Modern Cryptography

By: Isaac Schmidt Jonsen

In a time where governments and private companies are becoming increasingly interested in our data, it is more important than ever to feel that our information is truly secure. It is no longer outlandish to assume that any given communication channel is insecure and that the data we transmit is not always being used in ways we might want. To combat this, it is imperative that we have ways to transmit data that an eavesdropper would consider unintelligible, but the intended recipient could decode and recieve the right data. Modern cryptography is an ever evolving field of study that predates the internet itself and has produced innumerable encryption schemes that can achieve this desired secrecy. It is obviously impossible to cover every encryption in use today but the following sections will go over the most important and commonly used encryptions as well as the math that drives them. This will include RSA, AES, ECC, and the many vulnerabilities that come with them.

RSA and Number theory

The first major Encryption I will talk about is RSA. First published in 1977, RSA takes its name from the people behind the algorithm, Ron Rivest, Adi Shamir, and Leonard Adleman. RSA is known for being one of the first widely used public key encryption schemes. RSA is relatively slow compared to its more recently developed algorithms and as such has fallen out of favor especially pertaining to mobile computing and less powerful processors. However, Due to its relative simplicity in implementation and refusal on the part of many programmers to update security, RSA continues to be relied on quite heavily.

RSA works based on a few important results of number theory and the property of primes. It is crucial to understand these results if one wants to understand how RSA works. The first, and possibly most important theorem, is Euler's theorem. Euler's theorem states that for some $x, n \in \mathbb{Z}$ and x, n relatively prime:

$$x^{\varphi(n)} \equiv 1 \mod n \tag{1}$$

where $\varphi(n)$ is the Euler totient function (sometimes called Euler Phi function) defined as

$$\varphi(n) = |\{a \in \mathbb{N} \mid 1 < a < n, \gcd(n, a) = 1\}| \tag{2}$$

Euler's theorem follows from Lagrange's theorem and other features of finite groups. One important note on the totient function is that for any p, q prime:

$$\varphi(pq) = (p-1)(q-1) \tag{3}$$

If we then use this fact in tandem with Euler's theorem we see that

$$x^{\varphi(pq)} \equiv 1 \mod (p-1)(q-1) \tag{4}$$

RSA Algorithm

For two people, Bob and Alice, to communicate secretly over an insecure channel, Alice will calculate a private key for herself and release a public key for Bob to encrypt his data. This type of setup is the basis for Public-key Cryptography. Bob can then use the public key to encrypt his data in a way that it is infeasible[1] for some eavesdropper to decrypt the data given only what is sent over the channel. For RSA, this process starts with Alice choosing two sufficiently random distinct prime numbers p, q often between $2^{(2^9)}$ and $2^{(2^{10})}$. She then multiplies p and q to give the RSA modulus n. This modulus is released while its factors are kept private. The next step requires Alice to choose some (not necessarily random) $e > 2$ called the encryption exponent and calculates the decryption exponent d such that:

$$ed \equiv 1 \mod (p-1)(q-1) \tag{5}$$

Note that the reader should recognize the modulus as being the totient function of n. Finally, Alice releases the encryption exponent e, giving Bob all that he needs to encrypt his information. Bob will then use some agreed upon method to reversibly encode his data as some integer x strictly less than n. To encrypt his data, Bob simply raises his integer to the encryption exponent and mods out the RSA modulus giving him some encrypted data c or:

$$c = x^e \mod n \tag{6}$$

This encrypted data can be sent over the channel and Alice can decrypt it by doing the same thing but using the decryption exponent d instead of e or:

$$x = c^d \mod n \tag{7}$$

Alice then uses the encoding scheme agreed upon to recover Bob's data.
It may not be obvious that this works but we can make this more concrete by combining the encryption and decryption steps while also using the definition of modulus to write:

$$ed = N(p-1)(q-1) + 1 \tag{8}$$

for some $N \in \mathbb{N}$ and:

$$\begin{aligned} (x^e)^d \mod n &= x^{N(p-1)(q-1)+1} \mod n \\ &= x \cdot (x^{(\varphi(n)})^N \mod n = x \mod n \end{aligned} \tag{9}$$

RSA is effectively secure due to the time it would take to factor n. The fact that all known factoring algorithms are non polynomial time and that the set

[1]The security of RSA depends heavily on its implementation. Given how long RSA has been around, there are countless known attacks and ways to disrupt this encryption. If implemented by a professional, RSA is theoretically very secure. However, the following is simply a layout of how the encryption works and should NOT be used as a guide for writing your own code.

of primes is infinite ensures that regardless of classical computing speed, we can always use a large enough p and q to make RSA secure.

