-2\}$. He computes $s = a^b \ \% \ (p)$. He then broadcasts s.

2) Alice generates a random integer $z \in \{2, 3, ..., p-2\}$. She computes $j = a^z \ \% \ (p)$. She then broadcasts j.

3) Alice computes $s^z \ \% \ (p)$. Note that $s^z \equiv (a^b)^z \equiv (a^{bz}) \bmod p$. This will be the shared key.

4) Bob computes $j^b \ \% \ (p)$. Note that $j^b \equiv (a^z)^b \equiv (a^{bz}) \bmod p$. Hence $j^b \equiv s^z \bmod p$ thus $j^b \ \% \ (p) = s^z \ \% \ (p)$. Thus Alice and Bob have the same key.

Computational Cost of the Classical DHKE

Much of the analysis done on RSA can be applied when assessing the computational complexity of the classical DHKE. The protocol involves:

1) Generating a large prime p:

Any method used to generate large primes for RSA can be used when generating a large prime for DHKE. In my analysis of the *Miller-Rabin Primality Test*, I found that the asymptotic complexity of generating a k bit prime was $O(sk^4)$. (Where s was the security parameter.) For security measures, modern DHKE implementations typically use a p of at least 2048 bits.

2) Selecting random integers $a, b, z \in \{2, 3, ..., p-2\}$:

Aside from the difficulty of seeding random numbers to begin with, the computational cost of finding a random integer in the range $\{2, 3, ..., p-2\}$ does not tend to be significant.

3) Computing a^b % (p) and $(a^b)^z$ % (p)

Again the analysis of the complexity of RSA comes in handy here. Methods for fast modular exponentation such as the *Repeated Square and Multiply Technique* and the *M-ary Techniques* can be applied to classical DHKE. Recall that when using the *Repeated Square and Multiply Technique*, the expected number of multiplications or squarings that must be performed to compute a^b % (p) was approximately $\frac{3}{2}\lfloor log_2(b)\rfloor$. Once one has computed a^b % (p), they can take that value and use it in order to find $(a^b)^z$ % (p). This requires approximately $\frac{3}{2}\lfloor log_2(z)\rfloor$ operations of either squaring or multiplying. Since b and z are randomly selected integers from the range $\{2, 3, ..., p-2\}$, it seems reasonable to estimate that $\frac{p}{2} \cong b \cong z$. Then the expected number of multiplication/squaring operations is approximately:

$\frac{3}{2}\lfloor log_2(b)\rfloor + \frac{3}{2}\lfloor log_2(z)\rfloor \cong$

$\frac{3}{2}(log_2(bz)) \cong \frac{3}{2}(log_2(\frac{p}{2}\frac{p}{2})) =$

$3log_2(p) - 3.$

Altogether, for the user Bob (responsible for finding p and a), when using a k bit prime and s as a security parameter, the asymptotic computational cost of the key exchange is about:

$O(sk^4) + 3log_2(2^k) - 3 = O(sk^4) + 3k - 3 = O(sk^4)$

The Security of Classical DHKE

The algorithm presented above shows why both Bob and Alice end up with the same symmetric key: $(a^b)^z = a^{bz} = (a^z)^b$. But why is it that a malicious third party listening in on their communication wouldn't be able to easily determine the key as well? Note that at no point does Bob broadcast b, nor Alice broadcast z - instead they share only (a^b) and (a^z) respectively. It turns out that even while knowing a, p and a^b % (p), it is not trivial to determine b. In fact, the problem of determining b is so nontrivial (and significant to cryptography), that it has a special name: the Discrete Logarithm Problem (DLP).

Suppose G is a finite cylic group (written using multiplicative notation). Let g be a generator of G and k be any element of G. The discrete logarithm problem in G describes the problem of finding $b \in \mathbb{Z}^+$ such that $g^b = k$.

For Bob's selected prime, p, if an attacker were able to solve the DLP in $(\mathbb{Z}/p\mathbb{Z})^\times$, then that attacker could easily uncover Alice/Bob's secret key as follows:

Let $a^b \ \% \ (p) = c$. The attacker could take the values, p, a, c, all of which were broadcast, and find b such that $a^b = c$ in $(\mathbb{Z}/p\mathbb{Z})^\times$. He could then take Alice's publicly broadcast value, $a^z \ \% \ (p)$ and compute $(a^z)^b \ \% \ (p)$ - giving him the secret key.

