University of Birmingham

Final year project

Estimating Quantum Gate Costs for Implementing Shor's Algorithm in Hyperelliptic Curve Cryptography

Sam Fishlock

School of Computer Science

Supervised by

Dr. Christophe Petit
School of Computer Science

8 April 2019

1 Acknowledgments

I would like to thank Dr. Christophe Petit for his guidance and supervision over this project.

Contents

List of Figures

List of Tables

Abstract

This project provides an estimate for the number of quantum gates required to implement Shor's Algorithm for solving the discrete logarithm problem over hyperelliptic curves of genus 2 in polynomial time. We conclude that the problem can be solved on a quantum circuit with at most $11n + 2\lceil \log_2 n \rceil + 10$ qubits, with a gate count in the order of $1472n^3 \log_2 n$, which ultimately leads to a circuit which has a lower overall gate count than both elliptic curves and RSA. The circuit for both RSA and elliptic curves would require ~ 1000 qubits, whereas the circuit for hyperelliptic curves of equivalent security would require ~ 600.

2 Introduction

2.1 Background

Since Shor first introduced a polynomial time algorithm for prime factorisation and discrete logarithms [5], there has been a lot of interest in quantum computing and being able to create a working set of hardware which realises Shor's theoretical algorithm. Perhaps the largest difficulty faced when trying to create a quantum computer is quantum decoherence [6] - this is when a qubit interacts with its surroundings and leads to a collapse of the superposition. Quantum decoherence becomes increasingly hard to eliminate when more qubits are added to the processor. Despite this obstacle, there have been several quantum computers manufactured: in May 2016, IBM released the IBM Q Experience; a five qubit quantum processor, and in May 2017, they expanded this to add a 16-qubit processor which has been shown to be able to become fully entangled [7]. At present, the largest functional quantum computer, named Bristlecone, has 72 Qubits and is made by Google [8].

2.2 Appliances of Quantum Computing

Quantum computers have been shown to be faster than classical computers at solving certain problems [9], and one of these areas which is of particular importance is the area of cryptography. Common cryptographic schemes used widely today implement a function which, when applied to the text, is unable to be reversed. One such example of a scheme which follows this is RSA, which relies on the fact that it is very hard for classical computers to perform prime factorisation when the prime factors for that number are large. The most efficient classical algorithm to break RSA is the General Number Field Sieve [10], which is sub-exponential in complexity, Using Shor's algorithm, we can break RSA in bounded-error quantum polynomial time (BQP), which is almost exponentially faster than the General Number Field Sieve. Although the theory suggests that quantum computing can be very powerful, the hardware is not currently there to support it - for example the largest number factored by Shor's Algorithm is 21, which required 10 qubits [11], however the largest number factorised on a quantum computer is 56,153, which required 4 qubits [12]. These are still far off from classical computing capability, for scale, the largest number factored using classical computing is RSA-768, a 232 digit semiprime.

2.3 Objectives

Shor's algorithm can be adapted to solve elliptic curve discrete logarithm problems (ECDLP) [13], which is the set of non-inversible functions that are used in elliptic curve cryptography. These cryptographic schemes rely on the fact that given a point Q on an elliptic curve, which is some multiple of an original point, P, such that $Q = [m]P$, it is very hard to calculate m on classical computers. It has been estimated that it would require a quantum computer with approximately 1000 qubits to break a 160 bit elliptic curve cryptographic key, and approximately 2000 qubits

to break the equivalently secure 1024 bit RSA [13]. This project builds on previous work and provides an estimate for the number of quantum gates required to break the discrete logarithm problem for hyperelliptic curves of genus 2.

3 Elliptic and Hyperelliptic Curve Cryptography

3.1 Discrete Logarithm Problem and Cryptography

Elliptic and Hyperelliptic curve cryptography are schemes which can be derived from the larger general class of discrete logarithm cryptography schemes. In Whitfield Diffie and Martin Hellman's article: *New directions in Cryptography* [14] it is described how we can generate a cryptographic scheme from a discrete logarithm, this section will summarise this. We can define the general discrete logarithm problem as follows: suppose we have a prime p, we can create a prime field from this prime by setting the integers modulo p, denoted as: $\mathbb{Z}_p^*$ These types of fields are also typically called Prime Galois Fields and can be denoted as $GF(p)$. From this, we can define a cyclic subgroup: $g \in \mathbb{Z}_p^*$ so that we now have a group which we can define an arithmetic function over, and if we repeat that arithmetic function on an element some amount of times, we will return to our original element. Now we can define the discrete logarithm problem: Let $Y = \alpha^X \mod p$, for $1 \leq X \leq p-1$, where α is a fixed primitive element of $GF(p)$. Rearranging this equation, we then have $X = \log_a Y \mod p$ for $1 \leq Y \leq p-1$, and X can be referred to as the logarithm of Y to the base α, $\mod p$. To calculate Y is easy, and we can use the square and multiply algorithm, which is described in Algorithm 1. This takes at most $2\log_2 p$ multiplications to calculate Y from X [15]. For example for $X = 18$, we have

$$Y = \alpha^{18} = (((\alpha^2)^2)^2)^2 \times \alpha^2$$

It is very hard to calculate X given Y however, and for carefully chosen values of p, requires on the order of $p^{1/2}$ operations using the best known algorithm [16]. We can turn this one way function into a cryptographic scheme as follows: We have two parties, Alice and Bob and they agree publicly on a value for g, an element from a prime group of size p. Alice picks a random element

Algorithm 1 Square and Multiply Algorithm

Input: Element α, X
Output: Element $Y = \alpha^X \mod p$

```
1: function F(α, X)
2:     if X = 1 then
3:         return X mod p
4:     else if X mod 2 = 0 then
5:         return F(α^2 mod p, X/2)
6:     else if X mod 2 = 1 then
7:         return α^2 × F(α, X − 1) mod p
```

a and sends to Bob $g^a \mod p$. Bob does the same for some random element b : $g^b \mod p$. Alice then computes $(g^b)^a \mod p = g^{ab} \mod p$, and Bob computes $(g^a)^b \mod p = g^{ab} \mod p$. Now both parties have a shared secret. If some attacker Eve wants to find a, b or $g^{ab} \mod p$ from $g^a \mod p$ and $g^b \mod p$, she will have to solve a discrete logarithm problem. This scheme is called Discrete Logarithm Diffie Hellman, also known as Discrete Logarithm DH protocol, and is used to set up a secure communication channel between two parties. There are many more schemes we can derive from discrete logarithm problems, such as certificate generation and verification. Any cyclic group can have a cryptographic function defined from it, as long as the discrete logarithm of that group is considered hard to compute. The next sections will describe how we can use elliptic and hyperelliptic curves to define a group which we can then define a discrete logarithm problem over.

3.2 Elliptic Curve Cryptography

An elliptic curve is a set of points that satisfy an equation in the form: $y^2 = x^3 + ax + b$. This type of equation is called a Weierstrass equation. The elliptic curve must also be non-singular, which means that the curve has no cusps, self-intersections or isolated points. A cusp is where a function's gradient becomes infinitely large before returning back on itself, an example of this is shown in Figure 1. These properties can be achieved algebraically by making sure that the

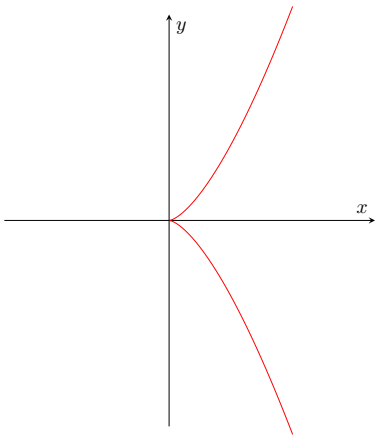

Figure 1: Example of cusp on graph $y^2 = x^3$

discriminant of the curve is not equal to 0:

$$\Delta = -16(4a^3 + 27b^2) \neq 0$$

Elliptic curves over $\mathbb{R}$ generally look similar to the one in Figure 2.

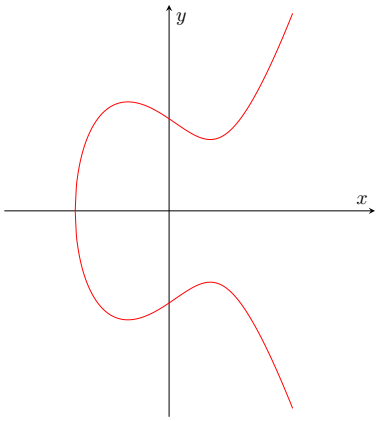

Figure 2: Elliptic Curve satisfying $y^2 = x^3 - 3x + 5$

3.2.1 Elliptic Curve Arithmetic

We can "add" two points on an elliptic curve P and Q to give another point R on the curve. This operation is called the dot operation. We can show this operation visually by drawing a straight line through points P and Q, and where this line meets the curve again, will be the "inverse" of the result of our dot operation, R'. To get R, we can draw a vertical line and take the point where this line meets the curve. This process is described in figure 3. We can then dot

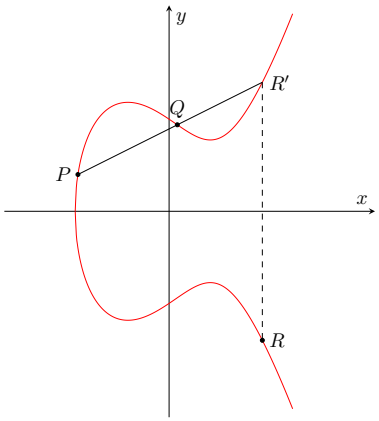

Figure 3: Dot operation on two points P and Q to give final point R

P and R using the same method as before to give us another point on the curve, S, as shown in figure 4. For our cryptographic function we also need one more operation, point doubling. Point doubling can be achieved by drawing the tangent of the curve at a point P, and similarly to point addition, where this tangent intersects the curve will be the inverse of our final point, Q'. To get our final point we draw a vertical line and where this line intersects the curve will be our final point Q. This process is shown in figure 5.

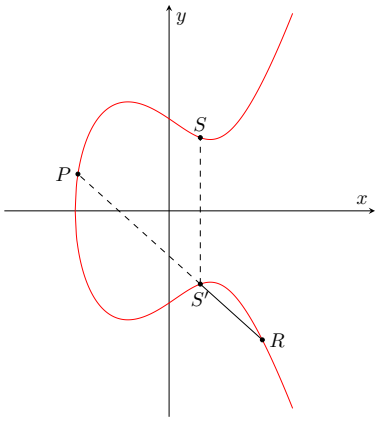

Figure 4: Dot operation on points R and P to give final point S

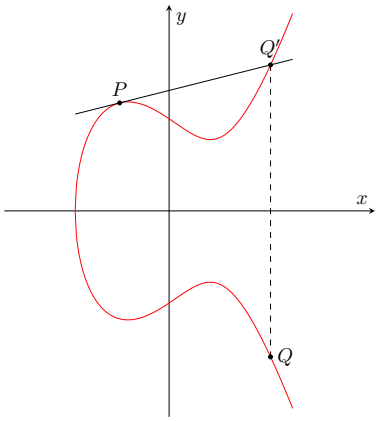

Figure 5: Point doubling operation on point P to give point $Q = 2 * P$

3.2.2 Adapting Elliptic Curve Arithmetic into a Cryptographic Function

In order to turn our arithmetic into a cryptographic function we must firstly restrict our infinitely many points to a subset of points which is finite. This gives rise to the idea of elliptic curves over finite fields. Particularly useful Galois Fields for our goal of creating a cryptographic function are prime fields of order p and binary extension fields, which can be represented as GF(p) and GF(2^m) respectively. GF(p) has a prime number of elements equal to p which can be represented as integers, and GF(2^m) has 2^m elements which can be represented as polynomials. Fields of the form GF(2^m) are particularly useful from a hardware perspective, as elements of the field can be represented as binary strings. For example, the element $x^3 + x + 1$ can be represented in binary as 1011, with each bit corresponding to a coefficient of the polynomial. The set of points of an elliptic curve over a finite field is the set of points in that finite field that satisfy the equation of the elliptic curve. Figure 6 shows our elliptic curve when it is restricted to elements of the finite field GF(97), it does not resemble our previous curve at all, this is because only the points that

hit whole number coordinates are displayed, and the points whose coordinates exceed our prime (97) are wrapped around, e.g. 100 would become 100 mod 97 = 3. With this representation of

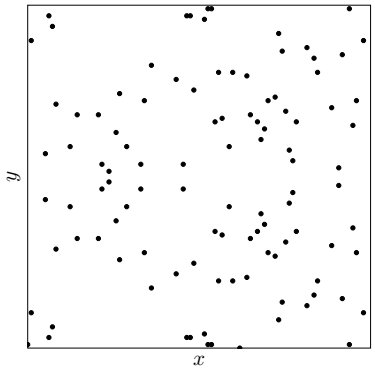

Figure 6: The elliptic curve $y^2 = x^3 - 3x + 5$ restricted to elements of the finite field GF(97)

the elliptic curve, we can also visually show how we can perform arithmetic on points on the curve - to add points, we draw a straight line between the points and when this line hits the "edge", it overlaps continuing from the opposite edge. The point that this line intersects will be the inverse result of adding our two points, so we again draw a vertical line and take the inverse to get our result. This process is shown in figure 7.

