Douglas Richardson

A study into modern cryptography

Abstract

In this paper I go into detail of many different modern encryption algorithms. I start by looking at outdated methods, then I will go into detail about the RSA encryption algorithm and the Elliptic Curve Digital Signature Algorithm. I will then discuss different methods I have implemented in my code to attempt to break RSA encryption as well as my findings after testing said methods. At the end of this paper, after my references, I have included all of the code used in these implementations and tests. They should all run properly with python 3.0 or higher. No additional packages are required.

Introduction

Before this paper properly begins, there are a few mathematical concepts that I will be talking about in the paper that I will be clarifying briefly here. Many of these are quite common concepts so it is likely you have seen some of them before if you are familiar with higher level mathematics or computer science. I will not be dwelling on them too long.

Greatest Common Divisor, also known as Greatest Common Factor and abbreviated as either GCD or GCF, is the largest integer that two other numbers can be divided by without having a remainder. I may say something like GCD(12,42), this means the largest integer that can evenly divide both 12 and 42, in this case GCD(12,42) = 6, since 6 can divide 12 (12/6 = 2) and 6 can divide 42 (42/6 = 7). While there are many methods to calculate GCD, whenever I mention it in any sort of algorithm it can be assumed I am using the euclidian algorithm for GCD which is based on the fact that if $x = GCD(a,b)$ then $x = GCD(b,a-b)$.

Relatively Prime is the notion that two numbers have no common divisors. That is to say, if $GCD(a,b) = 1$ then *a* is relatively prime to *b* and *b* is relatively prime to *a*. This does not mean that either number is necessarily a prime number. For example GCD(8,15) = 1, but 8 is a factor of 4, and 15 is a factor of 5. Neither 8 nor 15 are prime numbers,however they are relatively prime to each other.

Modular Arithmetic is the idea of a finite additive and multiplicative ring where a number is selected as a base to act like the ring’s zero. To put it in simpler terms, it is a section of integers that loop on itself. As an example, the hours in a day on a clock are using modular arithmetic base 12 (or just mod 12). It starts at 0 (or 12) then increments as numbers normally would until it loops on itself. So it goes 0,1,2,3,4,5,6,7,8,9,10,11 then back to 0. This acts much like the normal integers and many of the same algebra still works. 4+4mod 12 is still 8, but 10+5 mod 12 = 3. Technical mathematicians will point out we don't really use "equals" in modular arithmetic, we use "Congruent" but I don't have enough keys on my keyboard to fit that symbol so I will be using it interchangeably, and if you are the type of person who that matters a lot for you are also the type of person who knows what I mean when I say 15 mod 12 = 3.

Modular inverse is the multiplicative inverse in modular arithmetic. What I mean by that is that division doesn't really work in modular arithmetic so we come up with a fancy name to say 'the thing you multiply this thing by to get 1'. We use the normal multiplicative inverse symbol of using a power of -1 to indicate modular inverse. For example 2^{-1} mod 7 = 4 since 2*4 mod 7 = 8 mod 7 = 1 mod 7 so 4 is the modular inverse of 2.

A group is a set of symbols with an operation to go from two symbols to one symbol. Groups are a really big idea in mathematics, and this paper will not go into a huge amount of detail on them. Some examples of groups are the integers and integers with any modular arithmetics applied to them.

Rings, mentioned very briefly above, are groups that have two operations, usually called addition and multiplication, where addition and multiplication have the distributive property. The integers is also a Ring, the real numbers are a ring, the complex numbers are a ring. If you are guaranteed to be able to undo addition and multiplication the ring is called a field. The integers are not a field, since there is no symbol to undo the multiplication of a 2 in the integers, but the integers in a modular arithmetic with a prime base are a field since there is always a multiplicative inverse in prime modular arithmetic.

Group order means the number of distinct symbols within a group. I may also reference an element of a group having an order all it's own. The order of an element of a group is the number of times that element can be added to itself before ending up back to the original element.