Fortunately, the DLP in $(\mathbb{Z}/p\mathbb{Z})^\times$ is considered to be an intractable problem. It is believed that the DLP in $(\mathbb{Z}/p\mathbb{Z})^\times$ is of comparable difficulty to the problem of factoring large integers.[34] However, there is no known proof that there cannot exist a polynomial time algorithm that solves the DLP in $(\mathbb{Z}/p\mathbb{Z})^\times$.

The Generalized Diffie-Hellman Key Exchange

The DHKE can be generalized into a method for key exchange using any finite cycle group. The generalized DHKE is presented below (note that the procedure is written using multiplicative notation for G. $|G|$ denotes the order of the finite group G):

The Generalized Diffie-Hellman Key Exchange

Initialization:

1) A finite cyclic group, G, is chosen and publicly broadcast.

2) Bob finds $g \in G$ such that g is a generator of G. Bob broadcasts g.

Key Exchange:

1) Bob generates a random integer $b \in \{2, 3, ..., |G| - 1\}$. He computes $s = g^b$. He then broadcasts s.

2) Alice generates a random integer $z \in \{2, 3, ..., |G| - 1\}$. She computes $j = g^z$. She then broadcasts j.

3) Alice computes $s^z = (g^b)^z = g^{bz}$. This is the shared key.

4) Bob computes $j^b = (g^z)^b = g^{bz}$. This is the same shared key.

For any finite cyclic group G, Bob and Alice are guaranteed to end up with the same shared key. However, not all choices for G provide adequate security. The security of the generalized DHKE depends on the difficulty of the DLP in the chosen group, G. For example, consider the DLP when G is chosen as $(\mathbb{Z}/q\mathbb{Z})^+$ for some integer q. (Here $(\mathbb{Z}/q\mathbb{Z})^+$ describes the group formed by the set of integers modulo q under the operation addition.)

In such a scenario, an attacker would be given a generator g and $k \in G$ and be asked to find b such that $bg \equiv k \bmod q$. Since g is a generator of $(\mathbb{Z}/q\mathbb{Z})^+$, it is known that $gcf(g, q) = 1$, hence one could easily use the *Extended Euclidean Algorithm* to find a valid value of b. Since the *Extended Euclidean Algorithm* can be performed in polynomial time, the DLP in $(\mathbb{Z}/q\mathbb{Z})^+$ can be solved in polynomial time. As such, performing the DHKE using $(\mathbb{Z}/q\mathbb{Z})^+$ would not be secure. This illustrates the importance of selecting groups G such that the DLP in G is hard.

Note that when G is a subgroup of $(\mathbb{Z}/p\mathbb{Z})^\times$ for some prime p, then the generalized DHKE is precisely the classical DHKE. Another popular family of groups used in the generalized DHKE are groups formed by points on elliptic curves over finite fields. In the next section I will introduce these elliptic curve

based groups.

Nonsingular Elliptic Curves over Fields

Let $\mathbb{F}_p$ be a finite field of characteristic p. A nonsingular elliptic curve over $\mathbb{F}_p$ is the set of points (x, y) with $x, y \in \mathbb{F}_p$ satisfying the equation:

$y^2 = x^3 + ax + b$ where a, b are constants in $\mathbb{F}_p$ such that $4a^3 + 27b^2 \neq 0$.

The set of such points (x, y), combined with an identity element, $\mathcal{O}$, (often referred to as a point at infinity) can be used to form a group. The operation used to define this group is as follows:

Point Addition on Nonsingular Elliptic Curves Over a Finite Field

Let E be a nonsingular elliptic curve over $\mathbb{F}_p$ defined by the equation $y^2 = x^3 + ax + b$. Let $G = E \cup \{\mathcal{O}\}$. Suppose $P, Q \in E$ with $P = (x_1, y_1)$ and $Q = (x_2, y_2)$. Then define $P + Q$ as follows:

If $P \neq Q$ and $x_2 - x_1 = 0$, then $P + Q = \mathcal{O}$

If $P = Q$ and $y_2 = 0$, then $P + Q = \mathcal{O}$

If neither of the above hold, then define s as follows:

$$s = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} \text{ if } P \neq Q \\ \frac{3x_1^2 + a}{2y_1} \text{ if } P = Q \end{cases}$$

And let $P + Q = (x_3, y_3)$ where

$x_3 = s^2 - x_1 - x_2$

$y_3 = s(x_1 - x_3) - y_1$

For any $A \in G$, let it be the case that $\mathcal{O} + A = A$. Combining this with the definition of point-addition, this defines the addition of any two elements of G.

