Period Finding on a Quantum Computer using Shor's Algorithm

by

Samuel Marcillo-Gomez
Spring 2019

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Cybersecurity
With a Minor Mathematics
in cursu honorum

Reviewed and approved by:

Dr. Alberto La Cava
Thesis Supervisor

Submitted to
the Honors Program, Saint Peter's University

26th of March, 2019

Abstract

In recent years, there have been numerous developments in quantum computation. These developments have brought into question, how quantum computers could affect security have risen. For instance, Shor's algorithm is believed to be able to break certain encryptions faster on a perfect quantum computer faster than on, what is known as, classical computers. In a few years or decades, there could be significant developments made that allow for quantum computers to perform Shor's Algorithm. As quantum computers exist now, the implementation of the algorithm is known to be difficult as the computes are very basic. Attempts to create quantum circuits that can compute Shor's Algorithms aid in the understanding of the algorithm.

Introduction

The most import basis in cyber security is encryption. It provides privacy, anonymity, authorization, authentication, integrity, and assurance. Encryption is the process of converting information into a code, or cipher text, by use of an encryption algorithm, or cipher. In order for the encoded information to be useful, encryption needs to be reversible, or to decrypt-able (Paar and Pelzl 3). This is achieved by securely generating keys that allow for the cipher text to be decoded back into plain text. There exist two types of cryptographic algorithms that are used today. These two types of algorithms are symmetric and asymmetric (Public-Key) cryptography. These names refer to the keys used in cryptosystems, for encryption and decryption (Paar and Pelzl 4). In symmetric-key cryptography both processes of encryption and decryption use the same keys. This can be analogized as mailbox shared between two people, both persons have access to the same mailbox because that have duplicate keys. The counter to this, is asymmetric-key cryptography, where the keys used for encryption are not the same keys used in decryption. This is similar to the relationship between a mail sender and a mailman; the sender has the key their mailbox and puts their mail in their mailbox (the mailbox in this instance is the encrypted message); the mailman has a different key (like a master key) which allows him to open (decrypt) the sender's mailbox. While both algorithms are important, they are used for different things: symmetric cryptography is used for data encryption and integrity check of messages while asymmetric is used for digital signatures and key establishment (Paar and Pelzl 3).

While it is clear that cryptography has made cyberspace safer, some of the algorithms used today are results of many before them that were broken. Examples of this are seen in symmetric cryptography with algorithms like DES and Triple DES. Both precede the symmetric cryptographic algorithm, AES and both were broken, or insecure. RSA, an asymmetric

cryptosystem, has withstood attempts to break it algorithm; however, Elliptic curve cryptography has been becoming more popular as the sizes of its keys are not required to be as big a RSAs. That being said, with the innovation of quantum technology, there is the chance that asymmetric cryptosystems, like RSA and ECC, could potentially become in secure in the future. This idea stems an algorithm created by, MIT Professor, Peter Shor, in which he showed that a quantum computer could potentially solve RSA Keys and ECC Keys in polynomial time, which is considered to be much faster than what can be done on a classical computer. Important departments of government like the NIST (National Institute of Standards and Technology) expressed concern in the rise of quantum technology and are in the process of finding a potential replacement for the RSA cryptosystem, in a Post Quantum World (National Institute of Standards and Technology, et al.).

This excitement over quantum computing, leads one to take into account how and why quantum computers can do what classical computers cannot. To understanding the problem at hand, the goal of this paper is to comprehend how Shor's Algorithm works, specifically with RSA, and to see if the current state of quantum computers is able to implement the algorithm.

Literature Review

To begin, a basic understanding of RSA cryptosystem would be helpful in the pursuit to understand Shor's algorithm. RSA (Named after the initial letters of the surnames of Ron Rivest, Adi Shamir, and Leonard Adleman) (Paar and Pelzl 173), is an asymmetric cryptosystem that uses specially generated keys to separately encrypt and decrypt messages. From the book, *Understanding Cryptography: A Textbook for Students and Practitioners,* the equation for the cryptosystem goes as followed:

m = message in plain text
c = ciphertext after encrypting message
Encryption: $c \equiv m^e \pmod n$
Decryption: $c^d \equiv (m^e)^d \equiv m \pmod n$

The above reflects that processes of encrypting (*e*) and decrypting (*d*) messages. For instance if *m* = 4, *e* = 19, *d* = 59, and *n* = 123. For encryption $4^{19} \equiv 31 \pmod{123}$, for decryption $31^{59} \equiv 4 \pmod{123}$. This process is due to the generation of the of the keys used in the algorithm:

$$n = pq$$

Let *p* and *q* be randomly selected prime numbers. Next is to calculate the product of *p - 1* and *q - 1*:

$$\Phi(n) = (p-1)(q-1)$$

After, select encryption key *e*, such that is 1 < *e* < $\Phi(n)$ and *e* is relatively prime with$\Phi(n)$, meaning the gcd($\Phi(n)$, *e*) = 1. Lastly compute the decryption key, *d*, which is the modular multiplicative inverse of $e\ (mod\ \Phi(n))$:

$$e \times d \equiv 1 \bmod \Phi(n)$$

If applied to the previous example:

$$p = 41,\ q = 3, \text{ then } n = pq = 123$$
$$\Phi(n) = (41-1)(3-1) = 80$$
$$1 < e < 80$$
$$e = 19$$
$$d \times 19 \equiv 1 \bmod 123$$
$$d = 59$$