Running time is a ballpark estimate on how long something will take to complete. Specifically it has to do with the number of operations an algorithm will take to fully execute. I may write it in 'Big O notation' this usually looks something like this:$O(n^2)$. Big O notation is a function that says this algorithm will complete in some scalar multiple of this function to complete. So an algorithm that has a running time of $O(n^2)$ will take some constant multiplied by n squared operations to complete. In general the smaller the function is for larger numbers, the faster the algorithm will be complete. The value of n depends on the algorithm and what it is doing, but all the times I mention it in this paper I am talking about the same kind of algorithm (algorithms for factoring the product of two large primes) and in those instances n is referring to the product of the primes in question (AKA the input for the algorithm).

A Factorial is a number composed of a number of incrementing products. A factorial is usually denoted by an exclamation point. For example *5!* Isn't 5 really loud, it's *1*2*3*4*5* which is 120. Factorials also have the nice property that $(n+1)! = n! * (n+1)$. Factorials are only really a thing for integers and don't really mean anything for non-integer numbers.

A hash function is a function that scrambles a string in such a way that Hash(one string) is wildly different from Hash(the same thing with one bit changed). All the hash functions that are generally used are standardized. So Hash("one thing") is an easily calculable result, where you could not predict what "one thing" was given the hash of it.

Cryptography Introduction

Cryptography is the science and study of obscuring information. Cryptography attempts to encrypt information to make it illegible and then decrypt the information such that no information in the original message is lost. The idea of encryption is one that even kids understand, the idea of coming up with secret phrases and code words. In a way even simple concepts like slang words or innuendos share this idea of communicating in a way others don't understand. In this paper however we will be focusing on computer encryption, specifically, taking a message from one person and transferring it to another in a way that any person who received the message in transit would have no way of knowing what was being said.

The Caesar Cipher

The simplest form of encryption is typically referred to as a Caesar Cipher. A Caesar cipher is a method of shifting the letters along a line a fixed amount. You can think of this like modular arithmetic on the letters. This word *ecvu* means nothing unless you were told all of the letters were replaced with a letter 2 letters down in the alphabet, then you could shift the letters back so e becomes c, c becomes a and so on to get the decrypted word *cats*. Now this cipher is super easy to understand and easy to create however it has two massive weaknesses.

The first problem with Caesar ciphers is that repeated messages may give insight to the algorithm. For example, if in a military messages were sent with a Caesar cipher and every message had to have the sender's name at the end, it would give away what the cipher is because anyone intercepting can reverse engineer the cipher. You can remove such things however it is entirely possible simply because of how language works to decrypt such a message without needing such consistencies. For example, in this paragraph there are exactly 121 'e' characters, far more than any other single character. If someone received this paragraph that had been encrypted with any algorithm that simply shuffles the alphabet, they could see that any letter that shows up more than any other is probably an ‘e’. Similarly, if the space character was not included in this cipher, then the ‘in’,’if’, and ‘is’ words would be a dead giveaway as to what letter was really ‘i’. Add in the standard rules of grammar, subject matter, and human intuition, it isn’t unreasonable for a clever person to be able to crack the code within a matter of days, if not hours.

Now all of those problems are actually solvable, and it is possible to make an unbreakable encryption algorithm by using a similar process, however there is one massive flaw in this style of encryption. This kind of encryption is called Symmetric Key Encryption. This means that the key to encrypt is directly related (if not exactly the same as) the key to decrypt. The problem here is that if someone gets ahold of that key, they can read everything. Now this might not sound that bad, just don’t let anyone get your private keys, but the issue is this makes it vulnerable to man in the middle attacks. When you are first begin using a symmetric key encryption algorithm, two people need to communicate about what key they are using. If somehow someone else was listening in on their communication at the start, they would have the key and render the whole thing pointless. However there is a way to make it so someone else can know everything about the algorithm, how it works and how to use it, while still being unable to decrypt without a private key that only one party needs to keep. It is called Asymmetric key encryption.