Using this definition of addition makes G a finite abelian group under addition.[35] Any $g \in G$ can be used as a generator to form $< g >$, a cyclic subgroup of G. For certain classes of elliptic curves, G itself will be cyclic. As determined in the

analysis of the generalized DHKE, any finite cyclic group can be used for key exchange. When groups formed by elliptic curves over finite fields are used for the DHKE, the protocol is known as *Elliptic Curve Diffie-Hellman* (ECDH).

Comparing ECDH and RSA

Although the generalized DHKE can be implemented using any finite cyclic group, in order for the key exchange to be secure, the chosen group must have a difficult DLP. Below I present *Hasse's Theorem* (a proof is given by Washington [2008] in *Elliptic Curves: Number Theory and Cryptography*[35]).

Hasse's Theorem

Let $|E|$ denote the number of distinct points on a nonsingular elliptic curve over $\mathbb{F}_p$. Then:

$$p + 1 - 2\sqrt{p} \leq |E| \leq p + 1 + 2\sqrt{p}$$

Hasse's Theorem can be used to determine the size of p (the characteristic of the field) that will be needed in order to provide adequate security. For strategically chosen elliptic curves, the best known algorithms for solving the DLP over the group of points on an elliptic curve are weaker than the best known algorithms for solving the DLP in $(\mathbb{Z}/p\mathbb{Z})^{\times}$. The authors of *On the Security of Elliptic Curve Cryptosystem Against Attacks with Special-Purpose Hardware* [2006] implemented specialized algorithms to solve the DLP over groups of points on nonsingular elliptic curves.[39] Specifically, they considered groups of points formed on elliptic curves E, over $\mathbb{F}_p$ where p was an 163 bit prime. They found that solving the DLP over such groups was thousands of times more difficult than cracking an RSA implementation with a 1024 bit modulus. They showed that elliptic curve cryptography can provide comparable security to RSA while using significantly smaller key-sizes.

Due to the fact that ECC allows for smaller key sizes, initialization of systems using ECC tends to be faster than initializing an RSA protocol.[40] However, there remain some advantages of RSA. RSA has been time-tested and is compatible with many legacy systems. The details of the math and protocols behind ECC are more complicated than the details of RSA. As such, it's easier for mistakes to be made when implementing ECC algorithms. There are certain tasks that RSA accomplishes more efficiently than ECC: verifying the validity of digital signatures, and encrypting data. Many experts believe that in the coming decades, ECC will overtake RSA as the most popular method for public-key cryptography.[41]

Avenues for Further Research

Asymmetric cryptography is field with a great degree of both breadth and depth. Many variants of popular systems have been explored and implemented. RSA can be adapted into multi-power multiprime RSA, where the modulus used is of the form $\Pi_{i=0}^{i=n} p_i^{r_i}$ and each p_i is a distinct prime. Algebraic groups can be defined on curves other than nonsingular elliptic curves - such as hyperelliptic curves. These groups can then be used to implement key exchanges and generate digital signatures. There are many opportunities for further research into methods for fast modular exponentiation to expedite the process of encryption and decryption using asymmetric systems. Any research made in primality testing and integer factorization also has a significant effect on the field of asymmetric cryptography.

Conclusion

Asymmetric cryptography is commonly used to solve the key distribution problem and to create digital signatures. The validity of RSA, the most common method of asymmetric cryptography, can be proven using results from elementary number theory. Despite the simplicity of RSA, there is a great deal of nuance to its implementation. When performing modular exponentiation, using only straightforward multiplication is unfeasible. Efficiency gains can also be found through the strategic selection of parameters when initializing RSA. Specifically, the encryption key should be relatively small number with a low Hamming weight. Generating random large prime numbers for RSA initialization is a nontrivial task. In practice, nondeterministic primality tests are used to find integers that are very likely (but not certainly) prime.