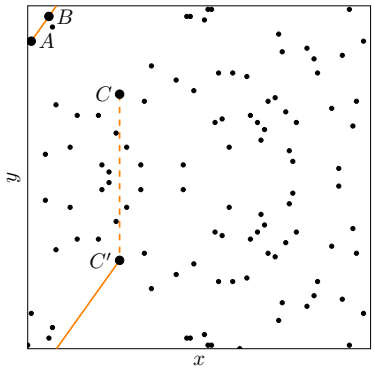

Figure 7: Dot operation on A and B when restricted to finite field GF(97)

Using our point double and point addition arithmetic, we can produce any scalar multiplication of a point on an elliptic curve. There are various algorithms which can produce a scalar multiplication of a point, perhaps the most simple would be the "double and add" algorithm. This algorithm is similar to Algorithm 1 for square and multiply, but we instead perform point doubling and point addition. Algorithm 2 shows this process. Once we have our point $Q = [m]P$, it is thought to be very hard to find m given Q. This is the basis of the cryptographic function, called the Elliptic Curve Discrete Logarithm Problem (ECDLP). Currently, the best known classical algorithms to solve ECDLP are exponential in the size of the input parameters [17]. The advant-

Algorithm 2 Double and Add Algorithm for Scalar Point Multiplication

Input: Point P, Integer m
Output: Point $Q = [m]P$

```
1: function SCALAR_MULTIPLY(P, m)
2:     if n = 1 then
3:         return P
4:     else if n mod 2 = 0 then
5:         return scalar_multiply(double_point(P), n/2)
6:     else if n mod 2 = 1 then
7:         return add_points(P, scalar_multiply(P, m-1)
```

ages of ECC over commonly used cryptography scheme RSA are that a shorter key can be used whilst keeping the same level of security. We can also generate keys and signatures faster, as well as verifying signatures faster [3]. This allows for us to generate same levels of security with less hardware, making ECC a good candidate for encryption in cloud services, phones, smart cards, where hardware is at a premium.

Symmetric	ECC	RSA
80	163	1024
112	233	2240
128	283	3072
192	409	7680
256	571	15360

Table 1: Key lengths needed for equivalent security in ECC and RSA [3]

3.3 Hyperelliptic Curve Cryptography

While elliptic curves are a subset of hyperelliptic curves with genus equal to 1. A hyperelliptic curve of genus greater than 1 can be formally defined as a curve which satisfies the equation $y^2 + h(x)y = f(x)$ subject to some constraints on $h(x)$ and $f(x)$. The function $f(x)$ is a monic polynomial, the degree of which determines the genus g of the curve. The degree of $f(x)$ can split hyperelliptic curves into two main categories: real hyperelliptic curves and imaginary hyperelliptic curves. When the degree of f(x), d is equal to $2g+1$, then the curve obtained is said to be imaginary. A degree of $2g + 2$ gives a real hyperelliptic curve. This project will focus on imaginary hyperelliptic curves, as these can be developed into a cryptographic function. We can see that our definition of elliptic curves obeys this definition, as an elliptic curve is a hyperelliptic curve of genus 1, and the defining function of elliptic curves must be a monic polynomial of degree 3. The function $h(x)$ is a polynomial of degree less than $g+2$, if the characteristic of the field that the hyperelliptic curve is defined over is not equal to 2, then $h(x) = 0$. Similarly to elliptic curves, the field that the hyperelliptic curve is defined over greatly affects the efficiency of performing arithmetic on the curve. In practice, fields of characteristic 2 have been proven to be the most

efficient fields to implement arithmetic over [18]. Along with these restrictions, we also require the curve to have no singular points, in the same way that elliptic curves must be. If we view a hyperelliptic curve as lying in the projective plane, $\mathbb{P}^2(K)$, with coordinates (X, Y, Z), then there is a particular point on the curve, known as the point at infinity: $O = (0, 1, 0)$. We can define the opposite of a point $P = (x, y)$ as $\bar{P} = (x, -y - h(x))$. Figure 8 shows a typical imaginary hyperelliptic curve over the real numbers.

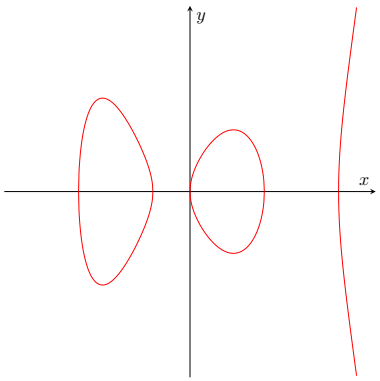

Figure 8: Hyperelliptic Curve satisfying $y^2 = x^5 - 5x^3 + 4x + 1$

This section will introduce hyperelliptic curves, their group arithmetic operations and how we can define a cryptographic function from these operations.

3.3.1 Hyperelliptic Curve Arithmetic

Unlike elliptic curves, there is no group law arithmetic operation over the points of a hyperelliptic curve, this can be shown by the fact that a straight line through two points on a hyperelliptic curve will not always intercept the curve at one more unique point, in fact, Bézout's theorem states that a straight line and a hyperelliptic curve of genus 2 intersect at 5 seperate points. Instead, we must introduce the notion of a divisor, and the Jacobian of a curve C, denoted J_C. The laws of arithmetic and notation described in this section have been taken from *Handbook of elliptic and hyperelliptic curve cryptography* [19]. A divisor, defined by:

$$D = \sum_{P_i \in C} n_i [P_i]$$

is a set of points on C with integer coefficients $n_i \in \mathbb{Z}$ which are 0 for almost all i. The degree of a divisor, $\deg(D)$, is the sum of its coefficients. The set of divisors over a curve is an Abelian group, denoted by Div_C, and the subset of degree 0 divisors is denoted by Div_C^0. A function on C, denoted as ϕ, is a rational fraction in the coordinates x and y of a point on C. The valuation of a function at a point C is denoted as $v_P(\phi)$. We can then define the divisor of a non-zero

function on C as:

$$div(\phi) = \sum_{P \in C} v_P(\phi),$$

a divisor of this type is known as a principal divisor. The set of functions over C is called the function field of C and is denoted by $K(C)$, from this, we can obtain the group of principal divisors of the curve C, denoted as $Prin_C$. The set of principal divisors form a subgroup of the group of degree 0 divisors:

$$Prin_C \subset Div_C^0$$

Finally, the Jacobian of the curve, denoted as J_C, can be defined as the quotient group: $J_C = Div_C^0 / Prin_C$. To turn the Jacobian into something we can define a group law arithmetic from, we look at the concept of reduced divisors. A divisor D of C can be called reduced if it has the form

$$D = \sum_{i=1}^{k} [P_i] - k[O],$$

where $k \leq g, P_i \neq O$. A reduced divisor can be represented by a unique pair of univariate polynomials, conventionally named u and v, such that u is monic, and $\deg(v) < \deg(u) \leq g$, this is called the Mumford representation of a reduced divisor, and is very useful for arithmetic purposes. These polynomials take the form:

$$u(x) = \sum_{i=1}^{r} (x - P_i(x)),$$

$$v(P_i(x)) = P_i(y),$$

where $P_i(x)$ and $P_i(y)$ represent the x and y coordinates of point P_i respectively. Therefore, to transform a reduced divisor into its Mumford representation, we first obtain u by setting its roots to the x coordinates of each of the points in the divisor. We can then obtain v as the lowest degree function which passes through each point in the divisor. For genus 2 curves, this is actually quite a simple process: Consider the curve $y^2 = x^5 - 5x^3 + 4x + 1$ over $\mathbb{R}$, we have two points on this curve: $P_1 = (0, 1)$, $P_2 = (1, -1)$. We can define a reduced divisor from these two points as $D = P1 + P2 - 2[O]$. Then, we can see that

$$u = (x - 0)(x - 1) = x^2 - x,$$

and

$$v(0) = 1, v(1) = -1 \therefore v = -2x + 1$$

Once we have our reduced divisors in Mumford representation, there are formulae for addition and doubling. The first explicit formula for addition was originally developed by David G. Cantor, hence the name Cantor's algorithm. Cantor's algorithm was further developed by Neal I. Koblitz to look at the general case for curves of any genus. Cantor's algorithm is described in Algorithm

3, originally taken from his paper: *Computing in the Jacobian of a hyperelliptic curve.* [20]. Cantor's algorithm uses 3 inversions and 70 multiplications, so it has a total complexity of 3I + 70M. Since he first introduced his algorithm, there have been many improvements to the complexity of algorithms for hyperelliptic curve arithmetic, the algorithms which we will use in this project have been taken from [19]. The addition of two divisors on a hyperelliptic curve

Algorithm 3 Cantor's Algorithm

Input: Two divisors $\bar{D_1} = [u_1, v_1]$ and $\bar{D_2} = [u_2, v_2]$ on the curve $C : y^2 + h(x)y = f(x)$ of genus g

Output: The unique reduced divisor D such that $\bar{D} = \bar{D_1} \oplus \bar{D_2}$

$d_1 \longleftarrow \gcd(u_1, u_2)$
$d \longleftarrow \gcd(d_1, v_1 + v_2 + h)$
$s_1 \longleftarrow c_1 e_1, s_2 \longleftarrow c_2 e_2$ and $s_3 \longleftarrow c_2$
$u \longleftarrow \frac{u_1 u_2}{d^2}$ and $v \longleftarrow \frac{s_1 u_1 v_2 + s_2 u_2 v_1 + s_3(v_1 v_2 + f)}{d}$ $mod\ u$
repeat
    $u' \longleftarrow \frac{f - vh - v^2}{u}$ and $v' \longleftarrow (-h - v)$ $mod\ u'$
    $u \deg u'$ and $v \longleftarrow v'$
until $deg(u) \leq g$
make u monic by dividing through by value of first coefficient
return $[u, v]$

can also be visualised: we can define a polynomial function which passes through each of the points in each of the divisors of degree d, and this line will intersect the curve in exactly d more positions, which will be the opposites of our points which make up our final divisor, similar to the straight line analogy for elliptic curve arithmetic. This process is shown in figure 9.

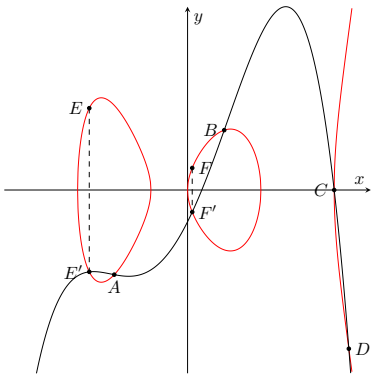

Figure 9: Visual representation of adding two reduced divisors: $(A + B) \oplus (C + D) = (E + F)$

3.3.2 Adapting Hyperelliptic Curve Arithmetic into a Cryptographic Function

Hyperelliptic curve arithmetic can be adapted into a cryptography scheme in a similar way that elliptic curve arithmetic can, since the set of reduced divisors which make up the Jacobian of the curve form a cyclic group. First, we choose a finite field to define the curve over, typical choices

for fields are binary fields and prime fields. Then we can generate a key pair by selecting an initial divisor $D_1 \in J_C$, and each party generating a random integer k, which we will use as a scalar multiplier to get a second divisor D_2. The key pair will then be (k, D_2) for each party.