Asymmetric Keys:RSA encryption

The RSA encryption algorithm is an asymmetric key algorithm. This means that one part of the algorithm is deliberately open for anyone to see. In the case of RSA, this allows anyone to securely communicate with the creator of the public key. What makes RSA unique compared to other encryption techniques discussed earlier is that it is a one way algorithm, it uses a different key to encrypt information than it does to decrypt it. This means that RSA is particularly resilient to man in the middle attacks. An overview of how the algorithm works is that the private key is a set of two large prime numbers and the public key is the product of those primes. To encrypt you can use the public key in a fancy way that scrambles things in a difficult to reverse way. To un-encrypt you use the private key and do some other fancy math that reverses the encryption. Because the private key is two primes and the public key is a product of those primes, the only way to get the private key, and therefore the only way to decrypt the message, is to factor the public key into the prime factors, something we will discuss in a later section. Obviously the exact process is more complicated than that, so let's dive into some of that 'fancy math'.

Okay so you start with your two massive prime numbers, normally we denote them with the letters 'p' and 'q'. The product of these two numbers is usually denoted as 'n', so $p*q=n$. One more part of our algorithm that we need is a number that is relatively prime to both n and the product of p and q minus one. We call this number 'e' and as stated it has the property that

$$GCD(e,n) = GCD(e,[(p-1)*(q-1)]) = 1$$

What number exactly isn't important as long as it isn't that small. There is one last number we will be referencing and that is the private key. The private key is found with the following formula

$$d = e^{-1} mod(p-1*q-1)$$

From this we have both our encryption and decryption keys. The public key is *e* and *n* while the private key is *d*. Now we can get into how to use this algorithm.

The specifics of it are up to exact implementation but at the end of the day if one person, call them Alex, wants to send a message to another person, call them Betsy, and they want to use RSA to communicate securely, here is what they do. To start Betsy goes through all the motions mentioned earlier, so she has *p,q,n,e* and *d*. Betsy then tells Alex the value of *n* and *e*, her public keys. Even if the channel they are using is insecure, that doesn't matter since it is a public key. Alex now has a message which can be encoded as a number or multiple numbers, again exact details depend on implementation. Let this message simply be represented by the number '*M*'. To encode, Alex performs the following operation

$C=M^e \bmod n$

Alex then sends Betsy this new number *C*. Using her private key, Betsy can get the original message back by using this operation

$M=C^d \bmod n$

This is great news because the only information traveling through the insecure channel is the public key, only useful for encrypting information, and the encrypted message, impossible to decode without the private key.

Now if you attempt using the algorithm like this you can see that it totally works, the encrypted message is illegible and when you use the decryption algorithm on the encrypted message you get the original message back again. Before we start with a proof that RSA always works just know there is a theorem this proof hinges on known as Fermat's Little Theorem. Fermat's little theorem states that for any prime *p* and any integer *a*, $a^{p-1} = 1 \bmod p$. What's more is that for any composite number $n = pq$ $a^{p-1} = 1 \bmod n$ as well. Now we can get into why exactly it works. The most complicated part is the fact that for some reason $(M^e)^d = C^d = M \bmod n$ where e is relatively prime to $(p-1)(q-1)$ and pq and d is the modular inverse of e modbase $(p-1)(q-1)$. All of this means that $de = 1 + k(p-1)(q-1)$ for some integer k. So

$(M^e)^d = M^{de} \bmod n$ by exponent rules

$M^{de} = M^{1+k(p-1)(q-1)} \bmod n$ by replacing de with $1+k(p-1)(q-1)$

$M^{1+k(p-1)(q-1)} = M*(M^{(p-1)})^{k(q-1)} \bmod n$ Exponent rules and shifting things around

$M*(M^{(p-1)})^{k(q-1)} = M * 1^{k(q-1)} = M \bmod n$ by fermat's little theorem $M^{p-1} = 1 \bmod p*q$. So $(M^e)^d = M \bmod n$ so the algorithm works.

One thing you might have noticed is that, technically, there is a way someone can reverse engineer a private key from a public key. Because the public key, n, is a product of two primes and the private key, d, is the modular inverse of the other part of the public key, e, with modbase $(p-1)(q-1)$. So all you need to do to get the private key is factor two numbers. Initially this seems like quite a significant flaw, after all we don't think of factoring as a difficult task, but it is harder the larger a number you try to factor, and we will explore this technical challenge in a later section. For now we have another problem that Alex and Betsy face, how do they know they are really talking to each other? This is the problem of the digital signature and leads us into the next topic.