My exploration of the generalized DHKE showed how any finite cyclic group can be used in a key exchange protocol. Although any such group will guarantee that a key is successfully exchanged, only groups with a difficult DLP provide adequate security. The generalized DHKE is most often performed over either $(\mathbb{Z}/p\mathbb{Z})^\times$ or a group formed by a nonsingular elliptic curve over $\mathbb{F}_p$. Given similar key sizes, ECC provides a greater degree of security than RSA or the classical DHKE. However, there remain some advantages to RSA - RSA is a time-tested protocol, while ECC is relatively new. Encryption of data and the verification of digital signatures can be performed faster when using RSA than it can with any other common asymmetric cryptosystem.

References

[1]Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 3). Springer.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., &; Stein, C. (2013). In Introduction to algorithms (pp. 44–53), The MIT Press.

[3] Kumar, Yogesh & Munjal, Rajiv & Sharma, Harsh. (2011). Comparison of Symmetric and Asymmetric Cryptography with Existing Vulnerabilities and Countermeasures. International Journal of Computer Science and Management Studies. 11.

[4] Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 179). Springer.

[5]Rivest, R.L., Shamir, A., & Adleman, L.M. (1978). A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM, 21, 120-126.

[6] Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 162). Springer.

[7] Barker, Elaine; Dang, Quynh (2015-01-22). "NIST Special Publication 800-57 Part 3 Revision 1: Recommendation for Key Management: Application-Specific Key Management Guidance" (PDF). National Institute of Standards and Technology: 12. doi:10.6028/NIST.SP.800-57pt3r1. Retrieved 2017-11-24.

[8] Kaliski, Burt (October 22, 1997). "Growing Up with Alice and Bob: Three Decades with the RSA Cryptosystem". Archived from the original on September 29, 2011. Retrieved April 29, 2017.

[9]L. Euler, "Speculationes circa quasdam insignes proprietates numerorum," Acta Acad. Sci. Imp. Petropol. 4 (1784), 18–30. URL https://scholarlycommons.pacific.edu/euler-works/564/. English translation at https://arxiv.org/abs/0705.3929v1.

[10]Liebow-Feeser, J. (2018, August 29). Randomness 101: LavaRand in production. The Cloudflare Blog. Retrieved March 30, 2022, from https://blog.cloudflare.com/randomness-101-lavarand-in-production/

[11]Rabin, M. O. (1980). Probabilistic algorithm for testing primality. Journal of Number Theory, 12(1), 128–138. https://doi.org/10.1016/0022-314x(80)90084-0

[12]Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 191). Springer.

[13] Berstein, Daniel J.; Birkner, Peter; Lange, Tanja; Peters, Christiane (January 9, 2008). "ECM Using Edwards Curves". Cryptology ePrint Archive.

[14]Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 154). Springer.

[15]Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 153). Springer.

[16]Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 174). Springer.

[17]Guner, H., Cenk, M., &; Cagdas, C. (2017). Efficient Modular Exponentiation Methods for RSA. METU Cryptography and Mathematics.

[18] Nedjah, N., & de Macedo Mourelle, L. (2005). Efficient pre-processing for large window-based modular exponentiation using Ant Colony. Lecture Notes in Computer Science, 640–646. https://doi.org/10.1007/11554028_89

[19] Schoof, René (2004), "Four primality testing algorithms", Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography, Cambridge University Press, ISBN 978-0-521-80854-5

[20]Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 183). Springer.

[21]Boneh, Dan (1999). "Twenty Years of attacks on the RSA Cryptosystem". Notices of the American Mathematical Society. 46 (2): 203–213.

[22]Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 184). Springer.

[23]Kritsanapong Somsuk. The Improvement of Elliptic Curve Factorization Method to Recover RSA's Prime Factors. Symmetry. 2021;13(1314):1314. doi:10.3390/sym13081314

[24] Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 186). Springer.

[25] Menezes, A. J., C., V. O. P., & Vanstone, S. A. (2001). In Handbook of Applied Cryptography (p. 145). essay, CRC.

[26] Hardy, G.H.; Littlewood, J.E. (1916). "Contributions to the theory of the Riemann zeta-function and the theory of the distribution of primes". Acta Mathematica. 41: 119–196. doi:10.1007/BF02422942. S2CID 53405990.