Now it should be noted, as it is an important part to the algorithm, that is that certain variables need to be kept private in order for the algorithm to be secure. Those variables are *p*, *q*, and *d*; while, both *n* and *e* are “public” and are used by the sender to encrypt the recipient’s message. Once the message is encrypted, it can only be decrypted with the variable *d*, which is the main

reason it is kept private. While *d* is only known to the recipient, an attacker could incept the encrypted message and attempt to decrypt it by performing a brute force attack. This is the process in which randomly generated decryption keys are repeatedly used to attempt to break the encryption. Performing such an attack is not practical as real RSA keys are significantly bigger than the examples previously used. In practice, RSA generates large prime numbers to calculate large keys sizes in the range of 512 to 1024 bits, and sometime larger. These can look like the following:

p =
2194108636009792562194488148340957485418603885173794623837108374189052283398558650893431061822433246927737366622349314730803894844755301767403140459603156875815925530481423637856657325644381841263805509146538949069769241043588235986924218540327041718983
q =
3347683745143246015031545700658288936021733304889051896302506135487296093843710163650337155441766372724178244906318369202933557290528100741300815791794362122353550649791295869947359689037826142245634543919013742164126844872269027306704042925804397558736934487928381093996471437233654788769517907637
e =
3638579677719833158717494329237705180923909963814345568154145752548628655638571784684069671816064470045431851623155421922901820740766190565129078282982380910801115075607164899704858131016820812387848482558594744916085433533278134791367366076199478726997835833593602694436811660562038113885379510453890707137046149720098891687387266530685086480521139052049544825313685181165484810747929444905062344403010815443762732609397778016170131837335200768151172760370763108948184181774143285094652175248352993936304450903892712564242932010775697610835862171073
d =
2001364020115476102605593688257148646728375818888238866226191737892454326569294560487056329853988233964431389099190639876917699214354558447354924049987438489459074832301667201490351119936380012828421003379188392833749857497087395950339722681165744029957686214088416259517882099317780138488366635391129274570651847548688082193622289002515429440039995667104043455158996107424393548014081550867480615004783060111776829969367958460240075467542528887147957727892798491747801779957491391778930052525036330311917565656620341082683251291771024146884625531321
n =
734518181584840153993509248255050372663070729613573975865561260960375638070716055796553769443623839789770105475610829553992150719211937950656639639436231870670022284263614463015774376946759411526048256182621021542144185318856571466958

3101465801669632233328586195149346180213512346399930307600968477117824259154511999984026193466324438115195186728333885287964934653843927008899655934204250192060958743495905661403956629437418302116641458007417084647764907800351195572424579049286069701532189065080545199750031331036826444434914520915889885805903573171

Since n is such a large number, performing this kind of attack is computationally infeasible as almost all the values less than n would need to be checked. For this reason, mathematical attacks have been conceived to calculate the decryption key more efficiently than brute force attacks (Paar and Pelzl 194). In a Mathematical Attack, instead of guessing the decryption key d, the attacker attempts to factor the prime the values of n, p and q (Paar and Pelzl 194). If p and q are calculated, the attacker can then figure $\Phi(n)$ as, $\Phi(n) = (p-1)(q-1)$, and thus calculate the decryption key, d, as $d \times e \equiv 1 \bmod n$. However, again arises the problem: because n can be a very larger number, like the ones seen above, determining p and q would take a very long time to calculate (Mermin). Specifically, the most asymptotically efficient classical algorithm is the number theoretic sieve, which factors an integer n in time $O(\exp[(\log(n))^{1/3} (\log(\log(n)))^{2/3}])$. (Lomonaco Jr.). Shor's algorithm provides a solution to this problem. Below it are the steps of Shor's Algorithm (Lomonaco Jr.).

Step 1:

Choose a number $1 < a < n$ such that is relatively prime with n, meaning $\gcd(a, n) = 1$, where gcd denotes the greatest common divisor. If the $\gcd(a, n)$ does not equal 1 then k is a factor of n and the algorithm ends; otherwise, proceed to **Step 2.** (Lomonaco Jr.).

Step 2:

With the value a determine the period r of the equation a mod n such that $a^r \bmod n \equiv 1$ (Lomonaco, 2000).

Step 3:

If r is odd then repeat **Step 1**. If r is even proceed to **Step 3** (Lomonaco Jr.).

Step 4:

Since $a^r mod\ n \equiv 1$,

$$a^r - 1 \equiv 0 \text{ mod } n$$

Meaning $a^r - 1$ is a multiple of *n*, thus some there must exist some value integer *k*, such that

$$a^r - 1 = kn$$

Since *r* is even, the above equation can be rewritten as, (*k* is unnecessary for the rest of the equation) (Lomonaco, 2000).

$$(a^{r/2} - 1)(a^{r/2} + 1) = pq$$

Step 5:

Finally, since neither $(a^{r/2} - 1)$ nor $(a^{r/2} + 1)$ are congruent to 0 mod *n*, and $n\ = ((a^{r/2} - 1)/p) * ((a^{r/2} + 1)/q)$, then it can be said that, (Lomonaco, 2000).

$$p = \gcd(a^{r/2} - 1, n)$$
$$q = \gcd(a^{r/2} + 1, n)$$

Thus *p* and *q* have been determined. The most significant part of Shor's algorithm **Step 2** as it is the hardest part, for classical computers to do (Lomonaco, 2000). However, again, when *n* is a large number calculating its period would be as effective as a brute force attack. For this reason, Shor uses a quantum computer to calculate the period of *n*.

To elaborate, a quantum computer is computer that uses quantum phenomenon's to manipulate data. Classical computer or non-quantum computers (the ones used now), store and manipulate information using bits (IBM). These bits are designated binary values 0 and 1. Quantum computers use qubits (portmanteau of quantum bits) instead of bits, which are able to leverage different physical phenomenas, such as superposition, entanglement, and interference (IBM). It is these physical phenomenas that give quantum computers an advantage over classical computers. In the case of Shor's algorithm, he sought to use the superposition to speed up the time of the period finding function. Superposition is the refers to a combination of states we

would ordinarily describe independently (IBM). In a quantum computer, superposition would then mean that qubit would be a combination of the classical bit states 1 and 0. Using this phenomenon Shor's algorithm is able to factor an integer n that takes asymptotically $O(\log(n)^2 \log(\log(n)) \log(\log(\log(n))))$ steps which is polynomial time in the number of digits $O(\log(n))$ of n (Shor 15). Essentially, using the property of superposition would allow to multiple testing $a^r \text{mod } n \equiv 1$.