Elliptic Curves and Digital Signatures

The idea of a digital signature is exactly the same as a real signature, it is a mark on a document designed to ensure the identity of the signer. Much like their real life counterparts, real digital signatures have curves. Around the mid 1980's the idea of using elliptic curves in cryptography started to become prevalent. An elliptic curve is a simple mathematical construct and generally is of the form $y^2=x^3+ax+b$ where a and b are constants. Elliptic curves have an interesting property, if you take any two points on an elliptic curve and draw a line through them they will always intersect the curve in exactly 1 more place if you include sometimes missing completely as one more point. That sounds completely useless, but as it turns out it does have the handy property that it is a group. Group addition in this instance is drawing a line between two positive points, finding the location where they intersect the line and flipping it to the positive axis, it behaves like any other group. To add a point to itself you need only draw a line passing through the point tangent to the curve. Because of these properties this funny elliptic curve can have both addition AND multiplication by a scalar value, meaning in addition to being a group it is also a field.

So now we have this funny field that acts in a bit of a strange way. We are almost back to talking about cryptography, but first we need to make a pitstop into modular arithmetic on elliptic curves. So when you are adding two points on an elliptic curve you could take the long route and find out where it intersects by drawing the curve and making a line and finding the intersection point, Or you can solve for what that intersection point is in general and get a formula for it.

In the end you get the following equation for adding two points A = (Ax,Ay) and B = (Bx,By)

$L = (Ay-By)/(Ax - Bx)$

$Cx = ((L^2) - Ax - Bx)$

$Cy = (L*(Ax - Cx) - Ay)$

It should be noted that if we are using the same point for *A* and *B*, we can replace *L* with this

$L = (3 * Ax^2 + a)/(2 * Ay)$

Where *a* is the same *a* from when we first described the elliptic curve. These formulas can then be used in modular arithmetic to finally get back to digital signatures. For any elliptic curve *E* we can construct a different curve in a prime modular base to get a different curve, generally referred to as *#E* and it is the object $y^2 = x^3 + ax + b \bmod n$ for some prime number n. With that out of the way, now we can get back to signature algorithms.

Here is how to use the algorithm. To start let us say that a person, call them Allison, is sending message *M* to another person, call them Bob. Once again the message in this case is a specific number that will be encoded to have meaning later. It is important to note that it doesn't matter if *M* is already encrypted already or not, just that both parties agree on if *M* is before or after encryption. First Allison generates her public key, this is a few different parts. She needs to generate a curve with a large prime modulus *#E*. She will also need a point on the curve *G* that will act as a generator for a group. This point *G* should have prime order, and we will call the order of *G* by the variable *n*. She will need to generate a private key *d* which is a random number in the range [1,n-1]. The last part of her public key is $Q = d * G$ and this is using the funny elliptic curve multiplication we talked about earlier, in *#E.* Ideally, Allison already created and published this public key a while ago so Bob knows that is hers. When Allison sends a message to Bob she takes her message and passes it through a hash function, like sha-256 for example, and then uses only a few bits from that hash's output. This creates a new number *e*. She now creates another random number *k* in the range[1,order of (G)-1] not equal to *d* and from that gets her the first part of the signature

$r = x\ value\ of\ (k*G)$

again using elliptic curve multiplication. The second part comes from

$s = k^{-1} * (e + r*d)\ mod\ n$

After all of that Bob then receives the following from Allison's public key: *#E,G,Q,n* and from her message Bob receives *M,r,s*. Here is how he verifies that the message he received was signed by the person who has access to Allison's key (Ideally Allison). First Bob Hashes the message with the same algorithm Allison uses, this will get Bob the exact same *e* that Allison used. From there bob calculates the following

$w = s^{-1}\ mod\ n$

$u = e*w\ mod\ n$

$v = r*w\ mod\ n$

$x = x\ value\ of\ (u*G+v*Q)$ using elliptic curve addition and multiplication

if $r == x\ mod\ n$ then the signature is from Allison.