[27]Merkle, Ralph C. (April 1978). "Secure Communications Over Insecure Channels". Communications of the ACM. 21 (4): 294–299. CiteSeerX 10.1.1.364.5157. doi:10.1145/359460.359473. S2CID 6967714. Received August, 1975; revised September 1977

[28] Corcoran, L., &; Jenkins, M. (2022). Commercial National Security Algorithm (CNSA) suite cryptography for internet protocol security (IPsec). https://doi.org/10.17487/rfc9206

[29]Phiamphu D, Saha P. Redesigned the Architecture of Extended-Euclidean Algorithm for Modular Multiplicative Inverse and Jacobi Symbol. 2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI), Trends in Electronics and Informatics (ICOEI), 2018 2nd International Conference on. May 2018:1345-1349. doi:10.1109/ICOEI.2018.8553922

[30]Bodapati N, Pooja N, Varshini EA, Jyothi RNS. Observations on the Theory of Digital Signatures and Cryptographic Hash Functions. 2022 4th International Conference on Smart Systems and Inventive Technology (ICSSIT), Smart Systems and Inventive Technology (ICSSIT), 2022 4th International Conference on. January 2022:1-5. doi:10.1109/ICSSIT53264.2022.9716495

[31]J. Jonsson and B. Kaliski. 2003. RFC3447: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC Editor, USA.

[32]:Yan, S. Y. (2008), Cryptanalytic attacks on RSA (pp. 71). Springer.

[33]Pomerance, Carl (December 1996). "A Tale of Two Sieves". Notices of the AMS. Vol. 43, no. 12. pp. 1473–1485

[34] Yan, S. Y. (2008), Cryptanalytic attacks on RSA (pp. 51). Springer.

[35] [Washington] Lawrence C. Washington. Elliptic Curves: Number Theory and Cryptography, ed. 2. Discrete Mathematics and Its Applications 50, 2008.

[36] Paar, C., & ; Pelzl, J. (2010). Understanding cryptography (pp. 251-252). Springer.

[37] Yan, S. Y. (2008), Cryptanalytic attacks on RSA (pp. 109). Springer.

[38] Atkins, D., Graff, M., Lenstra, A. K., &; Leyland, P. C. (1995). The magic words are squeamish ossifrage. Advances in Cryptology — ASIACRYPT'94, 261–277. https://doi.org/10.1007/bfb0000440

[39] Güneysu, Tim & Paar, Christof & Pelzl, Jan. (2006). On the security of elliptic curve cryptosystems against attacks with special-purpose hardware.

[40]National Security Agency (NSA), "The Case for Elliptic Curve Cryptography," USA, 2015

[41]Douligeris, C.; D. N. Serpanos; "Network Security: Current Status and Future Directions," IEEE, 2007

Appendix I: Auxiliary Proofs

Theorem A: Let n be a positive integer. Let a_1 and a_2 be integers such that $a_1 \equiv a_2 \bmod n$ and $0 \leq a_1, a_2 < n$. Then it must be the case that $a_1 = a_2$.

Proof: Suppose it were the case that $a_1 \equiv a_2 \bmod n$ and $0 \leq a_1, a_2 < n$ but $a_1 \neq a_2$. Without loss of generality one can assume $a_1 > a_2$. By the definition of modular congruence: $a_1 \equiv a_2 \bmod n \implies a_1 - a_2 = kn$ for some integer k.

Since $a_1 > a_2$, it is known that $(a_1 - a_2) > 0$ thus $kn = (a_1 - a_2) > 0$, and, since $n > 0$, it follows that $k > 0$.

Since k is an integer and $k > 0$, it follows that $k \geq 1$. Hence, $kn \geq n$. Then it must be the case that $(a_1 - a_2) = kn \geq n$.

But $a_1 < n$ and $a_2 \geq 0$, therefore $(a_1 - a_2) \leq a_1 < n$. So if $a_1 \neq a_2$, it would have to be the case that $n > (a_1 - a_2) \geq n \implies n > n$. Clearly this is a contradiction, thus it cannot be the case that $a_1 \neq a_2$. It follows that for any positive integers n, a_1, a_2, if it is the case that $0 \leq a_1, a_2 < n$ and $a_1 \equiv a_2 \bmod n$, then it must also be the case that $a_1 = a_2$.

Q.E.D.

Theorem B: Let n be a positive integer. Let a, b, x, y be integers such that $a \equiv x \bmod n$ and $b \equiv y \bmod n$. Then $a * b \equiv x * y \bmod n$.