RSA Security

To this day no one has publicly[2] broken the math behind RSA when using at least a 1024-bit key. However, there are countless known attacks that target the implementation itself as well as poor choices of variables. I will briefly go over a few of the more important ones.

One of the first major attacks that caused a lot of trouble in the early days of RSA is the common modulus attack. When RSA was first announced many security experts tried to simplify RSA and reduce its runtime. One way that many people tried to do this was to use the same RSA modulus for multiple people while distributing a unique encryption exponent to each person. Given two encrypted messages and separate encryption exponents it is possible for an attacker to factor the RSA modulus and decrypt either message. This attack can be easily avoided by using separate moduli for each sender. Another popular attack is the side-channel timing attack. This attack works by timing how long each step of the encryption takes. This requires incredibly precise timing and known timings on each operation from that chip. This attack can be avoided by ensuring that every process takes the same amount of time. Doing this requires that every process is set to the worst case scenario and often increases the time it takes to encrypt the message.

AES

The next major encryption scheme that is seen everywhere in the cryptographic world is AES. The Advanced Encryption Standard (AES) was the culmination of a selection process first introduced in 1997 by the National Institute of Standards and Technology (NIST). There was a need at the time for an encryption scheme with variable key sizes that was fast and unbreakable. Belgian cryptographers, Vincent Rijmen and Joan Daemen submitted what they called Rijndael in 1998. In 2002, Rijndael was chosen as the new encryption standard and adopted by most U.S. government agencies. Rijndael and AES are mostly synonymous and the names are used interchangeably.

AES is a symmetric key block cipher that uses key sizes of 128, 192, or 256 bits where increased key size corresponds to increased security. AES is a relatively complex algorithm in that it is almost impossible to follow any given bit through the whole process. Much like throwing fruit in a blender, I can de-

[2]It appears to be generally agreed within the cryptographic community that The NSA has broken RSA with a 1024-bit key, but this is unconfirmed and highly speculative

scribe the machinery behind AES but keeping track of every element as it gets encrypted is an herculean task. The flip side of this is that the math behind it is fairly straightforward and requires less math background.

AES algorithm

The algorithm starts with the user inputting some plaintext and choosing a key and key size. AES takes the plaintext and splits it up into 128-bit "blocks" and arranges these bits into a 4x4 matrix of bytes. The key is then expanded using the AES key schedule. The algorithm then applies four operations called a round. For a 128-bit key 10 rounds are applied, 192 applies 12 rounds, and 256 gives 16 rounds. Each block is bitwise XOR'd against the initial key. The resulting block then has the rounds applied to it.

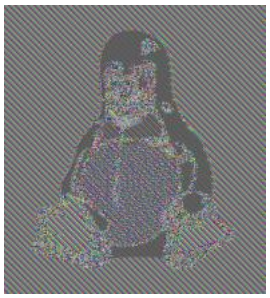
The first operation of the round is done by substituting every element according to a lookup table. There is a standard lookup table used for all AES encryption. Then the rows are shifted with the elements wrapping around based on the row index. In other words, the elements in the first row are unchanged, the elements in the second are shifted one to the left, the third row shifted two to the left, and the last row is shifted three to the left. The third operation is a matrix transformation on each column. This transformation is a left multiplication by a linear transform matrix done over the Galois field $GF(2^8)$ with field polynomial $x^8+x^4+x^3+x+1$. Like the lookup table, the transformation matrix is standard and can be found online. The round is then finished by applying the round key given from the key schedule. These four operations are applied 9, 11, or 13 more times based on the key size. The algorithm finishes by applying one more round without the column transformation and decomposes the blocks into a ciphertext.