The quantum portion of Shor's algorithm goes as follows: create two quantum registers and set their sizes to $n^2 < q < 2n^2$, and n - 1 respectively; put register 1 in the uniform superposition of states representing numbers a (mod q) and load register 2 with all zeros; we compute $a^r mod\ n \equiv 1$ in the second register; perform an inverse Quantum Fourier Transform on the first register; lastly, measure the first register (Shor 16-17; Lomonaco). For a visual representation *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance*:

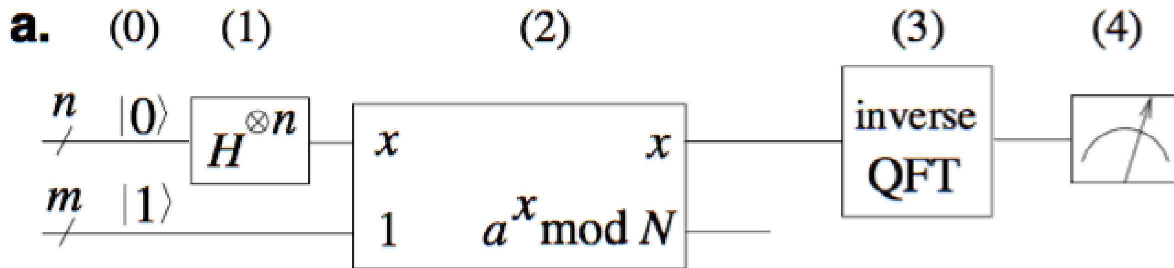

Figure 1: Visual of how the period finding function of Shor's Algorithm would be implemented on a quantum computer (Vandersypen, et al. 15)

The heart of Shor's algorithm is the Quantum Fourier Transform, it should use resonances to amplify the basis states associated with the correct period and the incorrect answers destructively interfere, which suppress their amplitudes, increasing the speed of algorithm to polynomial time (Shor 17).

Research Methods

In order to begin to determine how Shor's algorithm will affect RSA encryption, it would be beneficial to test current circuits that implement Shor's Algorithm on a quantum computer. For this testing process the web-based user interface, Jupyterlab, running Python Version 3.7.0, in conjunction with IBM's Qiskit API, will be used to create and simulate quantum circuits running on a quantum computer. These programs will be running on a 11-inch MacBook Air, with 8 Gigabytes of memory, running OS X El Capitan 10.11.6. Excel was used for data tables.

This research is to essentially examine the state of current state of quantum computing as well as the implementation of the algorithm. Simultaneously an examination of the algorithm on a classic computer will be conducted to potentially determine the speed difference on a quantum computer.

Research

To begin the research, conducting the period finding function on a classical computer should be used to compare to a quantum computer. To do this, a RSA program will be constructed. Below is the code used to construct RSA keys, which was ran on Python 3.7.0

```
import time
from random import randrange, randint
from math import gcd
from secrets import randbelow, SystemRandom

#Credit to Antoine Prudhomme
#Antoine Prudhomme's website:
#https://medium.com/@prudywsh/how-to-generate-bi
g-prime-numbers-miller-rabin-49e6e6af32fb
def is_prime(n, k=128):
    """ Test if a number is prime
        Args:
            n -- int -- the number to test
            k -- int -- the number of tests to d
o
        return True if n is prime
    """
    # Test if n is not even.
    # But care, 2 is prime !
```

```
#Credit to Antoine Prudhomme
def generate_prime_number(length):
    p = 4
    # keep generating while the primality test fail
    while not is_prime(p, 128):
        p = generate_prime_candidate(length)

return p

#https://en.wikibooks.org/wiki/Algorithm_Implementat
ion/Mathematics/Extended_Euclidean_algorithm
def xgcd(a, b):
    """return (g, x, y) such that a*x + b*y = g = gc
d(a, b)"""
    x0, x1, y0, y1 = 0, 1, 1, 0
    while a != 0:
        q, b, a = b // a, a, b % a
        y0, y1 = y1, y0 - q * y1
```

``
    if n == 2 or n == 3:
        return True
    if n <= 1 or n % 2 == 0:
        return False
    # find r and s
    s = 0
    r = n - 1
    while r & 1 == 0:
        s += 1
        r //= 2
    # do k tests
    for _ in range(k):
        a = randrange(2, n - 1)
        x = pow(a, r, n)
        if x != 1 and x != n - 1:
            j = 1
            while j < s and x != n - 1:
                x = pow(x, 2, n)
                if x == 1:
                    return False
                j += 1
            if x != n - 1:
                return False
    return True
#Credit to Antoine Prudhomme
 	
def generate_prime_candidate(length):
    # generate random bits
    p = SystemRandom().getrandbits(length)
    # apply a mask to set MSB and LSB to 1
    p |= (1 << length - 1) | 1
    return p
``
	``
	        x0, x1 = x1, x0 - q * x1
	    return b, x0, y0
	
	#https://en.wikibooks.org/wiki/Algorithm_Implementation/Mathematics/Extended_Euclidean_algorithm
	def mulinv(a, b):
	    """return x such that (x * a) % b == 1"""
	    g, x, _ = xgcd(a, b)
	    if g == 1:
	        return x % b
	
	#Personal
	def generate_coprime_number(n):
	    l = randbelow(n)
	    while gcd(l, n) != 1:
	        l = randbelow(n)
	    return l
	
	#Personal
	def RSA(p, q):
	    n = p * q
	    totienN = (p-1) * (q-1)
	    e = generate_coprime_number(totienN) #Public
	    d = mulinv(e, totienN) #Private
	    return e, d, n
	``

```
def RSA_Encrypt(plain, e, n):
    return pow(plain, e, n)

def RSA_Decrypt(cipher, d, n):
    return pow(cipher, d, n)
```

```
#Generate 2 random inputs based on Key Size
keySize = 7
p = generate_prime_number(keySize)
q = p
while q == p:
    q = generate_prime_number(keySize)
print(p,q)
```

```
#Generate RSA keys using two random numbers
RSA_Keys = RSA(p, q)

print(RSA_Keys)
cipher = RSA_Encrypt(4, RSA_Keys[0], RSA_Keys[2])
print("Cipher Text:\n", cipher)

plain = RSA_Decrypt(cipher, RSA_Keys[1], RSA_Keys[2])
print("Plain Text:\n", plain)
```