Proof: Since $a \equiv x \bmod n$, it must be true that $a = kn + x$ for some integer k. Similarly, since $b \equiv y \bmod n$, it is the case that $b = jn + y$ for some integer j. Thus:

$$ab = (kn + x)(jn + y) = (kjn) * n + (jx)n + (ky)n + xy = n(kjn + jx + ky) + xy$$

Since $(kjn + jx + ky)$ is an integer, $ab = tn + xy$ for some integer t. Thus $ab \equiv xy \bmod n$.

Q.E.D.

Theorem C: Let x, y be positive integers such that $gcf(x, y) = 1$. Suppose $a \equiv b \bmod x$ and $a \equiv b \bmod y$. Then $a \equiv b \bmod xy$.

Proof:

By definition, $a \equiv b \bmod x$ implies $a = jx + b$ for some integer j.

Similarly, $a \equiv b \bmod y \implies a = ky + b$ for some integer k.

Thus $a = ky + b = a = jx + b \implies ky = jx$

Since $ky = jx$, and x clearly divides jx, it must also be the case that x divides ky. Since $gcf(x, y) = 1$, if x divides ky then it must be the case that x divides k. Thus $k = tx$ for some integer t.

It follows that $a = ky + b = txy + b \implies a - b = t(xy)$ and $a \equiv b \bmod xy$.

Q.E.D.

Theorem D: Let a, b, n, e be integers with $n, e > 0$. If $a \equiv b \bmod n$ then $a^e \equiv b^e \bmod n$:

Proof: This will be proven by induction on e.

BASIS CASE ($e = 1$): Clearly if $a \equiv b \bmod n$ then $a^1 \equiv b^1 \bmod n$.

INDUCTION: Assume $a^{e-1} \equiv b^{e-1}$. Then:

$a^e = a * a^{e-1}$

Using the premise $a \equiv b \bmod n$, as well as the inductive assumption $a^{e-1} \equiv b^{e-1} \bmod n$, and the results from *Theorem B*:

$a(a^{e-1}) \equiv b(b^{e-1}) \bmod n$

And clearly $b^e \equiv b(b^{e-1}) \bmod n$.

Thus the inductive assumption holds, and $a^e \equiv b^e \bmod n$.

Q.E.D.

Theorem E: Let p be a prime and a be any integer such that $a^2 \equiv 1 \bmod p$. Then it is either the case that $a \equiv 1 \bmod p$ or $a \equiv -1 \bmod p$:

Proof: Suppose $a^2 \equiv 1 \bmod p$. Then:

$a^2 - 1 \equiv 0 \bmod p \implies$
$(a - 1)(a + 1) \equiv 0 \bmod p \implies$

$(a-1)(a+1) = kp$ for some integer $k \implies$
$p|(a-1)$ or $p|(a+1) \implies$
It is either the case that $(a-1) \equiv 0 \bmod p$ and $a \equiv 1 \bmod p$; or it is the case that $(a+1) \equiv 0 \bmod p$ and $a \equiv -1 \bmod p$.

Q.E.D.

Appendix II: Glossary of Terms

Asymmetric cryptography: Asymmetric cryptography is a family of cryptographic methods that use pairs of keys. Each pair consists of a public key (which is broadcast to everyone) and a private key (which should only be known by a select few). Asymmetric cryptography is also known as public key cryptography. Cryptographic protocols that make use of asymmetric methods are often referred to as public key cryptosystems.

Asymptotic complexity notation: Asymptotic complexity notation is a way of comparing the cost of different algorithms on arbitrarily large inputs. The specific meaning of some common asymptotic complexity notation is shown below[2]:

$$\boxed{g(n) = \Theta(f(n)) \longleftrightarrow \exists c_1, c_2, n_0 \in \mathbb{R}^+ \text{ such that } 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n) \text{ for all } n \geq n_0}$$

$$\boxed{g(n) = O(f(n)) \longleftrightarrow \exists c_1, n_0 \in \mathbb{R}^+ \text{ such that } 0 \leq g(n) \leq c_1 f(n) \text{ for all } n \geq n_0}$$

Asymptotic complexity is often used to compare a particular cost function to a class of functions. Suppose $g(n) = O(p(n))$ where $p(n)$ is some polynomial function. Then there exists some polynomial $p(n)$ with real valued coefficients such that $g(n) = O(p(n))$.