Other important AES notes

One important note on AES (and all block cyphers) is the mode of operation. The mode of operation determines how the plaintext is split up into blocks and then turned back into plaintext. Although AES is technically secure, this security only refers to each block. The naive but natural way to do this would be to keep the blocks in order without any rearrangement. This is called the Electronic Codebook (ECB) mode. The issue with ECB is that, since AES encrypts every block in the same way, ECB preserves patterns. The image often used to show what this looks like is the Linux penguin shown in Figure 1.

Another very important characteristic of AES is its speed. Because of AES being set as the encryption standard, most computer chips have built in support for AES rounds. This means that AES can encrypt Gigabytes of data almost instantly. With mobile computing becoming more and more prevalent, the speed of an encryption scheme is incredibly important. The importance of speed in

(a) Original



(b) Image after AES using ECB

Figure 1

an algorithm leads right into the next encryption scheme.

ECC

Elliptic curve cryptography is one response to our increasing need for efficient and secure encryption. First introduced in 1985, ECC has found popularity more recently in use by may widely used cryptocurrencies such as Bitcoin and Etherium. While most cryptocurrencies use it as a digital signature algorithm, ECC actually refers to all cryptographic algorithms that use elliptic curves. The following section will focus on the steps shared throughout all implementations of ECC. Similar to RSA, ECC encryption is a public key encryption. When you compare the two algorithms, the biggest difference is the key sizes and speed. The security associated with a 2048 bit RSA key is roughly equivalent to a 256 bit ECC key. When comparing the runtimes of the two algorithms using the above key sizes we see that ECC is about 20x faster than RSA.

Before moving forward it is important to know that ECC requires a more extensive mathematics background to understand than most other popular encryptions. Elliptic curve cryptography relies heavily on group theory and abstract algebra as a whole. For any reader without a mathematics background I would suggest reading at least introductory material on group theory.

ECC Algorithm

With that in mind let us define an Elliptic curve. Elliptic curves are defined independent of cryptography and have seen extensive research within the field of algebraic curves. An elliptic curve is any curve that can be written as:

$$y^2 = x^3 + bx + c \tag{10}$$

Where b and c are some constants over the desired field. In the context of cryptography, this equation is restricted to a finite field. To be concrete we will

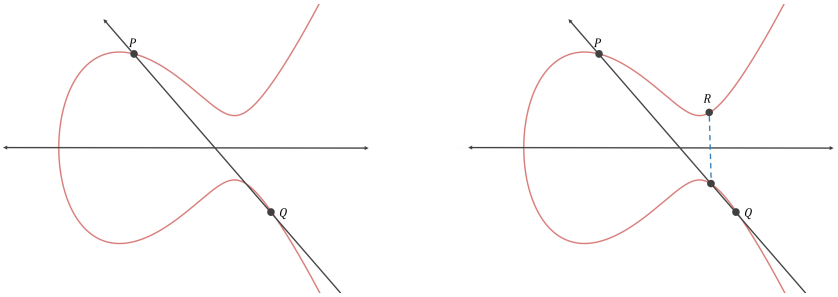
work with the curve used in Bitcoin called Secp256k1. Secp256k1 uses equation:

$$y^2 = x^3 + 7 \tag{11}$$

over the $\mathbb{Z}/p$ with:

$$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 \tag{12}$$

We then want to create a group using the elements in $mathbbZ/p$ that lie on this curve. To do this, we endow an operation often denoted "dot" onto this set of points. For two points P and Q, we define P dot Q as drawing a line through the two points, finding a third point where the line again intersects the curve, and taking that point reflected about the x-axis. This is slightly easier to visualize if we do this with a curve over the real numbers rather than a finite field. Figure 2 below shows this process over the real numbers.