Table 1: Code provided by Medium User Antoine Prudhomme, WikiBooks, and personal creations

After numerous test running the code in Table 1, it could be said that the code ran correctly and as indented. Next was to construct Shor's Algorithm with a classical version of the period finding function. Below is the code to implement period finding on a classic computer.

```
#Reducing factorization to period finding
def find Period(k):
    #pick a variable k that is not a factor of N; GCD of N and k = 1; k < N
    while gcd(k, RSA_Keys[2]) != 1:
        k += 1

    period = 2
    #find the period of x^period is congruent to 1 mod N
    while pow(k, period, RSA_Keys[2]) != 1 or (pow(k,int(period/2))+1)%RSA_Keys[2] == 0:
        if period < RSA_Keys[2]-1:
            #ensures that r is even and can be divided by two
            period += 2
        else:
            break
    #The above is the hardest part in breaking RSA cryptography
    return k, period

def getBaseCanidate():
    pCanidate, qCanidate = 1, 1
    a = 2
    #picks a different k if p or q = 1
    while pCanidate == 1 or qCanidate == 1:
        k, period = findPeriod(a)
        pCanidate = gcd(k**int(period/2)-1, RSA_Keys[2])
        qCanidate = gcd(k**int(period/2)+1, RSA_Keys[2])
        a = k + 1

    print("Factor:", k)
    print("Period:", period)
    print("Prime Keys:", pCanidate, qCanidate)

getBaseCanidate()
```

Table 2: Shor's Algorithm on a classical computer with no Quantum Part.

With these two programs constructed, Shor's algorithm will be tested on a classical computer with increasing levels of bit sizes of p and q. The speed of the program will be recorded and used to determine how effective the program is on a classical computer.

RSA Program				Period Finder Program		
RSA p, q bit size	*p*	*q*	*n*	*a*	*r*	*Execution Time (second)*
7	73	89	6497	3	264	0.001586208004
8	137	241	33017	2	408	0.001110128011
9	351	491	176269	2	87710	0.202961126924
10	821	773	634633	5	158260	1.999115473009
11	2027	1889	3829003	2	478136	0.926926229964
12	4007	3203	12834421	2	6413606	12.620535376947
13	5563	7817	43485971	3	21736296	85.532774055027
14	9931	13687	135925597	2	22650330	49.779755011084
15	17189	16451	282776239	2	28274260	59.323969553923
16	48761	52937	2581261057	2	161322460	573.520868192892

Table 3: displays the results of the classic period finding function

The above table displays the results of the period finding function running only on a classical computer. The execution time of the program was calculated using the python timeit API. It should be noted that after attempting to solve for 17 bit long p and q, the execution time took too long to wait for results. After viewing the results of the classical implementation, the next phase in the process is to try and implement the algorithm on a quantum computer. To do this, the circuit would be broken up into four parts based on Shor's algorithm. First part is superposition, which is easy to achieve since the Hadamard gates do that already; the second part is modular exponentiation, which is difficult to achieve considering that this can only be achieved using quantum logic gate; third step is to implement the Inverse Quantum Fourier Transform (QTF), which already exist and for the purposes of this experiment will be gathered from the IBM Q Experience community GitHub user, delapuente; lastly, the measurement which is included in the IBM quantum API.

```
#https://github.com/Qiskit/qiskit-terra/blob/master/examples/python/qft.py
def qft(circ, q, n):
    """n-qubit QFT on q in circ."""
    circ.h(q[0])
    for j in range(n):
        for k in range(j):
            circ.cu1(math.pi/float(2**(k+1)), q[j], q[k])
            if k == j-1: circ.h(q[j])
        circ.barrier()

regSize = 3
q = QuantumRegister(regSize, "q")
c = ClassicalRegister(regSize, "c")
qft3 = QuantumCircuit(q, c, name="qft3")

qft(qft3, q, regSize)
qft3.barrier()
for j in range(regSize):
    qft3.measure(q[j], c[j])
```

Table 4: The above code was provided by GitHub user, delapuente, to create the QTF.

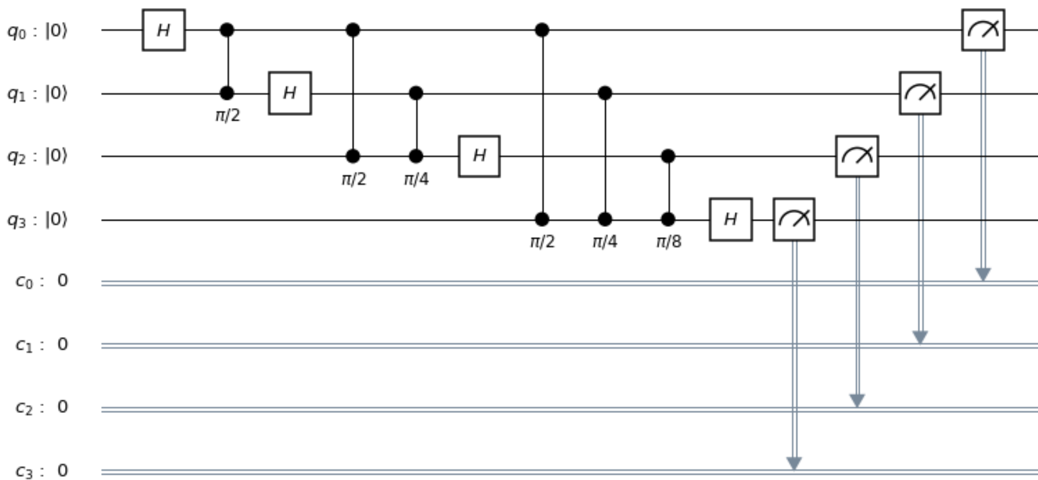

Figure 2: Is the Inverse Quantum Fourier Transform in a quantum register. The Quantum Fourier circuit is verified to be the ideal design in *Quantum Algorithm Implementations for Beginners*.