3.3.3 Performance of Hyperelliptic Curve Cryptosystems

A genus g hyperelliptic curve will have roughly q^g points, where q denotes the number of elements in the field that the Jacobian is defined over. This means when we have a larger genus, we can reduce the size of the field and keep the same levels of security because the order of the group will remain the same. There have been various studies into the performance of hyperelliptic curves against elliptic curves, the main obstacle facing hyperelliptic curve cryptography is the complex nature of the operations required. Table 5 from [21] shows the performance of standard hyperelliptic curves compared to standard elliptic curves, with the hyperelliptic curves of genus 2 being a factor of 1.5 slower than elliptic curves. However, there have been more recent studies such as [22] which suggests implementations of genus 2 curves using Kummer surfaces that can outperform standard NIST elliptic and hyperelliptic curves. Another advantage that hyperelliptic curves have is that the operand length for operations is half (or less) than that of elliptic curves due to the fields being smaller. This means that systems with less complex processors with low or constrained power sources such as those used in many devices in the field of mobile and ubiquitous computing can use hyperelliptic curve cryptography to provide a high level of security.

3.4 Attacks against Discrete Logarithm Cryptography

There are various classical attacks against all forms of discrete logarithm problem cryptographic functions. One such attack is called the "Pohlig-Hellman" attack, and essentially reduces the group into smaller prime subgroups, this is why often a prime order group is used to define the scheme over, as the group cannot be reduced into smaller subgroups, however it is sufficient to use a group that has a large prime factorisation, so that if someone were to reduce the group, they would still reach a large prime subgroup. Pohlig-Hellman can be used in conjunction with Pollard's rho algorithm for computing discrete logarithms to obtain an attack against the cryptography system which has a run time of $O(\sqrt{p})$, with p being the largest prime factor of the order of the group which the scheme is defined over. For this reason, we must choose a sufficiently large p so that this attack becomes infeasible. This prime p, is used to define the key size, denoted by $\lceil \log p \rceil$. The key size determines the security of a discrete logarithm cryptographic system, as well as the running time of encryption and decryption, so we do not want it to be too large as that will affect the speeds that we can encrypt and decrypt at. Another attack which can be used against hyperelliptic curve cryptography is the "index calculus algorithm", which becomes more efficient than Pollard's rho method when the genus of the curve is too large. This algorithm has a running time of $O(q^{2-\frac{2}{g}+\in})$ where g is the genus of the curve [23]. For this reason, the genus of a curve is suggested to be less than 3 to ensure security against this attack [24].

4 Quantum Computing

This section will introduce the field of Quantum Computing, the theories and concepts and some algorithms, leading up to an adaptation of Shor's algorithm for discrete logarithms of elliptic and hyperelliptic curves. The information in this section has largely been adapted from [25], with various diagrams and examples being modified from [4].

4.1 Properties of Quantum Computers

4.1.1 Qubits

To begin to describe quantum computing, we can start at the very basis of the concept. The basis of a classical computer is a bit, which is usually stored in a transistor which can either have a voltage or not, i.e. the bit holds either a 1 or a 0. In a similar way, we can define the notion of a *qubit*, the equivalent to a classical bit, and how it is physically implemented. Many particles and atoms posess a quality called "magnetic spin", which means that they can be deflected under a magnetic field. A particle's spin can either be up spin or down spin, meaning that the particle will either be deflected up or down when subject to a magnetic field, particles will never be deflected to the side or any other direction. Until we measure this spin property, the particle exists in a "superposition" of both up and down spin, and we have no way of predicting which type of spin the particle will have. Once the spin of the particle is measured, then this superposition collapses and the particle behaves in a non quantum way. A way of representing this spin is to assign a probability to each state as such:

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$$

Where $|\psi\rangle$ represents the superposition of the particle, $\alpha_0 |0\rangle$ represents the probability α_0 of the particle having up spin $|0\rangle$, and $\alpha_1 |1\rangle$ represents the probability of the particle having down spin. The notation $|0\rangle$ is known as *ket* notation, and is used frequently when describing quantum computing. This superposition then becomes our qubit, on which we can perform various operations to affect the output value, and the building block from which we can build a quantum system.

4.1.2 Quantum States

A classical system with n components, each of which can have two states (1 or 0) can be represented in full by n bits. A quantum system with n qubits would require 2^n complex number coefficients, or more precisely, the state of the quantum computer can be thought of as a point in a 2^n-dimensional vector space, and for each of these 2^n states, we can represent them as a probability multiplied by a basis state, for example: $\alpha_i |0010\rangle$ means that the probability of the system being in state $|0010\rangle$ is equal to α_i. The state of the whole system at any time can be

represented by a *Hilbert space* defined as:

$$|\psi\rangle = \alpha_0 |\phi_0\rangle + \alpha_1 |\phi_1\rangle + ... + \alpha_{2n-1} |\phi_{2n-1}\rangle = \sum_{i=0}^{2^n-1} \alpha_i |\phi_i\rangle$$

where the amplitudes a are complex numbers such that $\sum_i |\alpha_i|^2 = 1$, and each $|\phi_i\rangle$ is a basis vector of the Hilbert space. The probability of the machine being at a basis state $|\phi_i\rangle$ at any time is $|\alpha_i|^2$, however reading the machines state at any time will invalidate the rest of the computation since this state will then be projected to the observed basis vector $|\phi_i\rangle$. For an example of this, suppose we have the quantum state: $|\psi\rangle = \frac{1}{\sqrt{3}} |0\rangle + \sqrt{\frac{2}{3}} |1\rangle$, then the probability of measuring the basis state $|0\rangle$ is equal to $\frac{1}{\sqrt{3}}^2 = \frac{1}{3}$, and the probability of measuring the basis state $|1\rangle$ is equal to $\sqrt{\frac{2}{3}}^2 = \frac{2}{3}$. We also have the *conjugate* state, which can be represented in *bra* notation: $\langle\psi|$. If we have the quantum state $|\psi\rangle = \sum_i \alpha_i |\phi_i\rangle$, then the conjugate state is: $\langle\psi| = \sum_i \alpha_i^* \langle\phi_i| = \begin{pmatrix} \alpha_0^* & \alpha_1^* & ... & \alpha_{2n-1}^* \end{pmatrix}$ i.e. a row vector of the *complex conjugates* of the elements of $|\psi\rangle$ with the complex conjugate of a complex number $a + bi$ being $a - bi$.

4.1.3 Transformations on Quantum States

Machine states can be described in vector format by taking all the coefficients as such:

$$|\psi\rangle = \alpha_0 |\phi_0\rangle + \alpha_1 |\phi_1\rangle \equiv \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$$

We can then apply a transformation to this state such that the coefficients become α_0' and α_1':

$$|\psi'\rangle = \alpha_0' |\phi_0\rangle + \alpha_1' |\phi_1\rangle \equiv \begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix}$$

This transformation can be expressed as

$$|\psi'\rangle = U |\psi\rangle$$

where U is a unitary $m \times m$ matrix with m being the size of the state, which in our case is 2. Therefore, our transformation can be represented as:

$$\begin{pmatrix} \alpha_0' \\ \alpha_1' \end{pmatrix} = \begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix},$$

and we have that $\alpha_0' = u_{11}\alpha_0 + u_{12}\alpha_1$ and $\alpha_1' = u_{21}\alpha_0 + u_{22}\alpha_1$

4.1.4 Bell States and Entanglement

Bell states are quantum states where qubits are entangled, i.e. if we measure one of the qubits, we know what the other one will be without having to measure it. An example of such a state is:

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Entangled states cannot be created from independent qubits, instead they have to be programmed using quantum circuits. Consider the composite state:

$$a_0 b_0 |00\rangle + a_0 b_1 |01\rangle + a_1 b_0 |10\rangle + a_1 b_1 |11\rangle ,$$

composed of two independent qubits: $a_0 |0\rangle + a_1 |1\rangle$ and $b_0 |0\rangle + b_1 |1\rangle$. For this to be a bell state, we require the only possible states when measured to be $|00\rangle$ and $|11\rangle$, therefore $a_0 b_1$ and $a_1 b_0$ both have to equal 0, however this leads to a contradiction in classical mechanics, because whichever configuration we choose for $a_0 b_1$ and $a_1 b_0$ to equal 0, means that $a_1 b_1$ and $a_0 b_0$ are also equal to 0, but in reality, these are equal to $\frac{1}{\sqrt{2}}$. This is where the properties of quantum mechanics start to differ from classical mechanics. Entanglement is what makes quantum computers truly powerful, as otherwise, the same computation can be simulated on classical computers. Creating entangled qubits is very hard to do however, and at time of writing, the largest number of entangled qubits in a physical quantum computer is 20 [26].

4.2 Quantum Logic Gates

Using the knowledge of quantum state transformations, we can create quantum logic gates in an analagous way to classical logic gates, for example, the quantum version of a NOT gate must simply flip the input qubits such that an input qubit: $\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$ is flipped to $\begin{pmatrix} \alpha_1 \\ \alpha_0 \end{pmatrix}$. We can see simply that this transformation can be represented as the matrix: $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. There are various other quantum logic gates which are very useful in a wide range of different quantum algorithms, however all quantum gates must be reversible, i.e. the matrix that describes them must be unitary. For a matrix to be unitary, it must be equal to the identity matrix when multiplied with its *hermitian conjugate*. The hermitian conjugate of a matrix U, denoted as $U^\dagger$, can be calculated as the original matrix transposed with each element being the complex conjugate of its original entry. For an example, we can look at a commonly used gate in quantum logic circuits, the Pauli-Y gate: $\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$. The transpose of this matrix is: $\begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix}$, with the complex conjugate of this being equal to our original matrix: $\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$. The reason why quantum logic gates must be unitary and therefore reversible, is that for a quantum state to be valid, it must be *normalised*,

which means that the inner product of itself and it's own conjugate state must be equal to 1. For example, for the state $|\psi\rangle = \begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix}$, its inner product defined as $\langle\psi'|\psi\rangle = \alpha_0^*\alpha_0 + \alpha_1^*\alpha_1$ must be equal to 1. Now consider a transformation $|\psi'\rangle = U|\psi\rangle$, for the system to be normalised, we must have $\langle\psi|\psi\rangle = 1$ and $\langle\psi'|\psi'\rangle = 1$. This transformation can also be described in component notation as such:

$$|\psi'\rangle = \begin{pmatrix}\alpha_0'\\ \alpha_1'\end{pmatrix} = \begin{pmatrix}u_{11} & u_{12}\\ u_{21} & u_{22}\end{pmatrix}\begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix}$$

and

$$\langle\psi'| = \begin{pmatrix}\alpha_0'^* & \alpha_1'^*\end{pmatrix} = \begin{pmatrix}\alpha_0^* & \alpha_1^*\end{pmatrix}\begin{pmatrix}u_{11}^* & u_{21}^*\\ u_{12}^* & u_{22}^*\end{pmatrix},$$

therefore we have $\langle\psi'| = \langle\psi|U^\dagger$, which leads to:

$$\langle\psi'|\psi'\rangle = \langle\psi|U^\dagger U|\psi\rangle = \langle\psi|\psi\rangle,$$

therefore, $U^\dagger U$ must be equal to the identity matrix I.

4.2.1 Hadamard Transform

The Hadamard transform is a very commonly used quantum gate due to its useful properties: when the Hadamard transform is applied to a qubit, the qubit then has an equal chance of being measured as a $|0\rangle$ or a $|1\rangle$, regardless of the initial state of the qubit. The matrix for this transformation is:

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

We can prove that for a qubit in any initial state, when this transform is applied, will become an equal superposition of all base states. Suppose we have our initial qubit in state $|0\rangle$, i.e. its coefficients are: $\begin{pmatrix}1\\ 0\end{pmatrix}$, then when we apply the transform we get:

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}1\\ 0\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\ 1\end{pmatrix}$$

The same can be shown for initial state $|1\rangle$:

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}0\\ 1\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\ -1\end{pmatrix}$$

The negative sign does not mean that this vector is different than our previous vector, as we can only measure the magnitude of the vector when taking a measurement. Therefore, when measuring both of these states it appears as if they are the same. The next section will describe

how we can distinguish between these vectors. For an n qubit system, we can apply n Hadamard transform gates in parallel to gain a balanced superposition of all possible states.