(a) Find the line intersecting P and Q (b) Reflect the third point about the x-axis

Figure 2

In addition to this operation, we will define an identity element as the point at infinity. This point at infinity is hard to visualize but just imagine it as creating a vertical line when "dotted" with any element. The definition of this infinity element as well as the proof that the operation is well defined is quite rigorous and left as further research to the reader. For cryptographic purposes we will use some subgroup generated by a point on the curve. To do this we start with the generator, often specified in the definition of the curve, and "dot" it with itself. The resultant line is defined to be the line tangent to this generator point and will intersect exactly one other point. For the generator point P we denote P dot P as $2P$. From the abelian group structure we know that the operation can be done in a similar way to fast exponentiation by using square and multiply. So for example:

$$8P = 2(2(2P)) \tag{13}$$

This allows us to rapidly calculate relatively large multiples of P. The ECC algorithm with some random number N (often a 256-bit integer) will calculate the point:

$$X = NP \tag{14}$$

This will give a private key N and a public key X. From here there are many different algorithms that use these keys in different ways. For the sake of understanding elliptic curves these following steps are mostly unimportant as they work quite similar to other algorithms with the same purpose.

ECC security

Similar to the above encryption schemes, ECC is theoretically secure if implemented properly. Proper implementation, however, is notably harder than said other schemes. When it comes to elliptic curves there are more variables to keep track of and the math alone requires much more expertise than most programmers possess. Every step of the algorithm, including the choice of curve, is crucial to the security of the system. Because many security experts refuse to trust aspects of their code that they did not make, tons of ECC implementations contain garbage curves with random generators over poorly chosen fields. These errors on the part of complacent programmers have led to ECC being branded as too difficult to use and insecure in nature. The fact is that it is perfectly acceptable to use well established curves. In fact the only thing that someone planning on using ECC should customize is the key and even that should be done entirely by a random number generator. The difficulty of finding the key given only the generated point is proven monumentally difficult and the only known polynomial time method of doing this is by using quantum computers.

Quantum computing

When discussing cryptographic systems it is most natural to also analyze the vulnerabilities in these systems. It may not always be feasible to exploit these vulnerabilities, but understanding them is key. One such vulnerability in almost all modern systems is the advent of quantum computing. With quantum computing, the assumption that prime factorization and discrete logarithms are sufficiently difficult to compute no longer holds. There exists theory and algorithms that prove these calculations to be almost trivial with quantum computing. In this section I will discuss briefly the mechanics used in a quantum computer, some of the more important quantum algorithms, the implications of these algorithms on modern cryptography, and the difficulties of implementing quantum algorithms. For the general population it is certainly infeasible to utilize quantum computing and it is unclear whether it is even possible. For now we will work under the assumption that a quantum computer with a sufficient number of "qubits" is possible to build and operate without issue.

Quantum Mechanics

The field of quantum mechanics is vast and far from being completely understood. Although the concepts can be very difficult to fully grasp, we need only know some of the concepts and understand the math that describes them. I am far from an expert in the field and this is not a paper on quantum mechanics, thus I will keep it brief and try not to anger any physicists. By far the most

important quantum mechanical concept, as it pertains to quantum computing, is the idea of quantum superposition. Quantum superposition states that a quantum object (eg. Electron) can be in multiple quantum states at once and combining two or more together will provide a new valid quantum state. This is most famously described by the thought experiment of "Schrodinger's cat". In a more concrete sense, an electron can be seen as having an "up" spin, a "down" spin, or both. When we observe this electron however, it must take on only one state. For this reason, we use notation that gives a probability of this object's spin at any given time. From now on we will refer to this quantum object as a "qubit" as it is analogous to a classical computing bit. We can write any given qubit x as a combination of spin vectors and the probability amplitudes:

$$x = \alpha|1\rangle + \beta|0\rangle \tag{15}$$

Where $|1\rangle$ corresponds to the Bra-Ket notation for down-spin, and $|0\rangle$ up-spin, and $\alpha, \beta \in \mathbb{C}$ s.t. α^2 is the probability of down-spin, β^2 is the probability of up-spin, and $\alpha^2 + \beta^2 = 1$. We can make this even more simple by writing x as a single vector $x = (\alpha, \beta)$ If we now combine two qubits x and y with unknown spin, we have four equally likely quantum states. Namely: (down,down),(down,up),(up,down), and (up,up) written mathematically as:

$$x \otimes y = \alpha_1\beta_1|11\rangle + \alpha_1\beta_0|10\rangle + \alpha_0\beta_1|01\rangle + \alpha_0\beta_0|00\rangle \tag{16}$$