Base element in Miller-Rabin primality testing: A base element is an integer that plays a role in Miller-Rabin probabilistic primality tests. Suppose one has an odd integer p and wishes to determine whether or not p is likely prime. In doing so, one must select integers a such that $a \in \{2, 3, ..., p-2\}$ and test to see whether or not a serves as a witness of p's compositeness. An integer selected as a is referred to as a base element.

Canonical complete residue system modulo n: Let n be a positive integer. Then the set $\{0, 1, 2, ..., n-1\}$ is known as the canonical complete residue system modulo n.

Chinese Remainder Theorem: The Chinese Remainder Theorem states that if one knows the remainders of the Euclidean division of an integer n by several

integers, then one can determine uniquely the remainder of the division of n by the product of these integers, under the condition that the divisors are pairwise coprime (no two divisors share a common factor other than 1).

Ciphertext: Plaintext messages contain the original data intended to be communicated along a secure channel. When encryption occurs, a plaintext message is transformed into an associated ciphertext. When decryption occurs, the ciphertext is transformed back into the plaintext message as so that the receiver can interpret it.

Diffie-Hellman Key Exchange (DHKE): The Diffie-Hellman key exchange is a method of securely exchanging cryptographic keys over a public channel. The Diffie-Hellman key exchange, named after Whitfield Diffie and Martin Hellman, was one of the first public-key protocols.[27] Although the Diffie-Hellman key exchange was originally implemented using the multiplicative group of integers modulo a prime, the process has since been generalized. The generalized version of the Diffie-Hellman key exchange can be implemented using any finite cyclic group.

Digital signatures: A digital signature is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature, where the prerequisites are satisfied, gives a recipient very strong reason to believe that the message was created by a known sender, and that the message was not altered in transit.[30]

Discrete Logarithm Problem in G: Let G be a finite cyclic group (written using multiplicative notation.) Let g be a generator of G and let k be any element of G. The discrete logarithm problem in G describes the problem of finding $b \in \mathbb{Z}^+$ such that $g^b = k$.

Elliptic-Curve Cryptography (ECC): Elliptic-curve cryptography is an approach to public key cryptography that is based on the algebraic structure of elliptic curves over finite fields.[28]

Elliptic Curve Factorization Method: An efficient method for finding small factors of a positive integer. This method uses elliptic curves. It is considered the best modern algorithm for finding divisors that do not exceed 50 to 60 digits.[13]

Euler Totient Function: The Euler totient function is a function that takes positive integers as its input. It is defined as follows:

$\phi(p) = |S|$ where $S = \{n | n \in \mathbb{Z}^+; n \leq p; gcf(p, n) = 1\}$ and $|S|$ denotes the cardinality of S.

Hamming weight: The Hamming weight of a string is the number of symbols that are different from the zero-symbol of the alphabet used. If x is a binary

string, then the hamming weight of x is equal to the number of 1's in x.

Key distribution problem: When symmetric key algorithms are used in cryptography, a secret key must be shared between all parties communicating. The problem of securely sharing that secret key is known as the key distribution problem.

Malleable cryptosystems: A cryptosystem is referred to as malleable if there is an easy way for an attacker to transform a ciphertext, c into another valid ciphertext, c' with a different meaning.

M-ary Techniques for modular exponentiation: M-ary techniques are a family of techniques used to expedite modular exponentiation. In order to compute x^e % (n), this family of techniques represents e using a base M system, where M is some power of 2.

Padding in cryptography: Padding describes a process by which random structures are embedded into plaintext before it is encrypted. This provides additional security, and, in some protocols, standardizes message length.

Plaintext: Plaintext messages contain the original data intended to be communicated along a channel. When encryption occurs, a plaintext message is transformed into an associated ciphertext. When decryption occurs, the ciphertext is transformed back into the plaintext message as so that the receiver can interpret it.

Public key Cryptography Standards: A family of standards published by *RSA Laboratories*. [8] It provides the basic definitions of and recommendations for implementing the RSA algorithm for public key cryptography. It defines the mathematical properties of public and private keys, primitive operations for encryption and signatures, secure cryptographic schemes, and related syntax representations.

Public key cryptosystem: When asymmetric methods are used to develop a system for encryption and decryption, that system is known as a public key cryptosystem.