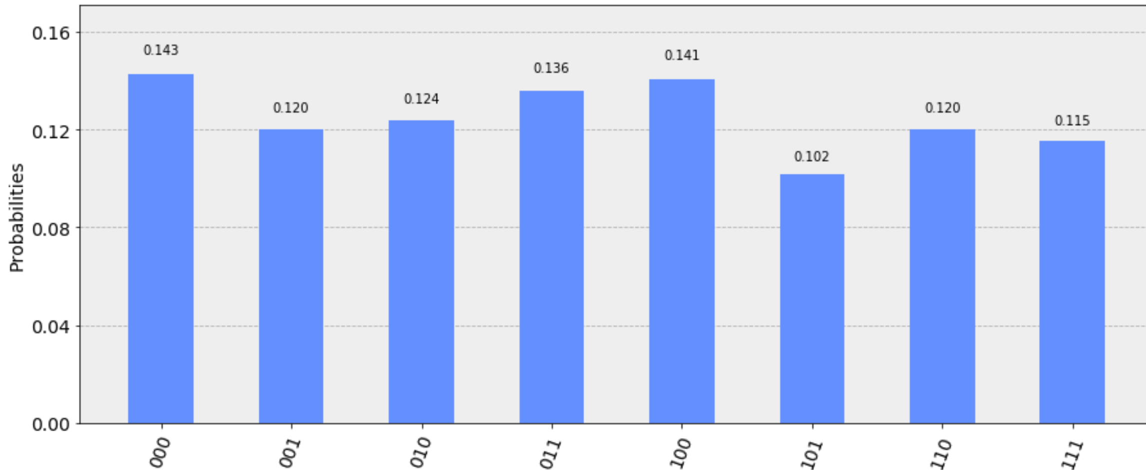

Figure 3: Results from running the the Inverse Quantum Fourier Circuit in Figure 2.

The code in Table 4 will be used to design the Inverse QFT for the period finding circuits that will be use simulated in the experiments. Since the Inverse QFT is achievable and the first and last part of Shor period finding function are provided by the using the Quantum API, the only part needed it the modular exponentiation part. In *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance*, there is circuit that was used to find the period of $7^r \text{mod } 15$. The circuit yields the correct results, outputting 4 and 0. The value 0 can be ignored since $7^0 \text{mod } 15$ will always be 1 and is a side effect of the Inverse QFT. The other result 4, is the period of $7^r \text{mod } 15$. It is the only circuit that was researched so far, that is able to find the period of modular exponentiation. Figure 4 is a recreation of the algorithm in which n = 15 and a = 7. The design of the circuit is different from Shor's Algorithm, as all the operations are done in one register instead of two.

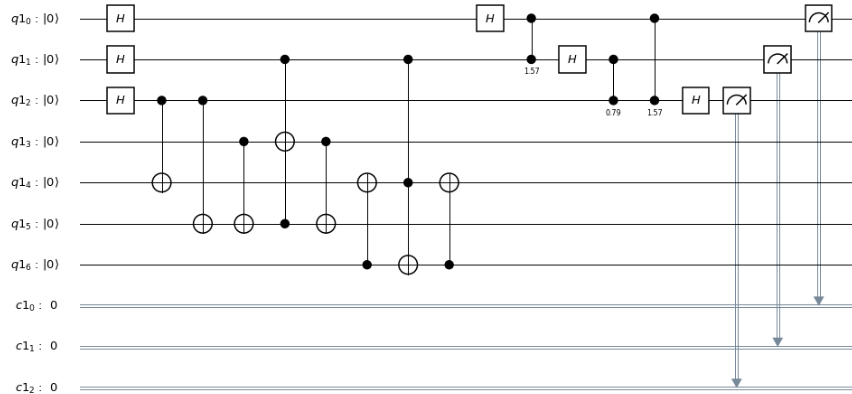

Figure 4: Circuit designed to calculate the period of $7^x mod\ 15$

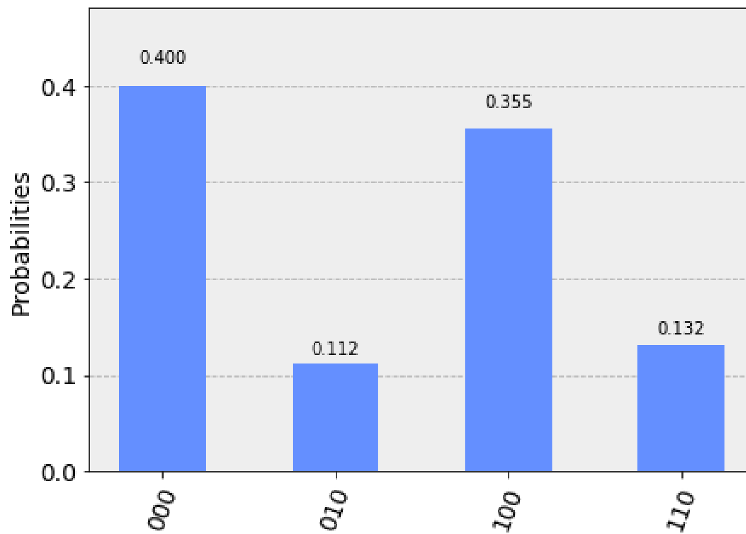

Figure 5: Results from circuit of Figure 2. Yield 0 and 4, which when calculated is $7^4 mod\ 15 \equiv 1$. Therefore 4 is the period which yields the results $gcd(7^2 - 1, 15) = 3$ and $gcd(7^2 + 1, 15) = 5$ which are the prime factors of 15.

Using the circuit from Figure 2 as a guide, a new circuit will be designed. Replacing the part of the circuit that is used for modular exponentiation should yield a different result. To construct to this part of the circuit, this was attempted with trial and error. The logic behind the circuit is to store the results of 2^2mod 35, 2^4mod 35… 2^{n-1}mod 35 in the second half of the register depending on which qubit in superposition are in the 1 basis states.