Because these are all entirely valid states, we can (crude but functionally) consider these to be independent objects that can be acted on somewhat separately[3]. Therefore, given n number of qubits, you have, at the very worst, 2^n quantum states with values in $[0, 1]$. From this it starts to become clear why quantum computers could complete certain tasks with ease. The goal is now to write an algorithm to "influence" each of these states in a way that when the algorithm is finished, we can observe the qubits and the corresponding quantum state gives us a solution with high probability.

Grover's algorithm and quantum gates

One of the first and most important algorithms to do this is called Grover's algorithm. Developed in 1996 by Lov Grover, Grover's algorithm is an incredibly powerful and one of the more palatable algorithms out there. To understand any quantum algorithm we must first nail down a definition for how we can influence these quantum states. This can be done using quantum logic gates. Similar to logic gates in classic computers, quantum logic gates take in one or more inputs and provide some type of output. For example, one of the more famous quantum gates is called the Pauli X-gate. The Pauli X-gate takes in a single qubit

[3]I say "somewhat" because these states cannot be acted upon directly. Mathematically however, they take on different values and can be thought of as separate entities

and flips its probabilities. So if you feed a qubit with probability of down-spin 80 % through a Pauli X-gate, it will now have a probability of down-spin 20%. Another important gate is the Hadamard gate. This gate takes any qubit and resets it after detection. Meaning, if we initialize a qubit and know its spin, we send it through the Hadamard gate and it will come out with equal down and up spin probabilities. when it comes to sending multiple qubits through a single gate we deal with multi qubit gates. Multi qubit gates take the superposition amplitudes as input rather than the individual probability amplitudes for each qubit. More specifically, for a 2 qubit gate, we use the vector that represents each four quantum states as described equation 2 with $(\alpha_1\beta_1, \alpha_1\beta_0, \alpha_0\beta_1, \alpha_0\beta_0)$. When it comes to what these gates look like and how they are physically implemented there is no one agreed upon method. There are methods that alter the magnetic field around a qubit and others that use the polarity of light to "split" the spin similar to pointing a laser at a pane of glass. We will once again make an important assumption about quantum computers. We assume that these gates are possible to implement, work perfectly every time, and we can build a computer with sufficiently many of them. It is natural to represent these gates as transform matrices on n-dimensional vectors. In practice, many of the gates we use to describe an algorithm are not possible with a single physical gate. To achieve the desired result, a combination of many "elementary" gates are used.

With that out of the way we can discuss Grover's algorithm. The goal of Grover's algorithm is to search any unordered list. Grover's algorithm utilizes the quantum superposition of states to check many different items against some Oracle function simultaneously. In classical computing, one would have to check each item in the list one-by-one. From this simple explanation it becomes clear that Grover's algorithm is potentially significantly faster than classical computing and could theoretically be used for prime factorization. As we will see however, Shor's algorithm is significantly faster at prime factorization.

To start we define the oracle function as a function with input x against some $x*$ that you are looking for where:

$$f(x) = \begin{cases} 1 & x = x* \\ 0 & x \neq x \end{cases} \tag{17}$$

This oracle function is then used to form a unitary operator that inverts the amplitude if $f(x) = 1$ and does nothing if $f(x) = 0$. Lastly we define the Grover operator that finds a mean amplitude and flips all amplitudes about the mean. Given that all states are initialized to be equal probability, once the desired amplitude is inverted, the mean drops significantly. In the end the desired quantum state is observed with a very high probability as is shown in Figure 1. Physically, Grover's algorithm requires clever implementations of limited quantum gates to achieve the desired results. To show how to implement Grover's algorithm I will briefly go over one designed by Coles, Eidenbenz, et. al for IBM's