Repeated Square and Multiply Technique: A technique for efficiently computing m^e % (n) for some $m, e, n \in \mathbb{Z}^+$. If e is a randomly selected positive integer, the number of squaring or multiplication operations expected to be performed in order to compute m^e % (n) is approximately $\frac{3}{2}log_2(e)$.

Rivest-Shamir-Adleman (RSA): An asymmetric cryptographic scheme introduced in 1977 by Ronald Rivest, Adi Shamir, and Leonard Adleman. The security of RSA relies on the difficulty of factoring large integers.[16]

Symmetric key cryptography: Symmetric key algorithms are algorithms for cryptography that use the same cryptographic keys for both the encryption of plaintext and the decryption of ciphertext. The keys may be exactly identical, or there may be a simple transformation used to go between the two keys.

Sliding Window Techniques: A family of techniques used for efficient modular exponentiation. When computing $x^e \ \% \ (n)$, sliding window techniques involve representing e in binary. Supposed e is a k bit integer. The k bits representing e will be partitioned into sections and computations will be performed on each of those sections independently. The results of those computations will be combined in order to compute $x^e \ \% \ (n)$.

The RSA Assumption: The assumption that if given an RSA public key, (e, n) and an RSA ciphertext c, then the problem of finding the plaintext p that is associated with the ciphertext c is at least as hard as factoring the modulus n.

Appendix III: Tables and Charts

Year	Digits	Bits
1964	20	
1974	45	
1984	71	
1991	100	332
1992	110	365
1993	120	398
1994	129	428
1996	130	431
1999	140	465
1999	155	512
2003	160	530
2003	174	576
2005 (Nov)	200	663
2005 (May)	193	640

Figure 1: Taken from *Cryptanalytic Attacks on RSA (2008)*[37]. Records for the largest RSA modulus factored between the years 1964 and 2005. The first column indicates the year the modulus was factored, the second column indicates the decimal length of the modulus, and the third column indicates the bit length of the modulus. Each modulus listed here is of the form $p * q$, where p and q are distinct primes. For each modulus, the prime factors of that modulus are each approximately half the bit length of the modulus.

TABLE IX
RUNNING TIME RESULTS FOR THE MODIFIED M-ARY METHOD WITH POWER OF 2

modulus size	m	running time (ms)	comparison with m=2	saving (%)
1024-bit	2	4.7	1	0
	4	4.07	0.867	13.3
	8	3.67	0.781	21.9
	16	3.5	0.745	25.5
	32	**3.38**	**0.719**	**28.1**
	64	3.39	0.721	27.9
	128	3.59	0.766	23.4
	256	4.54	0.967	3.3
2048-bit	2	32.32	1	0
	4	29.03	0.898	10.1
	8	26.20	0.811	18.9
	16	24.94	0.772	22.8
	32	24.14	0.747	25.3
	64	**23.74**	**0.735**	**26.5**
	128	23.9	0.74	26
	256	25.58	0.791	20.9
3072-bit	2	103.96	1	0
	4	93.48	0.899	10.1
	8	87.08	0.838	16.2
	16	81.79	0.787	21.3
	32	77.33	0.744	25.6
	64	76.02	0.731	26.9
	128	**75.32**	**0.724**	**27.6**
	256	77.63	0.747	25.3
4096-bit	2	213.79	1	0
	4	194.59	0.91	9
	8	182.15	0.852	14.8
	16	173.11	0.81	19
	32	168.24	0.787	21.3
	64	163.15	0.763	23.7
	128	**161.25**	**0.754**	**24.6**
	256	164.07	0.767	23.3

Figure 2: Taken from *Efficient Modular Exponentiation Methods for RSA*[17]. m represents the size of the base chosen for the M-ary method. $m = 2$ corresponds to the repeated square and multiply method.

Table 7.2 Number of runs within the Miller–Rabin primality test for an error probability of less than 2^{-80}

Bit lengths of $\tilde{p}$	Security parameter s
250	11
300	9
400	6
500	5
600	3

Figure 3: Taken from *Understanding Cryptography*[12]. A chart relating the bitlength of a candidate prime, $\tilde{p}$ to the security parameter s needed to ensure that the probability that $\tilde{p}$ is falsely classified as prime is less than 2^{-80}