4.2.2 Phase Shift Gates

There are a group of quantum logic gates which can be grouped under the set of "phase shift gates". These gates perform a rotation around a certain axis when applied to a qubit. To understand how these rotations work, it is helpful to think of qubit states as being a vector in a 3 dimensional unit sphere, where the typical notation:

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$$

can be represented as:

$$\begin{aligned} |\psi\rangle &= e^{i\gamma}(\cos\frac{\theta}{2}|0\rangle + (\cos\phi + i\sin\phi)\sin\frac{\theta}{2}|1\rangle) \\ &= e^{i\gamma}(\cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle), \end{aligned}$$

Where θ and ψ are the relative phases, and γ is the global phase of the vector. Figure 10 describes this. If two vectors are only differing in their global phase, then they are indistinguishable when

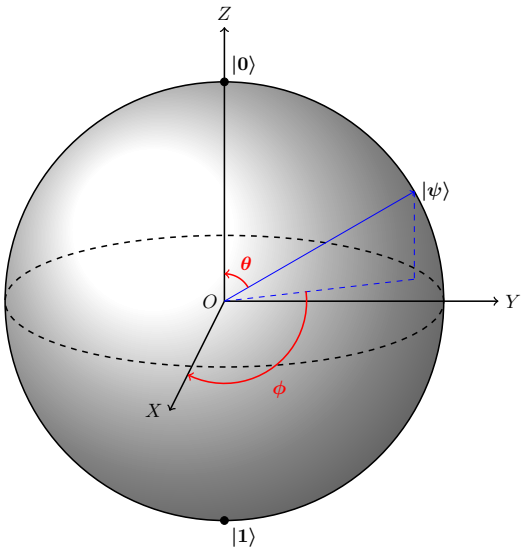

Figure 10: Vector Representation of Qubit Position

measuring, i.e. the two vectors below are the same when measured.

$$\frac{-1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \equiv \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

When a Hadamard transform is applied, then their relative phases become different and hence the measurements become $|0\rangle$ and $|1\rangle$ respectively.

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle, \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

We can now define phase shift gates as the gates which transform an input by rotating it along a certain axis. The base matrices for the rotations in X,Y and Z axis are defined below:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} Y = \begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix} Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

These are often referred to as Pauli-X Pauli-Y and Pauli-Z gates. More generally, we can have a rotation of a certain angle in the Z axis by using the matrix:

$$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix},$$

which is commonly referred to the global phase shift gate, denoted as R_θ.

4.3 Quantum Circuits

Quantum gates can be applied in a circuit in the same way that classical logic gates can. We use symbols to denote quantum gates as described in Figures 11, 12, 13 and 14, where the lines denote inputs and outputs.

Figure 11: Quantum circuit describing Hadamard gate

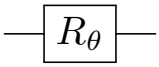

Figure 12: Quantum circuit describing phase shift gate by arbitrary angle θ

Figure 13: Quantum circuit describing measurement gate

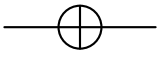

Figure 14: Quantum circuit describing NOT gate

4.3.1 Control Gates and Entanglement

More commonly in quantum circuits, we use a controlled version of gates. An example of a control gate is the controlled not (CNOT) gate. This gate flips the target qubit if the control qubit is 1, otherwise it leaves it. The controlled not gate is described in Figure 15. The controlled

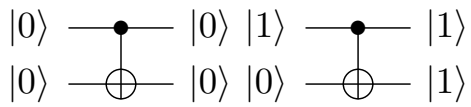

Figure 15: Quantum circuits describing the effect of controlled not gate on different inputs

not gate is an example of a multi-input gate, we can mix multi-input gates and single input gates in circuits, and this is required to create *entangled* qubits which are useful in many quantum circuits. Consider the circuit in Figure 16. We have a Hadamard gate acting on a single input, and a controlled not gate which takes the output from the Hadamard transform as its control input. To see exactly what is happening in this circuit, we can split the circuit up, as described

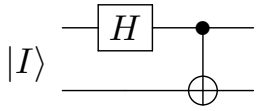

Figure 16: Quantum circuit describing how to create entangled states

in Figure 17 When the circuit is split up in this way, we can describe what transformations are

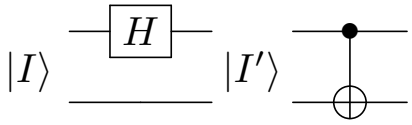

Figure 17: Quantum circuit for entanglement split into components

happening at each stage. The first Hadamard transform is only applied to one qubit. Since we have a two qubit input, the second qubit is left unchanged, and the output is only affected by the Hadamard transform on the first qubit. The transformation that a Hadamard gate performs on one qubit is as follows

$$|0\rangle \longrightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$
$$|1\rangle \longrightarrow \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

When we have a two qubit input, the transformation is as follows:

$$|00\rangle \longrightarrow \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$$
$$|01\rangle \longrightarrow \frac{1}{\sqrt{2}}(|01\rangle + |11\rangle)$$
$$|10\rangle \longrightarrow \frac{1}{\sqrt{2}}(|00\rangle - |10\rangle)$$
$$|11\rangle \longrightarrow \frac{1}{\sqrt{2}}(|01\rangle - |11\rangle)$$

The matrix for this transformation is:

$$H_1 = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}$$

Since this is the output for the first part of the circuit, the input $|I'\rangle$ it is the input to the next part of the circuit. We can then obtain the matrix for the whole circuit, M, by multiplying this first matrix with the matrix for a controlled not gate as follows:

$$M = N_c \times H_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{pmatrix}$$

This matrix is then the transformation for the whole circuit, and transforms states in the following way:

$$|00\rangle \longrightarrow \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$
$$|01\rangle \longrightarrow \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$
$$|10\rangle \longrightarrow \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$$
$$|11\rangle \longrightarrow \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

This set of output states is called a *bell state*, and means that if we measure one of the output qubits, we know the value of the other without having to measure it, meaning that the qubits are entangled. Consider the first state:

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

If we split this into all possibilities of each qubit with probabilities for each qubit in each state as such:

$$(a\,|0\rangle + b\,|1\rangle) \times (c\,|0\rangle + d\,|1\rangle),$$

then we can represent all possible output states as

$$ac\,|00\rangle\,, ad\,|01\rangle\,, bc\,|10\rangle\,, bd\,|11\rangle$$

That means, for our state to be possible, we must have $ad, bc = 0$ and $ac, bd = \frac{1}{\sqrt{2}}$, so our qubits are entangled.

4.3.2 Toffoli Gate

Typically, classical logic gates such as NAND are irreversible, so they have no quantum representation, however there exists a quantum gate known as the Toffoli gate, invented by Tommaso Toffoli, which can simulate classical logic in a quantum circuit. The toffoli gate takes 3 qubits as input, and outputs 3 qubits. Two of the input qubits are control qubits that are unnaffected by the action of the Toffoli gate, the third qubit is the *target* qubit, which is flipped if both control qubits are in state $|1\rangle$, otherwise it is left alone. The Toffoli gate can be used to simulate NAND gates in quantum circuits, which can then be used in larger circuits to simulate classical arithmetic operations. The truth table, transformation matrix and circuit diagram are shown in Table 2, Figure 18 and Figure 19 respectively.

Input			Output
$\|a\rangle$	$\|b\rangle$	$\|c\rangle$	$\|c \oplus ab\rangle$
$\|0\rangle$	$\|0\rangle$	$\|0\rangle$	$\|000\rangle$
$\|0\rangle$	$\|0\rangle$	$\|1\rangle$	$\|001\rangle$
$\|0\rangle$	$\|1\rangle$	$\|0\rangle$	$\|010\rangle$
$\|0\rangle$	$\|1\rangle$	$\|1\rangle$	$\|011\rangle$
$\|1\rangle$	$\|0\rangle$	$\|0\rangle$	$\|100\rangle$
$\|1\rangle$	$\|0\rangle$	$\|1\rangle$	$\|101\rangle$
$\|1\rangle$	$\|1\rangle$	$\|0\rangle$	$\|111\rangle$
$\|1\rangle$	$\|1\rangle$	$\|1\rangle$	$\|110\rangle$

Table 2: Truth table for Toffoli transformation

4.4 Quantum Fourier Transform

Another key example of a quantum logic circuit which is used in many quantum algorithms is the Quantum Fourier transform (QFT). This is a quantum version of the classical Fast Fourier transform (FFT), and can be recreated in a quantum circuit with a collection of gates. The QFT allows us to find the underlying periodic behaviour of a function, and so it is useful in many quantum algorithms.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 18: Matrix representing Toffoli transformation

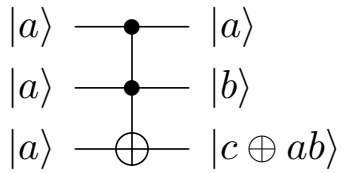

Figure 19: Circuit representing Toffoli transformation

4.4.1 Discrete Fourier Transform

To understand how the Quantum Fourier transform allows us to find the period of a function, we can look at the classical discrete Fourier transform. The discrete Fourier transform essentially allows us to take a finite sequence of equally spaced samples of an input function with high noise (irregularities from the base function) and find the underlying periodic behaviour. This transformation is useful because it can break down a seemingly random signal composed of many different frequency waves into its component parts. The transformation is done using properties of sinusoidal waves and Euler's formula. For a simple example of the discrete Fourier transform, we can apply it to a sample of a sine wave with frequency 1Hz and amplitude 1: We can take

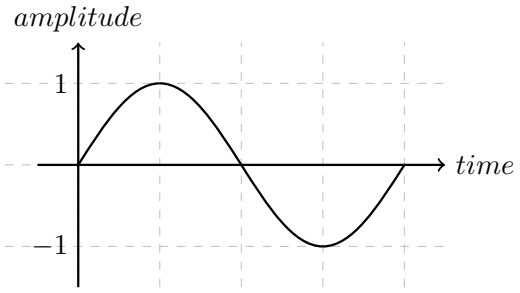

Figure 20: Sine wave with frequency 1Hz and amplitude 1

samples of this function's amplitude at equally spaced points. For this example we use a sampling frequency of 8 Hz: The eight values for amplitude we get are displayed in Table 3 Using these values and the formula for the discrete Fourier transform, we can work out the frequency bins for each value by substituting in the values for k, n and x_i. When this is done, we get values for

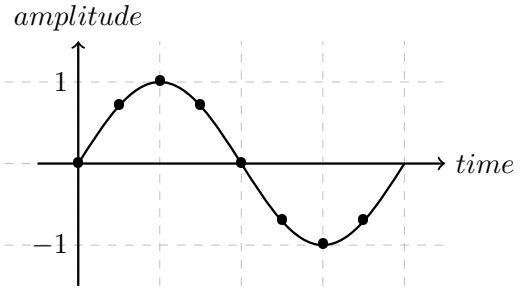

Figure 21: Sine wave sampled at 8Hz

	amplitude
x_0	0
x_1	0.707
x_2	1
x_3	0.707
x_4	0
x_5	-0.707
x_6	-1
x_7	-0.707

Table 3: Amplitudes of samples

X_k as shown in Table 4 From this we can see that the only non-zero values are those for X_1 and

	Fourier Coefficient
X_0	0
X_1	0 - $4i$
X_2	0
X_3	0
X_4	0
X_5	0
X_6	0
X_7	0 + $4i$

Table 4: Fourier Coefficients of samples

X_7. We choose to take the magnitude of these rather than the full complex number for reasons that will be shown later. We can plot this as a frequency against magnitude graph as shown in Figure 22 The Nyquist limit is the highest frequency that can be coded at a given sampling rate to be able to fully reconstruct the original signal, and it is equal to $\frac{N}{2}$ [27]. Due to this, we remove all values above the Nyquist limit, and double all values below the Nyquist limit to account for the removed values. This leaves us with a result for only X_1, being a magnitude of 2*4 = 8. To get the amplitude of the original sample, we divide this amplitude by the number of samples. In this case we have $\frac{8}{8}$ which tells us that the amplitude of the original sample was 1. To find the frequency of the original sample, we have to look at the value of k. In this case it

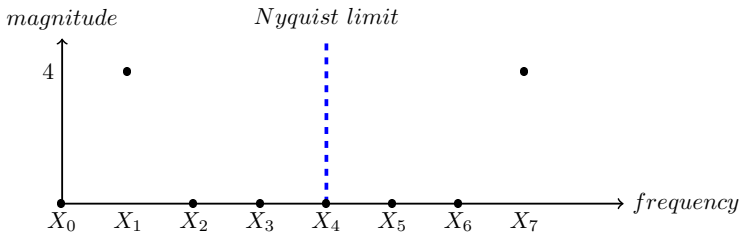

Figure 22: Frequency against magnitude plot for sample

is 1, so our frequency was 1 Hz. We can also calculate the phase that this sinusoidal wave was shifted compared to a normal cosine wave. In order to do this we need to look at the complex number we obtained for X_k, which in this case was $0 - 8i$. If we plot this on a complex plane we get an angle from the positive real axis to the plot, as shown in Figure 23 The angle θ is equal to

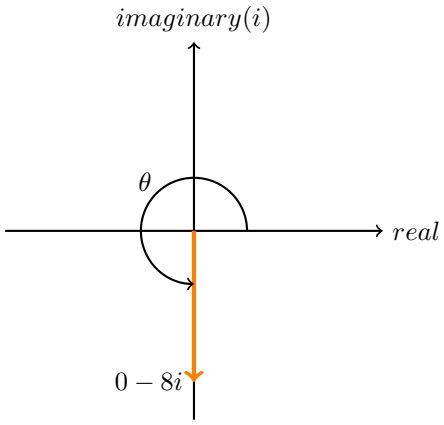

Figure 23: Angle between positive real axis and sample

$\frac{3\pi}{2}$, which is our phase shift from a normal cosine wave. This method can be used for any type of sample, and the results will be normalised against a cosine wave, so we can use it for statistical analysis of the sample.