Briefly here is how it works

$r = kG,\ s = k^{-1} * (e + r*d)\ mod\ n$

That means $w = s^{-1} = (k^{-1} * (e + r*d))^{-1} = k * (e + r*d))^{-1}$

$u = e*w = e* k * (e + r*d))^{-1}$

$v = r * w = r* k * (e + r*d))^{-1}$

$x = x\ value\ of\ (e* k * (e + r*d))^{-1} * G + r* k * (e + r*d))^{-1} * Q)$

$x = x\ value\ of\ (e* k * (e + r*d))^{-1} * G + r* k * (e + r*d))^{-1} * G * d)$

$x = x \text{ value of } (k *G * (e + r*d))^{-1}*(e+r*d))$

$x = x \text{ value of } (k*G) = r$

Factoring Methods

So we have a method for two people to communicate with each other with secure means and we let them be certain that they are talking to the correct person. In this next section I will be going over how to break this encryption by attacking what should be it’s weakest point. This is assuming someone has complete and total access to whatever channel is being used to communicate. The easiest way to break the encryption is to start factoring.

Recall from earlier when I mentioned that someone could in theory break RSA encryption by factoring. The key is the fact that the public key *n* is a composite of two prime numbers and the private key *d* is derived from those two numbers. If someone was able to find out even one of those primes, they could get the private key and therefore decrypt the message and read everything they are saying.

In general we don’t think of factoring as being too difficult. Even the concept is something that 5th graders understand. Most people can think of a simple factoring algorithm just from their heads, try everything. This brute force attempt is easy to describe, start with a variable *i=2*, if *n mod i == 0* then *i* is a factor, otherwise increment *i* and try again. This has a running time of O(sqrt(n)). Out of context, that sounds pretty good, it's actually better than linear running time. Now for composites of small primes, this actually works fine. But we aren’t dealing with small primes, and after about 15 to 20 digits, this algorithm takes far too long. We need something faster.

Our next attempt is a birthday attack. Back in 1975 a man by the name of J. Pollard introduced a birthday-attack style algorithm to attempt to factor faster. This is Pollard's Rho and here is how to use it. Start with a psudo-random number generator f, for example $f(x) = x^2+c \bmod n$, c is a fixed constant and may start at 1. Now we get into the algorithm. Let $x = y = 2$ At the start. Now set $x = f(x)$ and $y = f(f(y))$. Lastly let $p = GCD(|y-x|,n)$. If p isn't n or 1, then p is a non-trivial factor of n. If it is n or 1 then set $x = f(x)$ and $y=f(f(y))$ again and repeat. If p is zero, then you have some problems and should change c and try again (maybe increment it).

Pollards Rho might look like it's just randomly picking two numbers and checking if their difference is the number we are looking for. However Pollard's Rho is a bit smarter than that. You see, the function $f(x)$ in mod p (let that be the smallest factor of n) must repeat on itself eventually, meaning that for any integer x $f(f(f(...f(x) = x$ for some number of f's. We usually denote multiple function calls like this by saying $f^{(i)}(x)$ so $f^{(3)}(x) = f(f(f(x)))$. Furthermore, if f is random we expect it will loop on itself after about $sqrt(p)$ steps. More importantly this means that we would expect that $f^{(i)}(x) = f^{(2i)}(x) \bmod p$ at least somewhat often. The exact reason for this is a bit more complicated than this paper will go into but what this means is that at some point we find that $f^{(i)}(x) = r + k * p$ for some integer k, and $f^{(2i)}(x) = r + l * p$ for some integer l. Putting this together that means that $|f^{(i)}(x) - f^{(2i)}(x)| = v * p$ for some integer v. Now that doesn't look super helpful but it does mean that we can use the GCD algorithm to calculate $GCD(|f^{(i)}(x) - f^{(2i)}(x)|,n)$. If $f^{(i)}(x) = f^{(2i)}(x) \bmod p$ as we expect to happen at least sometimes then the equation can turn into $GCD(|v * p|,n)$. Since we know p is a common factor, we will get it as an output and therefore have successfully factored n.