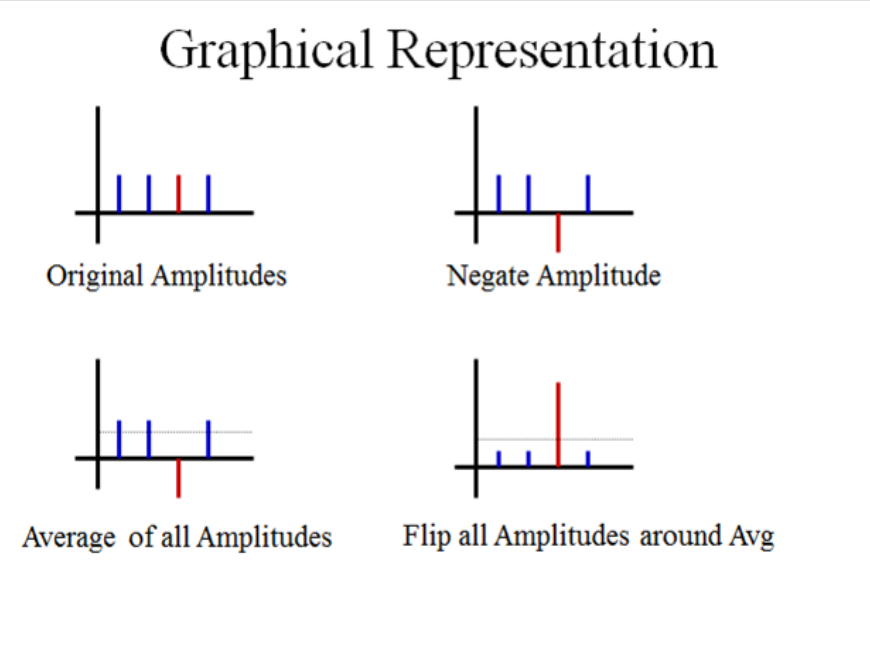


Figure 3: relative probability amplitudes at each step

5 qubit quantum computer. I would like to reiterate that the reader should not be thinking about how each of these gates might look in a real quantum computer. Quantum gates are a continued area of research and, although there are interesting developments, they are way beyond the scope of this paper. For our purposes, one must only understand how the math behind each gate works, and that these gate can be implemented with a high degree of consistency.
This implementation of Grover's algorithm aims to find $x^* = [1, 1]$ in a list containing the 4 binary strings of length 2. While this operation seems beyond trivial it is important to note that it can be extended to much larger lists and that the binary string would normally be representative of some more complex criteria. To achieve the search this algorithm uses five different quantum gates in varying amounts. These five in matrix representation are:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, H = 1/\sqrt{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix},$$

and $CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

The gates are laid out as shown in Fig. 2. The first two steps are used to initialize the qubits and put them in a uniform superposition. The following 13

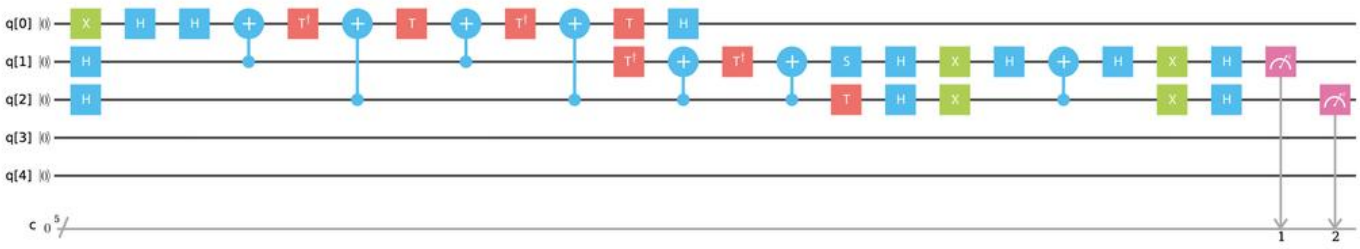


Figure 4: Quantum circuit diagram for Grover's algorithm

steps perform the oracle operation or what is equivalently known as a Toffoli gate. The Toffoli gate can be described as a 3 qubit gate that flips the last qubit if the other 2 are down spin. The matrix representation is then:

$$\text{Toffoli}= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The following 7 steps then perform the flipping about the mean operation. At this point I believe that the best way to understand this circuit is to simply go through the math on paper. If you do this on paper, and keep track of how the probability amplitudes are affected by each gate, it quickly becomes clear how Grover's algorithm works.