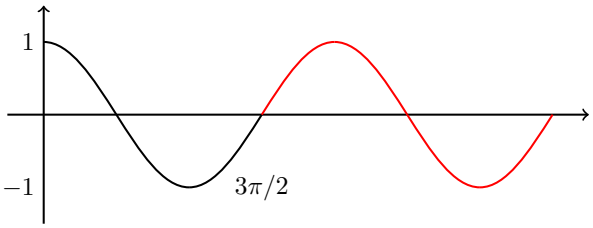

Figure 24: Phase shift of sample from cosine wave

4.4.2 Applying the Quantum Fourier Transform

The Quantum Fourier transform (QFT) works in a similar way as the discrete fourier transform, with the one major difference being that the QFT takes quantum states as inputs and outputs. It takes the quantum state $|x\rangle = \sum_{j=0}^{N-1} x_j |i\rangle$ and maps it to the quantum state $\sum_{k=0}^{N-1} y_k |k\rangle$ according to the formula:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \cdot e^{2\pi i j k/N}$$

Consider the 2-qubit state $|x\rangle = a_0 |00\rangle + a_1 |01\rangle + a_2 |10\rangle + a_3 |11\rangle$, in this state, then we have $N = 2^2 = 4$. We can apply the QFT to each instance by subsituting in k as 0,1,2,3 respectively as follows:

$$y_0 = \frac{1}{2} \sum_{j=0}^{3} a_j = \frac{1}{2}(a_{00} + a_{01} + a_{10} + a_{11})$$

$$y_1 = \frac{1}{2} \sum_{j=0}^{3} a_j \cdot \omega^j = \frac{1}{2}(a_{00} + a_{01} \cdot \omega + a_{10} \cdot \omega^2 + a_{11} \cdot \omega^3)$$

$$y_2 = \frac{1}{2} \sum_{j=0}^{3} a_j \cdot \omega^2 j = \frac{1}{2}(a_{00} + a_{01} \cdot \omega^2 + a_{10} \cdot \omega^4 + a_{11} \cdot \omega^6)$$

$$y_3 = \frac{1}{2} \sum_{j=0}^{3} a_j \cdot \omega^3 j = \frac{1}{2}(a_{00} + a_{01} \cdot \omega^3 + a_{10} \cdot \omega^6 + a_{11} \cdot \omega^9)$$

These values can be represented in a matrix form as shown below:

$$F = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

4.4.3 Quantum Fourier Transform Circuit

The Quantum Fourier transform can be represented as a quantum circuit, and we can prove that this performs the same transformation that we just obtained by observing the matrix for the transformation of the whole circuit. The QFT for a two qubit circuit is shown in Figure 25. The gate at the end is called a *swap* gate, and simply swaps the order of the qubits. This

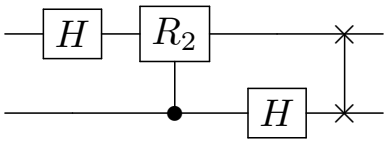

Figure 25: Quantum Fourier Transform for a 2-qubit circuit

transformation is then as follows:

$$|00\rangle \longrightarrow |00\rangle$$
$$|01\rangle \longrightarrow |10\rangle$$
$$|10\rangle \longrightarrow |01\rangle$$
$$|11\rangle \longrightarrow |11\rangle$$

and can be represented in matrix form as:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Other than the swap gate, we have a Hadamard transform acting on the first qubit only, then a controlled phase shift of order 2, then another Hadamard transform acting on the second qubit only. The generalised matrix for the controlled phase shift gate can be represented as:

$$R_k = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & e^{2\pi i/2k} \end{pmatrix}$$

Therefore, the matrix for the R_2 gate is:

$$R_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & i \end{pmatrix}$$

We can then split the circuit into its seperate components as shown in Figure 26, and represent it as a series of matrix multiplications which take the input from $|I\rangle$ to $|A\rangle$, $|B\rangle$, $|C\rangle$, and finally $|O\rangle$ respectively. These matrices can be denoted as H_1, $R_c(2)$, H_2, and *swap* respectively. When

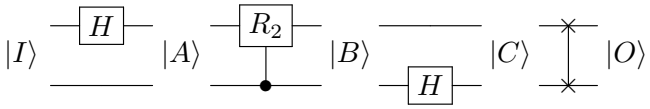

Figure 26: Quantum Fourier Transform for a 2-qubit circuit split into components

we multiply these matrices out:

$$
\begin{aligned}
M &= swap \times H_2 \times R_c(2) \times H_1 \\
&= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix},
\end{aligned}
$$

we can obtain the matrix for the whole circuit as:

$$
\frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}
$$

We can see that indeed this is the same matrix as the one we obtained earlier, proving that this circuit performs the QFT for a two qubit circuit. This circuit can be extended to a circuit with any number of input and output qubits as described in Figure 27, and we can denote it in a larger circuit as a single gate to make things simpler using the notation in Figure 28.

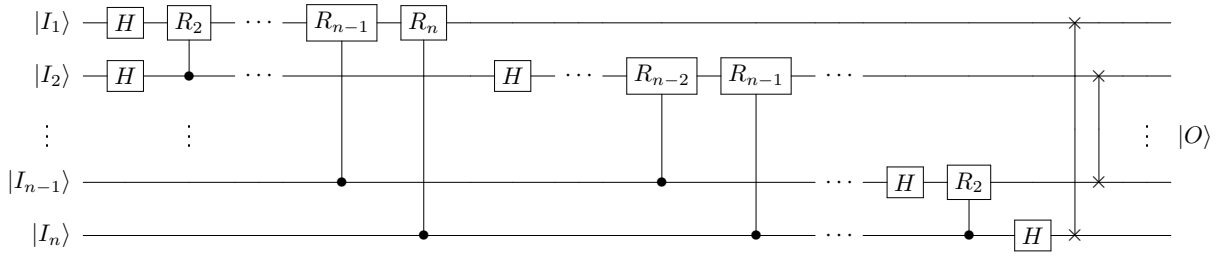

Figure 27: Quantum Fourier Transform for a n-qubit circuit

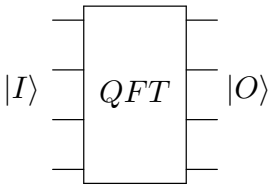

Figure 28: Quantum Fourier Transform gate

4.5 Shor's Algorithms

In his paper: *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer* [5], Peter W. Shor introduced two algorithms which are able to solve the

prime factorisation problem and the discrete logarithm problem respectively in polynomial time. The algorithms both involve reducing the problem of finding either a prime factor or a discrete logarithm into a period finding problem, which can be completed on a quantum computer. The algorithms involve a number of steps which can be computed on a classical computer to avoid unnecessary complexity in the quantum circuit. The hardest part of the computation is calculating the period of the function which the problem can be reduced to.

4.5.1 Shor's Algorithm for Prime Factorisation

Shor's Algorithm for prime factorisation can be summarised in 5 steps. Suppose we have a number N, which has the factors p and q:
1. If N is even, prime or a prime power, there exists classical algorithms to calculate factorisation so exit.
2. Choose a random number $a < N$ and calculate $\gcd(a, N)$, if this is not equal to 1, then we have found a factor of N in a, so exit.
3. Find the period r of the function$f(x) = a^x \mod N$. (This is the quantum aspect of the algorithm)
4. If r is odd, or $a^{r/2} \mod N = -1 \mod N$ then go back to step 1.
5. $\gcd(a^{r/2} \pm 1, N)$ are prime factors of N
Since we already have fast algorithms to calculate the classical parts of Shor's algorithm, this section will describe how we can use the properties of quantum computation to calculate the period of the function $f(x) = a^x \mod N$.

4.5.2 Phase Estimation

For some unitary operation on quantum state: $U|\psi\rangle$, if it is known that the state $|\psi\rangle$ is an *eigenstate* of that operator, i.e. the quantum state does not change when it is operated on by U, then there exists a corresponding complex eigenvalue such that $U|\psi\rangle \equiv e^{2\pi i\phi}|\psi\rangle$, $0 \leq \phi < 1$ [4]. We can design quantum circuits to estimate this value of ϕ to a certain number of decimal places. If we know both the value of U and $|\psi\rangle$, we can estimate ϕ using the circuit described in Figure 29. This circuit involves two registers of qubits, R_1 and R_2. Register 1 contains n qubits which are put into a superposition of all states using a series of parallel Hadamard transforms, i.e. the first register is initially in the state:

$$\frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^m-1} |y\rangle$$

Register 2 contains m controlled U gates, each of which performs the square of the operation before it. The result of this is that the first register is put into a superposition of all values of our function. The state of the first register becomes:

$$\frac{1}{\sqrt{2}}((|0\rangle + e^{2\pi i(2^0\phi}|1\rangle)(|0\rangle + e^{2\pi i(2^1\phi}|1\rangle)(|0\rangle + e^{2\pi i(2^{m-2}\phi}|1\rangle)(|0\rangle + e^{2\pi i(2^{m-1}\phi}|1\rangle))$$

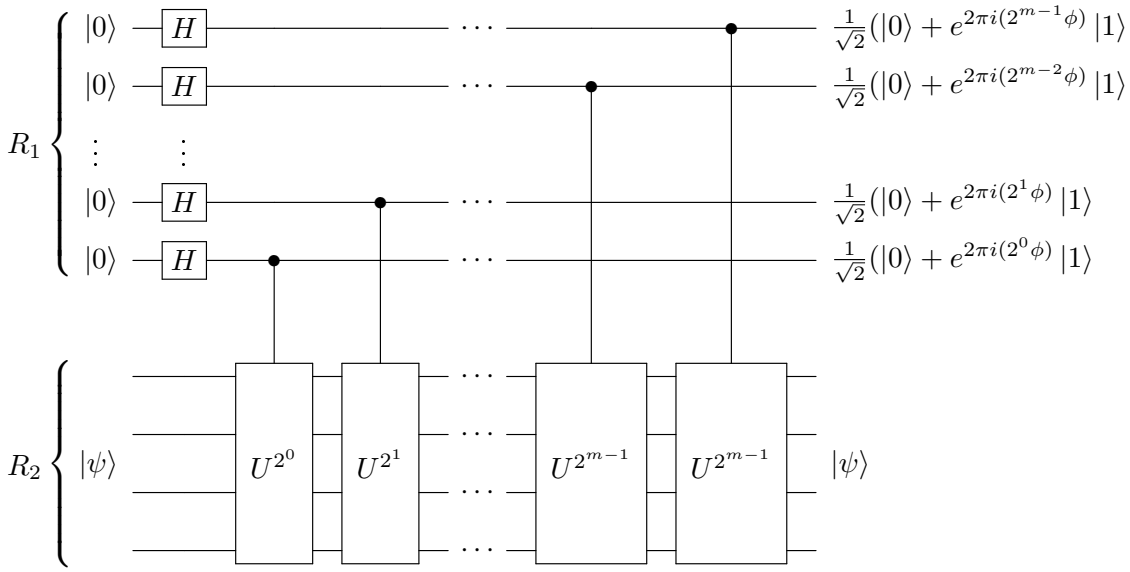

Figure 29: Phase Estimation Circuit [1]

$$= \frac{1}{\sqrt{2^m}} \sum_{y=0}^{2^m-1} e^{2\pi i \phi y} |y\rangle$$

recall that the Quantum Fourier transform can be represented as

$$\sum_{j=0}^{N-1} x_j |i\rangle \longrightarrow \sum_{k=0}^{N-1} y_k |k\rangle ,$$

where

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \cdot e^{2\pi i jk/N}$$

This means that, because $N = 2^m$ in our example, if we apply the *inverse* of the QFT to this output, we get the state $|2^m\phi\rangle$ in our first register of n qubits, and so we can measure the phase. The numbers of qubits needed in registers 1 and 2 are $n = 2\log_2 N$ and $m = \log_2 N$ respectively [28].