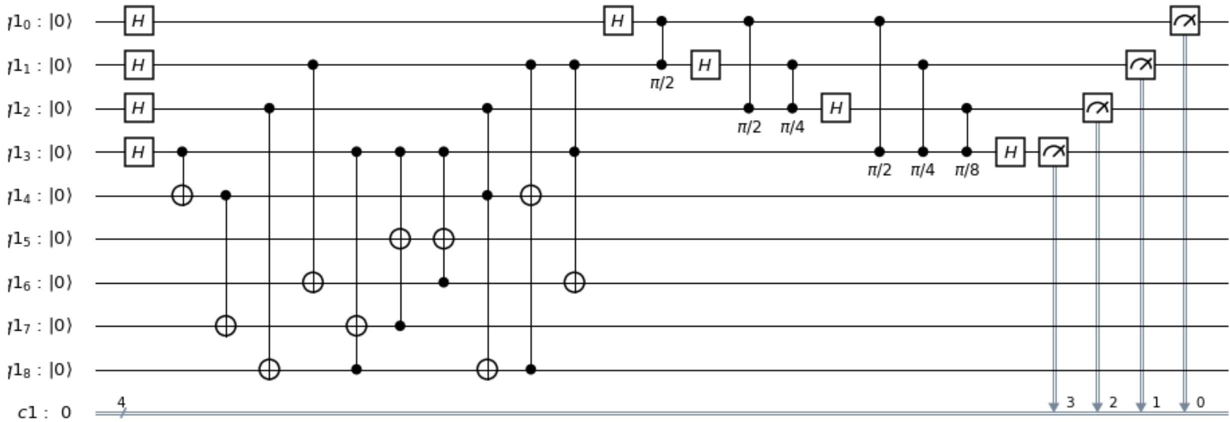

Figure 6: Circuit designed to calculate the period of $2^r \text{mod } 21$

For example if the first four qubits were 0100, meaning 4, the last five qubits should be 10000, or 16, which is $16 \equiv 2^4 \text{mod } 35$. The results, shown in from the histogram in Figure 7, show that circuit does not represent a modular exponentiation equation with a period.

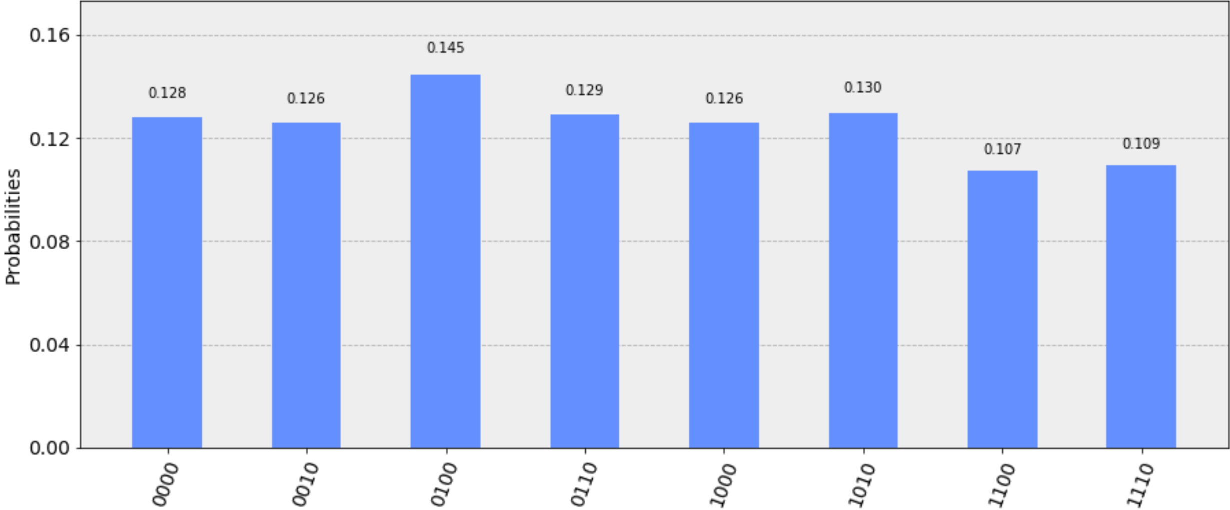

Figure 7: Results from running Circuit in Figure 4

After testing the above circuit five times it would appear that the resulting numbers are all equally likely meaning and that no period was detected. The circuit in Figure 6 and approach were essentially a failure.

Numbers Tested		Probability				
0bn	n	Test 1	Test 2	Test 3	Test 4	Test 5
0000	0	11%	12%	12%	12%	13%
0010	2	13%	12%	13%	12%	12%
0100	4	11%	11%	11%	13%	12%
0110	6	12%	14%	14%	13%	15%
1000	8	13%	13%	13%	12%	13%
1010	10	14%	11%	13%	14%	12%
1100	12	12%	12%	13%	12%	13%
1110	14	13%	15%	13%	13%	12%

Table 5: Results from the Circuit in Figure 6

Since the attempt was a failure, a new approach to the quantum circuit design will involve less gates. Moving on to Quantum Circuit Design 2; referring back circuit in Figure 4, it would appear that the control qubits of the controlled not gates (CNOTs) are in the same position as the results of the circuit; there may be a correlation. To test this, in the second attempted period finding circuit, the control qubits of CNOTs should represent the desired period in the circuit. The circuit is designed to find the period of $7^r \text{mod } 55$. For the equation $7^r \text{mod } 55$ the expected period is 20, therefore the control qubits will be in Register bit position 2 and 4, which represent the values 4 and 16 respectfully. The number of the initial Hadamard gate and size of the QFT were determined by the number of bits in 55, which is 6.