Shor's algorithm

Shor's algorithm is arguably the most important quantum algorithm that applies to cryptography. As stated previously, Shor's algorithm takes any number n with prime factors p and q and gives you p and q. Shor's algorithm utilizes the quantum Fourier transform to find the period of some element $x \ mod \ n$ where $gcd(x, n) = 1$. We know that there is a period $x^\sigma \equiv 1 \ mod \ n$ then $x^\sigma - 1 \ mod \ n \equiv 0$ this factors out to $(x^{\sigma/2} - 1)(x^{\sigma/2} + 1) \ mod \ n \equiv 0$ if we then let $P = x^{\sigma/2} - 1$ and $Q = x^{\sigma/2} + 1$ then $p = gcd(P, n), q = gcd(Q, n)$. The quantum Fourier transform F is defined as the NxN matrix with elements $F_{ij} = \omega^{ij}$ where ω is the Nth root of unity with $\omega^N = 1$. This is then a matrix transformation on some input vector X. If we then feed in a vector $X = (1, x^1, x^2 ... x^n)$ The nature of the quantum Fourier transform provides complete destructive interference for any power of x s.t. $x^i \neq 0 \ mod \ n$ so the final result will give roughly equal probability of all multiples of the period σ. Shor's algorithm is

much harder to understand and there is no real visual to help like there is in Grover's algorithm. Due to how quantum algorithms work, we can't really draw parallels between algorithms to try to understand them. However, we can say that similar to Grover's algorithm, Shor's algorithm uses a clever series of gates to achieve the desired result. The most important thing to understand is that Shor's algorithm has the potential to break RSA with relative ease.

Limitations

So far we have been operating under the assumption that Quantum computing is feasible. Arguing about the feasibility in the future is a somewhat philosophical debate but for the time being we can at least discuss the concrete limitations that are holding us back. The most important limitation is creating a quantum computer with a sufficiently large number of qubits. The largest quantum computer to even be theorized as of writing this is Google's 1000 qubit model. For reference, a paper was released in early december of 2019 that described how we might be able to break 2048 bit RSA encryption in 8 hours using 20 million qubits. This paper was revolutionary in that it took quantum noise into account and gave a concrete implementation of Shor's algorithm for modern RSA. It also highlights just how ridiculously far from actual implementation we are given that we have to increase the size of our quantum computers by 19,999,000 qubits. With even the most optimistic estimates, we are at least 15 years from reaching a 20 million qubit computer.

Another major limitation is what is referred to as the "life" of the qubit. The life of the qubit refers to how long we can manipulate a qubit before it loses its important quantum property. It is observed that if an algorithm goes on for long enough, a qubit can never truly be in equal probability superposition and will start to act like a classical bit. The longest "life" that has been achieved is around .1 seconds. When compared to an algorithm that takes 8 hours to run we are again way short of the finish line.

The last major limitation is positioning of quantum gates and the amount of power they draw. When placed too close to each other and in large enough numbers, the quantum gates start to interfere. This manifests more significantly when certain gates are placed in certain positions. This makes it almost impossible to use certain gates in tandem requiring clever positioning and often limits which gates can be used. We also have to consider the amount of electricity required to run all of these gates. The 1000 qubit machine from Google also requires near 0 kelvin temps to run. All this means that their machine will use just under 25 kilowatts which is an outlandish amount of power for a single machine. These limitations suggest that although quantum computing has the potential to break most modern crypto systems, we have time to implement post quantum algorithms to combat it.