4.5.3 Quantum Circuit for Prime Factorisation

The problem for finding the period of a function can be reformulated into a phase estimation problem. If we want to find the period r of a function

$$f(x) = a^x \mod N,$$

and we have a unitary operator which performs:

$$U|y\rangle \longrightarrow |xy \mod N\rangle,$$

which, as discussed in the previous section, can be replaced by its eigenvalue and eigenvector as such:

$$e^{2\pi i\phi}|\psi_s\rangle$$

In fact, there are r eigenvectors with corresponding eigenvalues, so $|\psi_s\rangle$ just denotes one of these eigenvectors. These eigenvalues can be represented as $e^{2\pi is/r}$ [4], which means that if we can measure the phase using our phase estimation circuit, we have a value for $\frac{s}{r}$. We do not need to know the specific eigenvector $|\psi_s\rangle$, as we can just put the system in a superposition of all possible vectors, and thus containing all r eigenvectors. If we run a simulation representing this circuit, we see that the probability distribution forms peaks, these peaks correspond to each eigenvector, so it is more likely that when we take a measurement, we will measure the value of one of these peaks, which will give us a correct value for r. An example of the full circuit needed to factor 21 is described in Figure 30. The next section will describe a practical example of how we can obtain the factors of a number using Shor's algorithm.

4.5.4 Example of Factorisation Using Shor's Algorithm

Using the software LIQUi|> [29], we were able to simulate a quantum circuit which would implement Shor's factorisation algorithm. When our input N was 253, and a value of a as 2, we obtained a measurement of $|0001000001001010\rangle = 4170$. Using the continued fraction algorithm described in Box 5.3 from [4], we can represent $\frac{4170}{2^{16}} = \frac{4170}{65536}$ as:

$$0 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{10 + \cfrac{1}{1 + \cfrac{1}{7 + \cfrac{1}{1 + \frac{1}{2}}}}}}}}}}$$

with values for d and r as described in Table 5. Equation 5.3 in Shor's paper describes how we can obtain the value for r from this set of values. There is at most one fraction d/r with $r < N$ which satisfies the equation:

$$\left|\frac{c}{q} - \frac{d}{r}\right| < \frac{1}{2q},$$

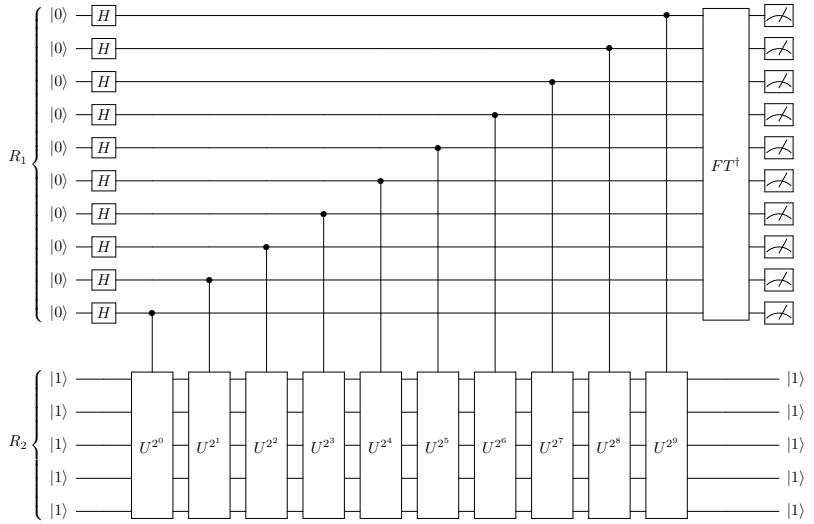

Figure 30: Quantum Circuit to Factorise 21

d	1	1	3	4	7	74	81	641	722	2085
r	15	16	47	63	110	1163	1273	10074	11347	32768

Table 5: Values for d and r obtained from continued fraction algorithm [Box 5.3] [4]

where q is the power of 2 with $n^2 \leq q < 2n^2$, equal to $2^{16} = 65536$, and c is our observed value $= 4170$. Therefore the only values which satisfy this requirement in the table are $d = 7$ and $r = 110$. So we have obtained a value for the period of the function, r, as 110, now we can perform the last two steps in Shor's algorithm to calculate the factors of 253. Since r is even and $a^{r/2} \mod N = 2^{55} \mod 253 = 208 \neq -1 \mod 253$, our value for r is correct. We can then obtain factors for 253 as $gcd(2^{55} \pm 1 \mod 253, 253)$ which are 11 and 23 respectively. We can verify that 11×23 is indeed equal to 253, so we have correctly obtained the prime factors of 253.

4.5.5 Shor's Algorithm for Discrete Logarithms

Shor's paper also describes an algorithm for solving the discrete logarithm problem for prime groups. This algorithm can be extended to solve the discrete logarithm problem for a generic cyclic group G which has some generator g, and a group operation $\odot$, given that the group operation can be applied multiple times efficiently using quantum circuits [30]. For a generic group such as this, we denote the process of applying the group operation a number d of times to an initial element as $x = d[g]$, thus the discrete logarithm problem for a generic group such as this, is to find the discrete logarithm d given g and x. The algorithm below describes how we can solve this problem using a quantum subroutine, taken from [30].

1. Use a quantum circuit to calculate a "good pair", (j, k), on input of a generator g and an

element x of the group G. This good pair is what we can use in the next step to calculate the discrete logarithm from

2. Using a classical algorithm, compute d from this good pair (j, k) using the equivalence:

$$d \equiv \left\lfloor \frac{kq}{2^l} \right\rceil t^{-1} \mod q,$$

where l is the length of the qubit registers in the quantum circuit, q is the order of the group and t is an integer that satisfies:

$$t = \frac{\{jq\}_{2^l} - jq}{2^l} \in \mathbb{Z}$$

4.5.6 Quantum Subroutine to Compute a Good Pair

This section will outline the quantum algorithm described in [30], that computes the "good pair" (j, k) which allow us to find the discrete logarithm d from a group member x and a generator g. We start by finding l, the largest positive integer such that $2^l < q$, where q is the order of the group. The first step is to create two registers of qubits of length l which are initially in an equal superposition, and a third register in the state $|0\rangle$. The state of the system at this point can be viewed as:

$$|\psi\rangle = \frac{1}{2^l} \sum_{a=0}^{2^l-1} \sum_{b=0}^{2^l-1} |a\rangle |b\rangle |0\rangle$$

We then compute the value $[a]g \odot [-b]x$ for all values a and b in parallel, and store the result in the third register, such that the state of the system is now:

$$|\psi\rangle = \frac{1}{2^l} \sum_{a=0}^{2^l-1} \sum_{b=0}^{2^l-1} |a, b, [a]g \odot [-b]x\rangle$$

This is equal to:

$$|\psi\rangle = \frac{1}{2^l} \sum_{a=0}^{2^l-1} \sum_{b=0}^{2^l-1} |a, b, [a - bd]g\rangle$$

Finally, we apply two Quantum Fourier transforms to the first two registers, which puts the system in a final state:

$$\frac{1}{2^{2l}} \sum_{a=0}^{2^l-1} \sum_{b=0}^{2^l-1} \sum_{j=0}^{2^l-1} \sum_{k=0}^{2^l-1} e^{2\pi i(aj+bk)/2^l} |j, k, [a - bd]g\rangle$$

We can then observe the system to obtain values for (j, k) and $[e]g$. This quantum algorithm for a general discrete logarithm problem can be represented by a quantum circuit as described in Figure 31.

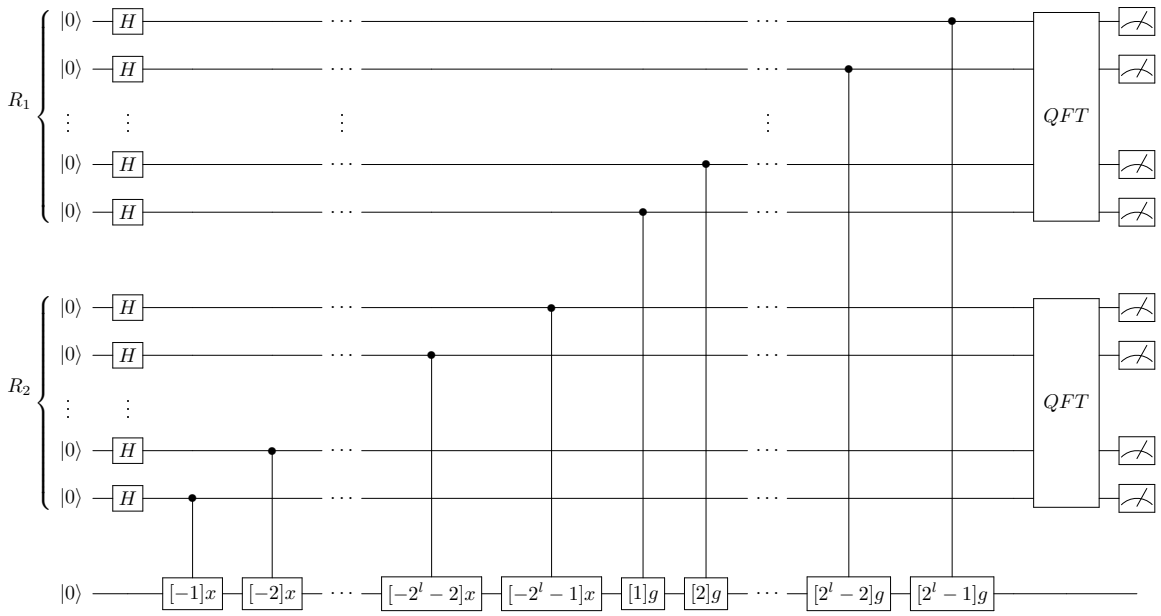

Figure 31: General Discrete Logarithm Quantum Circuit

4.6 Shor's Algorithm Adapted for the Elliptic Curve Discrete Logarithm Problem

Roetteler, Naehrig, Svore, and Lauter [2] describe a quantum circuit adapted for solving the discrete logarithm problem in elliptic curves, which is shown in Figure 32. This high-level circuit relies on being able to calculate subsequent controlled point additions on a general elliptic curve using quantum circuits, and will be analogous to the circuit for computing the discrete logarithm problem for hyperelliptic curves, with the only difference being the implementation of the repeated group operations. The paper also describes how these point additions can be implemented for elliptic curves using basic quantum circuits. The circuits use combinations of modular additions, multiplications and inversions to calculate point additions. The Toffoli gate costs for these "base" operations are shown in Table 6. Using this, and the algorithms for hyperelliptic curve arithmetic, we are able to calculate the number of Toffoli gates required in a circuit which solves the discrete logarithm problem for a hyperelliptic curve. The total number of qubits needed in the circuit for elliptic curves is:

$$9n + 2\lceil \log_2(n) \rceil + 10,$$

where n is the bit-size of the inputs. This is calculated from the maximum required qubit size for modular arithmetic operation, which is the inversion at $7n + 2\lceil \log_2(n) \rceil + 9$, in addition to a qubit required for the control qubit of the overall operation, and $2n$ qubits which are required