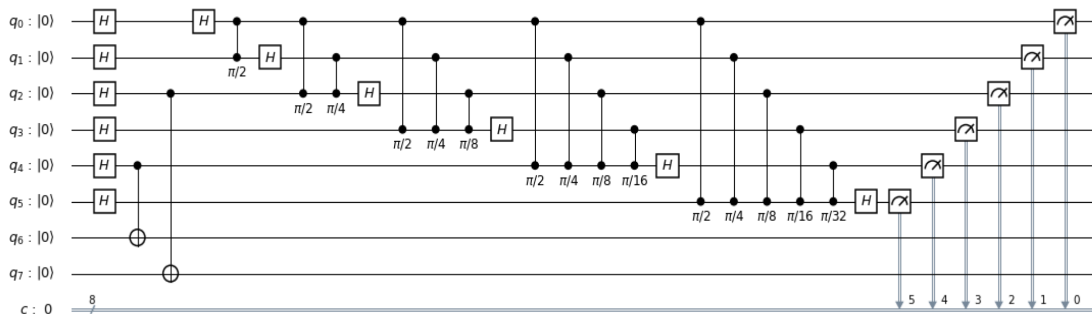

Figure 8: Quantum circuit designed to find period of $7^r \text{mod } 55$

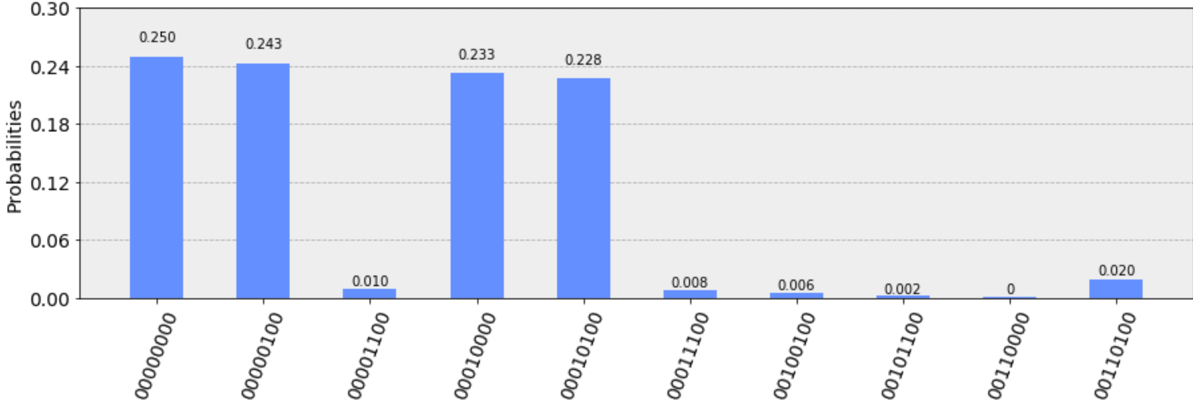

Figure 9: Results of Quantum circuit designed to find period of $7^r \bmod 55$

The circuit results show the expected result, 20, has a high probability of appearing on a quantum computer; however, it also shows unexpected results, 4 and 16, with close to or equal probability. It may be that because 4 and 16 equal 20, which is the desired result, 4 and 16 are equally likely as 20. This may mean that the circuit design is close to our desired results. To achieve the desired result, the next circuit will be a modification of the circuit in Figure 8. Two more gates will be added to the circuit to attempt to remove 4 and 16. The logic behind the placement of the gates is that if it is assumed that there is some quantum entanglement between superposition of the control qubits and their respective target qubits then configuring the target qubits so that they can logically only be 10100, then the control qubits should have the same behavior which would be reflected in the Inverse QTF.

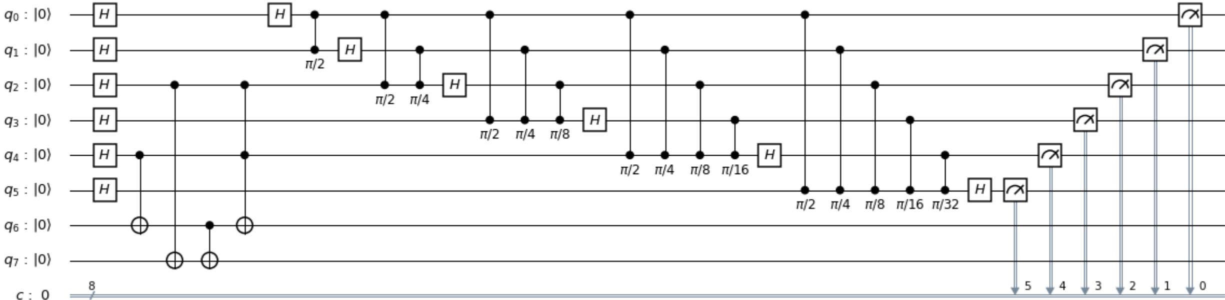

Figure 10: Modified version of the circuit in Figure 8. Added one CNOT on qubit 7 with target qubit qubit 6. Added a Toffoli Gate to Target qubit 6 with control qubits, 2 and 4.

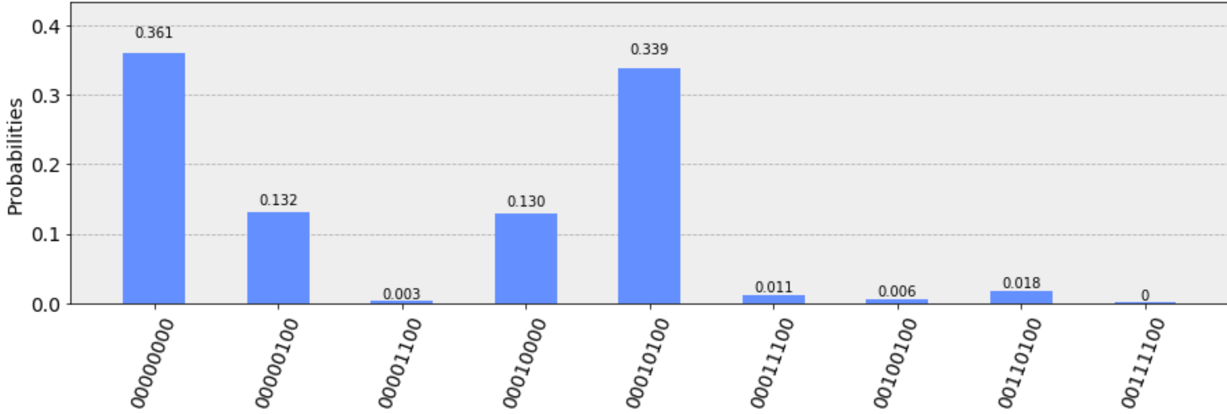

Figure 11: The results of the circuit in Figure 10 have found the desired result

The circuit yields the correct result. To examine whether this logic is exclusive to this circuit, the logical placement of the gates will be applied different circuit attempting to find a different period. The circuit in Figure 10 will be modified again to find the period of equation 5^rmod 33 with expected period of 10.