Now while this is a random algorithm, it takes a random amount of time to complete. However from my general testing with it, it can factor quite large numbers if given enough time.In theory it's running time is around O(sqrt(sqrt(n)*ln(sqrt(n))). This is pretty good, but we can do better.

The Lenstra Elliptic Curve Method is the fastest factoring algorithm that will be discussed in this paper and the third fastest publicly available factoring algorithm. The other two are only slight improvements on the Lenstra's ECM. As the name suggests, the Lenstra's ECM uses elliptic curves to factor large numbers. Recall from earlier when I said that an elliptic curve acts like a group even when under modular arithmetic of a prime number, well it turns out that this is not true for a composite number, and if you are familiar with modular arithmetic and look at the equations used before, it should be clear why. The problem is at some points we take modular inverses. This is fine in a prime modulus since all numbers have inverses, however in a composite modulus any number that is a factor will not have a modular inverse and will break our algorithm. This is where Lenstra's ECM shines. Because while doing arithmetic in an elliptic curve with a composite modular inverse, it is surprisingly likely you will run into a situation where you will break the algorithm, but when the algorithm does break, you have found the factor you were searching for. Here is how to use the Lenstra's ECM. For a composite of two primes n, Start with a random curve. In order to easily find a point on the curve, we generate the point first $P=(Px,Py)$ and one part of the elliptic curve a. Then we can calculate $b = Py^2-aPx^3-aPx$ all these variables are mod n. So now our curve is $y^3=x^3+ax+b$ and a point on the curve is P. Now we are looking for a time when addition and multiplication break, so we calculate *2!*P,3!*P,4!*P...For some upper bound*. This is particularly easy because all we need to do is keep multiplying *P* by larger numbers. If we reach the upper bound, we can try a larger bound or a different curve. Eventually something will break when we are taking the modular inverse. This would happen when *GCD(g,n)!= 1*. If we found this and the GCD isn't *n*, then we found the factor, if it is *n* then we can try a different curve.

Once again this is a random algorithm so there is no real way to know how long it will take, however its running time is approximated to be $O(e^{(sqrt(lg(n)*lg(lg(n))))})$. It isn't obvious at this point but in the long run this is faster than Pollards at particularly large numbers. From my testing I have found that in general Lenstra's ECM was slower than Pollard's Rho for composites of primes less than 14 digits, although just barely. I found that rarely, Lenstra ECM finds factors extremely quickly, far faster than it should. I also found that Lenstra's ECM is significantly faster than Pollard's Rho for larger products of primes. The largest example I had tested was a product of two 18 digit prime numbers, Pollard's Rho took more than 20 times longer than Lenstra's ECM to factor. I had tested even larger primes, such as those in the 22-26 digit range, and while Lenstra's ECM was able to factor after a generous amount of time (usually in the 1-4 thousand second range) Pollard's Rho never completed them in the time I had allotted them to factor. In the grand scheme of things this is quite fast, but how does it stack up to modern RSA? The short answer is, not well.

These days RSA uses at least 2048^(Citation Needed)bits for their keys. This means the product of the two keys took 2048 bits to store. This is a number that is at most 2^{2048}, or about 600 digits long. Our measly 40 digit numbers are an amazing feat, but they are nothing compared to the monstrous key size of modern RSA. Even if we had computers that were thousands of times faster than what we have today, it would still take millions of years to break RSA-2048.

CITATIONS:

Crandall, R. E., & Pomerance, C. (2010). Prime numbers: a computational perspective. New York: Springer.

Hoffstein, J., Pipher, J., & Silverman, J. H. (2008). Introduction to mathematical cryptography. New York, NY: Springer.

CODE:

All of the code for this project can be found on my github at this address

https://github.com/Doug-Richardson/Combined-Cryptography-

This includes a library created that has many of the methods described in this paper. It includes all the necessary information to encrypt with RSA, sign with ECDSA, and attempt to factor with Lenstra’s ECM and Pollard’s Rho all in one library. There is also some example code for how the methods are used.