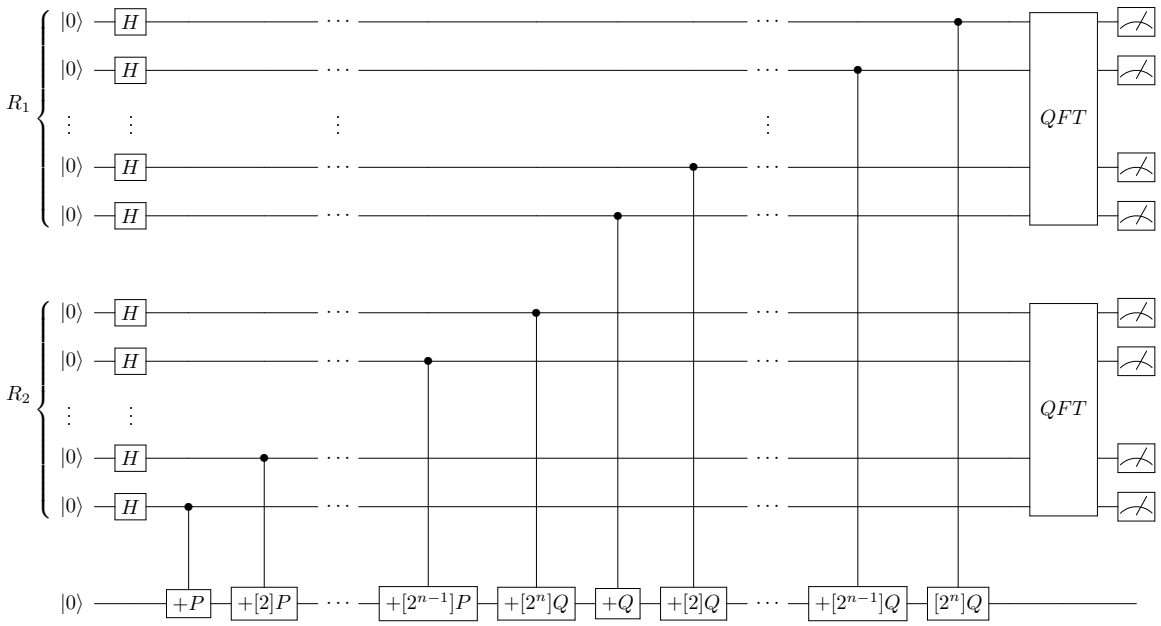

Figure 32: Quantum Circuit to Solve Elliptic Curve Discrete Logarithm Problem [2]

Modular Arithmetic Circuit	Number of Qubits		Number of Toffoli Gates
•	Total	Ancillas	•
add_const_modp	$2n$	n	$16n\log_2(n) - 26.9n$
sub_const_modp	$2n$	n	$16n\log_2(n) - 26.9n$
ctrl_add_const_modp	$2n+1$	n	$16n\log_2(n) - 26.9n$
ctrl_sub_const_modp	$2n+1$	n	$16n\log_2(n) - 26.9n$
ctrl_sub_modp	$2n+4$	3	$16n\log_2(n) - 23.8n$
ctrl_neg_modp	$n+3$	2	$8n\log_2(n) - 14.5n$
mul_modp (dbl/add)	$3n+2$	2	$32n^2\log_2(n) - 59.4n^2$
mul_modp (Montgomery)	$5n+4$	$2n+4$	$16n^2\log_2(n) - 26.3n^2$
squ_modp (dbl/add)	$2n+3$	3	$32n^2\log_2(n) - 59.4n^2$
squ_modp (Montgomery)	$4n+5$	$2n+5$	$16n^2\log_2(n) - 26.3n^2$
inv_modp	$7n + 2\lceil\log_2(n)\rceil + 9$	$5n + 2\lceil\log_2(n)\rceil + 9$	$32n^2\log_2(n)$

Table 6: Costs for modular arithmetic in quantum circuits in terms of n, the bit-size of the prime which the group is defined over [2]

for holding the intermediate results of the inversion operation. Roetteler, Naehrig, Svore, and Lauter estimate the total number of Toffoli gates required for the elliptic curve discrete logarithm problem is in the order

$$(448\log_2(n) + 4090)n^3$$

This result is found by calculating the number of gates required for one point addition, which contains 4 inversions, 2 squarers and 4 multipliers:

$$224n^2 \log_2(n) + 2045n^2$$

and multiplying by $2n$, which is the number of consecutive point additions that are required. Using this method, we can calculate the number of required qubits and Toffoli gates required to solve the hyperelliptic curve discrete logarithm problem, the results of which are displayed in the next section.

5 Methods

This section will discuss the various algorithms for hyperelliptic curve arithemtic, and calculate bounds for the gate costs for the quantum circuits in each case, to provide estimates for how large a quantum circuit is needed to break hyperelliptic curve cryptography.

5.1 Quantum Circuit for Shor's Algorithm on Hyperelliptic Curves

Using the same principle idea as the circuit for elliptic curves described in Figure 32, we can construct a circuit which, given an original divisor D_p of a hyperelliptic curve, and the result of some scalar multiplication on that divisor, D_q. This circuit contains two registers, intially in a superposition due to the parallel Hadamard transform. It then implements a series of scalar divisor multiplications, analogous to the elliptic curve case, and then performs a Quantum Fourier transform on the two registers. As stated in [2], if we take m, the scalar integer which we apply to the divisor, in its binary representation, we can eliminate the need for divisor doublings by precomputing all n 2-power multiples of our original divisor, D_p, which has the advantage that all these doubling operations can be performed on a classical computer and the quantum circuit is only required to do the generic divisor addition, reducing the complexity of the overall operation. This circuit is described in Figure 33.

5.1.1 Quantum Circuit for Divisor Addition

Cantor's algorithm, described in Algorithm 3, can be rewritten explicitly for genus 2 curves, as shown in Algorithm 4, which has complexity $I + 22M + 3S$, to show each modular operation individually. This can then be used to construct a quantum circuit, comprised of reversible operations, which performs this algorithm. The modular arithmetic operations can be completed using the circuits described in [2]. Using the complexity of the algorithm, and the estimates for circuit costs, as shown in Table 6, we can calculate a rough bound for the number of low level gates required to construct this circuit. We can then get an estimate for the total number of gates required in the whole circuit for Shor's Algorithm applied to Hyperelliptic Curves, the results of which will be displayed in the next section.

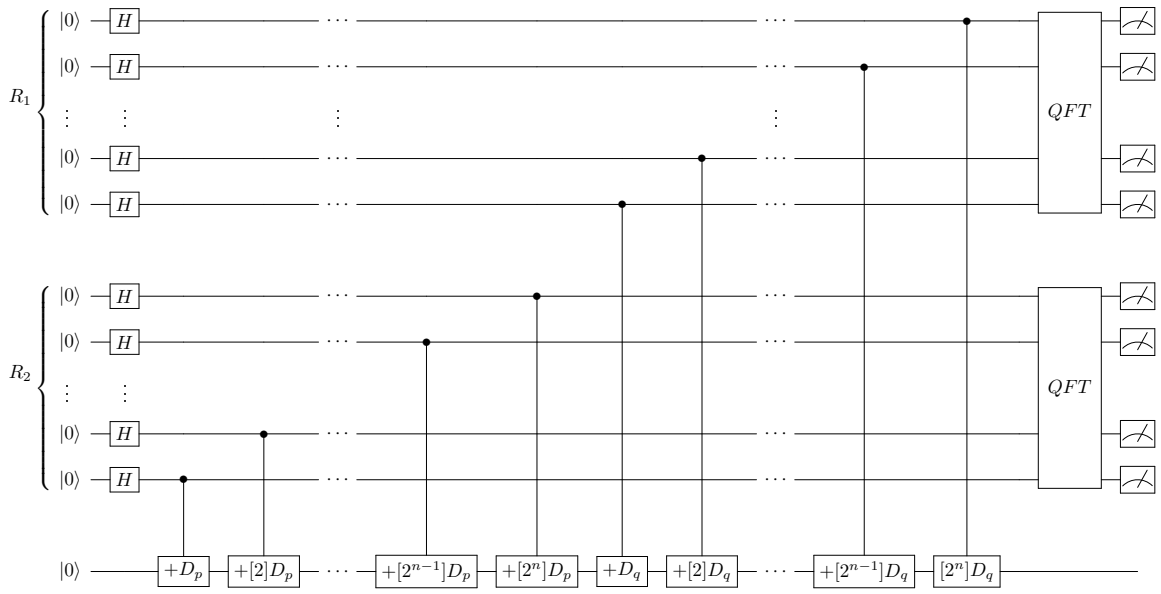

Figure 33: Quantum circuit for applying Shor's algorithm on hyperelliptic curves

5.2 Alternative Hyperelliptic Curve Arithmetic Algorithms

Cantor's algorithm is general and holds for hyperelliptic curves of any genus over any field. There are however, algorithms which use other techniques to the most costly operation, inversion. An example of this is the algorithm for addition in *projective coordinates* over an odd characteristic field. Projective coordinates have a third coordinate, along with x and y, conventionally named z. z is typically constant, and we can represent a divisor in projective coordinates as

$$[x^2 + \frac{u_1}{z}x + \frac{u_0}{z}, \frac{v_1}{z}x + \frac{v_0}{z}]$$

From this, we can define an inversion-free algorithm, as described in [19], with complexity $47M + 4S$. We can convert the coordinates from affine (classical representation) to projective and vice versa on classical computers as preprocessing and postprocessing, to reduce the complexity of the quantum circuit. Similarly to projective coordinates, we can define a set of coordinates called *new coordinates*. In this case, the denominators (z values) differ for u and v. A divisor in new coordinates can be represented by:

$$[x^2 + \frac{u_1}{z_1^2}x + \frac{u_0}{z_1^2}, \frac{v_1}{z_1^3 z_2}x + \frac{v_0}{z_1^3 z_2}],$$

where z_1 and z_2 are the two z values. The complexity of addition in new coordinates is $47M+7S$. We can also combine different sets of coordinates and define algorithms for arithmetic operations when we have mixed inputs. The costs for each combination of affine, projective and new

Algorithm 4 Hyperelliptic Curve Addition (Genus = 2, $\deg(u_1) = \deg(u_2) = 2$, adapted from [19]

Input: Two divisors in mumford representation: $[u_1, v_1], [u_2, v_2]$, with $u_i = x^2 + u_{i1}x + u_{i0}$, and $v_i = v_{i1}x + v_{i0}$ from curve with equation $y^2 + h(x)y = f(x)$
Output: The divisor result $[u', v'] = [u_1, v_1] \oplus [u_2, v_2]$

1: $z_1 \longleftarrow u_{11} - u_{21}$
2: $z_2 \longleftarrow u_{20} - u_{10}$
3: $z_3 \longleftarrow u_{11}z_1 + z_2$
4: $r \longleftarrow z_2 z_3 + z_1^2 u_{10}$
5: $w_0 \longleftarrow v_{10} - v_{20}$
6: $w_1 \longleftarrow v_{11} - v_{21}$
7: $w_2 \longleftarrow z_3 w_0$
8: $w_3 \longleftarrow z_1 w_1$
9: $s'_1 \longleftarrow (z_3 + z_1)(w_0 + w_1) - w_2 - w_3(1 + u_{11})$
10: $s'_0 \longleftarrow w_2 - u_{10}w_3$
11: $w_1 \longleftarrow r{s'_1}^{-1}$
12: $w_2 \longleftarrow rw_1$
13: $w_3 \longleftarrow {s'}_1^2 w_1$
14: $w_4 \longleftarrow rw_2$
15: $w_5 \longleftarrow w_4^2$
16: $s''_0 \longleftarrow s'_0 w_2$
17: $l'_2 \longleftarrow u_{21} + s''_0$
18: $l'_1 \longleftarrow u_{21}s''_0 + u_{20}$
19: $l'_0 \longleftarrow u_{20}s''_0$
20: $u'_0 \longleftarrow (s''_0 - u11)(s''_0 - z_1 + h_2 w_4) - u10$
21: $u'_0 \longleftarrow u'_0 + l'_1 + (h_1 + 2v_{21})w_4 + 2u_{21} + z_1 - f_4)w_5$
22: $u'_1 \longleftarrow 2s''_0 - z_1 + h_2 w_4 - w_5$
23: $w_1 \longleftarrow l'_2 - u'_1$
24: $w_2 \longleftarrow u'_1 w_1 + u'_0 - l'_1$
25: $v'_1 \longleftarrow w_2 w_3 - v_{21} - h_1 + h_2 u'_1$
26: $w_2 \longleftarrow u'_0 w_1 - l'_0$
27: $v'_0 \longleftarrow w_2 w_3 - v_{20} - h_0 + h_2 u'_0$
28: **return** $[u', v'] = (x^2 + u'_1 x + u'_0), (v'_1 x + v'_0)$

coordinates are displayed in Table 7. There also exists Montgomery form arithmetic, analogous to the case for elliptic curves which can be obtained by using Kummer surfaces [31]. At present time of writing, this is the leading type of arithmetic in terms of complexity, and out performs equivalently secure elliptic curves [22]. It involves converting the coordinates to their equivalent representation on a Kummer surface, denoted as $k(D_i)$, where D_i is a divisor, which requires 16 multiplication operations. When this is added to the number of operations required for the actual algorithm ($15M + 2S$), we get a result for the total number of required operations as $31M + 2S$. Further investigation would be required to determine whether we could implement the conversion as pre processing to the quantum circuit, thus saving arithmetic operations.