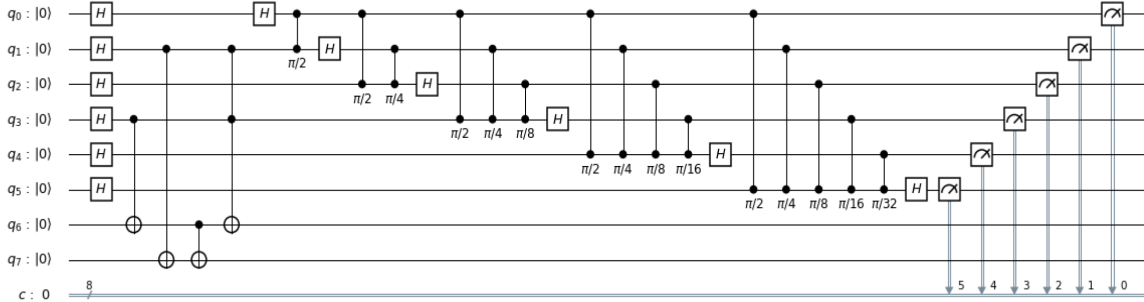

Figure 12: Shows the circuit design to find the period of 5^rmod 33

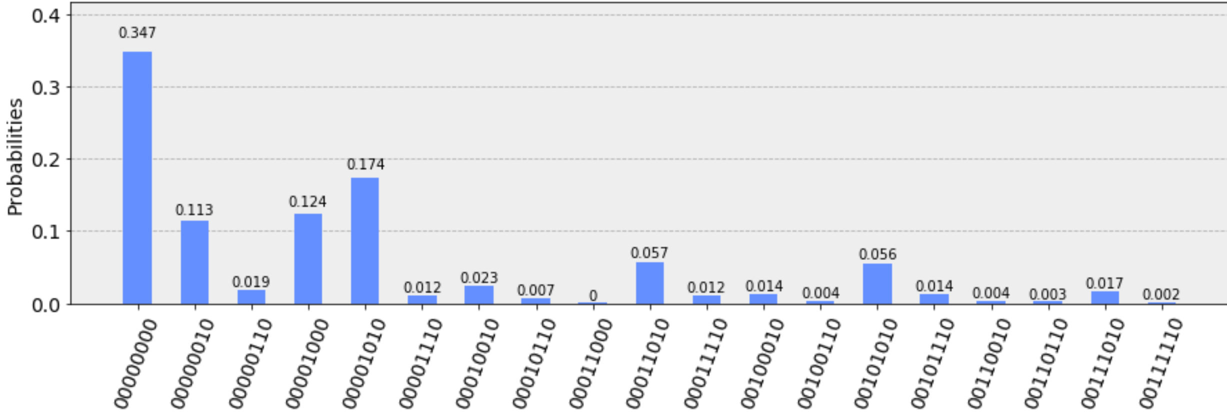

Figure 13: Results from circuit designed to find the period of 5^rmod 33

The results from the second modified circuit, shown in Figure 13, show that the expected period, 10, was achieved. Ignoring 0, 10 has the second highest probability of appearing on a classical computer. This experiment show that when considering entanglement, results from the quantum circuit can be manipulated to the desired results.

Conclusion

To conclude the results of the experiments, although not highly significant, represents an approach on how to potentially realize the modular exponentiation of Shor's period finding function. For this research quantum entanglement may be the cause of the logic of the circuit working. This highlights the fact that quantum registers are not like the registers on classical computers. Quantum Phenomenas like interference and super position play a role in the behavior of the circuits, meaning that a classic approach to logically understanding quantum circuit may not be the correct approach. The approach of simply storing the results on the second half of the register may not be the right approach for Shor's algorithm. Entanglement of the qubits should be taken into consideration when trying to reach Shor's Algorithm.

Future Research

Since the field of quantum computing is changing constantly, there is plenty of opportunity to do further research. This includes other area of science like physics, chemistry, and biology. In regards to cybersecurity, areas to research involving quantum computers are creating quantum circuits designed to apply Shor's algorithm to Elliptic Curve Cryptography, creating quantum circuits that can perform modular exponentiation, creating circuits that can be

applied to more than just equation, and testing Shor's algorithm with other quantum simulations. While this research may not be immediate available, as quantum computers are still being developed and improved, coming up with theorems or testing existing algorithms could benefit the quantum community and security communities.

Works Cited

Coles, Patrick J., et al. *Quantum Algorithm Implementations for Beginners*. Los Alamos National Laboratory, Los Alamos, New Mexico, USA, 2018. *arXiv.org*.

IBM. "What is Quantum Computing?" *IBM Research - Home*, www.research.ibm.com/ibm-q/learn/what-is-quantum-computing/.

Lomonaco Jr., Samuel J. "A Lecture on Shor's Quantum Factoring Algorithm." *ArXiv.org*, 9 Oct. 2000, arxiv.org/abs/quant-ph/0010034v1.

Mermin, David. "Breaking RSA Encryption with a Quantum Computer: Shor's Factoring Algorithm." Physics 481-681, CS 483; Spring, 2006, Cornell University. Lecture.

National Institute of Standards and Technology, et al. *Report on Post-Quantum Cryptography*. National Institute of Standards and Technology, 2016.

Paar, C., and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Science & Business Media, 2010.

Shor, Peter W. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." *SIAM Journal on Computing*, vol. 26, no. 5, 1997, pp. 1484-1509, *arXiv.org*. doi:10.1137/S0097539795293172.

Vandersypen, Lieven M., et al. "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance." *Nature*, vol. 414, no. 6866, 2001, pp. 883-887, *arXiv.org*. doi:10.1038/414883a.