Operation	Complexity
$N + N = P$	$51M + 7S$
$N + P = P$	$51M + 4S$
$N + N = N$	$47M + 7S$
$N + P = N$	$48M + 4S$
$P + P = P$	$47M + 4S$
$P + P = N$	$44M + 4S$
$A + N = P$	$40M + 5S$
$A + P = P$	$40M + 3S$
$A + N = N$	$36M + 5S$
$A + P = N$	$37M + 3S$
$A + A = A$	$I + 22M + 3S$
$k(A) + k(A) = k(A)$	$31M + 2S(15M + 2S)$

Table 7: Arithmetic costs for different coordinate systems

6 Results

As described in [2], to calculate exact gate costs would require simulation of the quantum circuits involved, since different underlying bit patterns of the prime p which the field is defined over give rise to different quantum circuits, so the actual estimation will be a interpolation based on points gathered for specific values of p. This does not affect the leading coefficient of the estimation, however, so we are able to provide leading coefficient estimates for each of the algorithms described in Table 7, and the actual estimate will be bounded by this leading order coefficient. The results of this are displayed in Table 8. Unsuprisingly, the cost for kummer surface arithmetic is the lowest, and the cost for the affine coordinate addition was the highest, but this is purely focusing on the complexity of the quantum aspect of the algorithm. The pre and post processing are not taken into account, which can affect the time complexity for the whole algorithm drastically.

6.1 Comparison of results

As shown in Table 8, the gate costs when using projective and new coordinates are larger than affine coordinates, this being due to the large amount of extra multiplication and subtraction operations required in the algorithms. Usually in classical computing, these models would provide a much more efficient implementation, however due to the complex natures of the circuits required to implement the reversible operations in quantum computing, they provide a more inefficient implmentation. They do, however, require fewer total qubits and so a tradeoff is presented between hardware scale and circuit size. The two results for both variations of the Kummer surface representation are shown at the bottom of the table, with the representation containing the conversion from affine to kummer having a larger overall estimate than the normal affine coordinate representation, but with fewer required qubits. The second variation where the conversion is pre computed has a lower overall gate cost and a lower number of required qubits, however, as

Operation	Toffoli Gates for Addition	Gates for Shor's Algorithm	Qubits for Addition	Qubits for Shor's Algorithm
$N+N=P$	$1632n^2\log_2 n$	$3264n^3\log_2 n$	$5n+4$	$9n+5$
$N+P=P$	$1632n^2\log_2 n$	$3264n^3\log_2 n$	$5n+4$	$9n+5$
$N+N=N$	$1504n^2\log_2 n$	$3008n^3\log_2 n$	$5n+4$	$9n+5$
$N+P=N$	$1536n^2\log_2 n$	$3072n^3\log_2 n$	$5n+4$	$9n+5$
$P+P=P$	$1504n^2\log_2 n$	$3008n^3\log_2 n$	$5n+4$	$9n+5$
$P+P=N$	$1408n^2\log_2 n$	$2816n^3\log_2 n$	$5n+4$	$9n+5$
$A+N=P$	$1280n^2\log_2 n$	$2560n^3\log_2 n$	$5n+4$	$9n+5$
$A+P=P$	$1280n^2\log_2 n$	$2560n^3\log_2 n$	$5n+4$	$9n+5$
$A+N=N$	$1152n^2\log_2 n$	$2304n^3\log_2 n$	$5n+4$	$9n+5$
$A+P=N$	$1184n^2\log_2 n$	$2368n^3\log_2 n$	$5n+4$	$9n+5$
$A+A=A$	$736n^2\log_2 n$	$1472n^3\log_2 n$	$7n+2\lceil\log_2 n\rceil+9$	$11n+2\lceil\log_2 n\rceil+10$
$k(A)+k(A)=k(A)[1]$	$992n^2\log_2 n$	$1984n^3\log_2 n$	$5n+4$	$9n+5$
$k(A)+k(A)=k(A)[2]$	$480n^2\log_2 n$	$960n^3\log_2 n$	$5n+4$	$9n+5$

Table 8: Gate and qubit costs for algorithms with regard to n, the bit-size of the prime p for which a hyperelliptic curve is defined over

discussed in the previous section, this may not be feasible to implement. The number of toffoli gates for one divisor addition was calculated by multiplying the number of each type of operation by the gate cost for the equivalent circuit described in [2]. This estimate could then be used to calculate the first coefficient for the polynomial that bounds the overall gates required for Shor's algorithm by multiplying by $2n$, since the controlled divisor addition is iterated $2n$. The number of qubits required can be calculated as the most costly operation in terms of qubits, added to 4n+1, which is the number of ancillary qubits required for the operations. It is 4n in the case of hyperelliptic curves as opposed to the 2n of elliptic curves due to the way the divisor is stored: $[u_1, u_0, v_1, v_0]$.

7 Conclusions

To make a comparison between our results and the results found in [2], we must take into account the fact that an equivalently secure hyperelliptic curve cryptosystem requires approximately half of an elliptic curve cryptosystem's prime in terms of its bit-size [22]. Given this, we can divide our input bit-size by 2 to more accurately compare the complexities of the two different circuits. In terms of n, the bit-size of the prime, we have a result of $448n^3\log_2 n$ for elliptic curves, and $960n^3\log_2 n$ in the case of hyperelliptic curves when using the Kummer surface model, or $1472n^3\log_2 n$ in case of affine coordinates. As shown, the leading integer coefficient is much larger in hyperelliptic curves, this is ultimately due to the more complex nature of hyperelliptic curve arithemtic, with even the most stripped down version requiring 15 multiplications and 2 subtractions, as opposed to the 4 multipliers, 2 squarers and 4 inversions of elliptic curves. Table 9, adapted from [2], compares the circuit size and complexity for various bit-size primes of RSA, elliptic curves and hyperelliptic curves. As shown in the table, although the leading integer coefficient is much larger for hyperelliptic curves, the overall circuit complexity is lower,

Factoring of RSA modulus N			ECDLP in $E(\mathbb{F}_p)$			HECDLP in $E(\mathbb{F}_p)$				
$\log_2(N)$ bits	Qubits	Toffoli gates	$\log_2(p)$ bits	Qubits	Toffoli gates	$\log_2(p)$ bits	Qubits (Affine)	Qubits (Kummer)	Toffoli gates (Affine)	Toffoli gates (Kummer)
512	1026	$6.41 \cdot 10^{10}$	110	1014	$\sim 4.04 \cdot 10^{9}$	55	627	500	$\sim 1.42 \cdot 10^{9}$	$\sim 9.23 \cdot 10^{8}$
1024	2050	$5.81 \cdot 10^{11}$	160	1466	$\sim 1.34 \cdot 10^{10}$	80	903	725	$\sim 4.76 \cdot 10^{9}$	$\sim 3.11 \cdot 10^{9}$
-	-	-	192	1754	$\sim 2.41 \cdot 10^{10}$	96	1080	869	$\sim 8.58 \cdot 10^{9}$	$\sim 5.59 \cdot 10^{9}$
2048	4098	$5.20 \cdot 10^{12}$	224	2042	$\sim 3.93 \cdot 10^{10}$	112	1256	1013	$\sim 1.41 \cdot 10^{10}$	$\sim 9.18 \cdot 10^{9}$
3072	6146	$1.86 \cdot 10^{13}$	256	2330	$\sim 6.01 \cdot 10^{10}$	128	1432	1157	$\sim 2.16 \cdot 10^{10}$	$\sim 1.41 \cdot 10^{10}$
7680	15362	$3.30 \cdot 10^{14}$	384	3484	$\sim 2.18 \cdot 10^{11}$	192	2138	1733	$\sim 7.90 \cdot 10^{10}$	$\sim 5.15 \cdot 10^{10}$
15360	30722	$2.87 \cdot 10^{15}$	521	4719	$\sim 5.72 \cdot 10^{11}$	260	2887	2345	$\sim 2.08 \cdot 10^{11}$	$\sim 1.35 \cdot 10^{11}$

Table 9: Comparison of estimations of circuit sizes between different cryptographic schemes

due to the smaller bit-size inputs. The number of qubits required in the circuits for hyperelliptic curves is approximately half of that for elliptic curves.

7.1 Possible Extensions

In order to calculate a more accurate estimate for the number of gates required for a divisor addition, we would have to simulate the circuit in a quantum circuit simulator such as LIQUi|> [29] for different bit-sizes of p, and use interpolation to calculate a curve of best fit of these points, as per the method described in [2]. Another possible extension could be to verify whether we can calculate the conversion from affine coordinates to Kummer surface coordinates in pre processing to avoid complexity in the quantum circuit. We can do this using the same quantum simulation software, as we can verify that our results match those gained using the conventional method using affine coordinates.

8 Appendix

I have included a .zip folder, which contains some sage worksheets, and some LIQUi|> code in F#. The sage worksheets were for the gate cost calculations, and an implementation of hyperelliptic curve arithmetic to help me understand it better. The LIQUi|> code is an attempt at creating a 5 bit full adder as a quantum circuit, in attempts to eventually creating a quantum circuit which simulated hyperelliptic curve addition. It is not complete due to time constraints. I also used LIQUi|> to simulate a run of shor's alrgorithm applied to the factorisation problem. I have also included a copy of the code available at https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2018/sxf546

8.1 Instructions to run code

Both the sage worksheets can be run in any sage compilation environment, I used the Cocalc online compiler, which can be found at https://cocalc.com/. To use this, you must first create an account, then import my worksheets to a project. The LIQUi|> code will require a download of the LIQUi|> software, which can be found at http://stationq.github.io/Liquid/. The factorisation code is included with the example code as a function called __shor(). The user manual for LIQUi|> can be found at http://stationq.github.io/Liquid/docs/LIQUiD.pdf.

References

[1] Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1969):339–354, 1998.

[2] Martin Roetteler, Michael Naehrig, Krysta M Svore, and Kristin Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 241–270. Springer, 2017.

[3] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *International workshop on cryptographic hardware and embedded systems*, pages 119–132. Springer, 2004.

[4] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[5] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[6] Abhilash Ponnath. Difficulties in the implementation of quantum computers. *arXiv preprint cs/0602096*, 2006.

[7] Yuanhao Wang, Ying Li, Zhang-qi Yin, and Bei Zeng. 16-qubit IBM universal quantum computer can be fully entangled. *arXiv preprint arXiv:1801.03782*, 2018.

[8] Julian Kelly. A preview of bristlecone, Google's new quantum processor. *Science*, 2018.

[9] Sergey Bravyi, David Gosset, and Robert König. Quantum advantage with shallow circuits. *Science*, 362(6412):308–311, 2018.

[10] Lenstra A.K., Lenstra H.W., Manasse M.S., and Pollard J.M. The number field sieve. *Lecture Notes in Mathematics, vol 1554*, 1993.

[11] Enrique Martín-López, Anthony Laing, Thomas Lawson, Roberto Alvarez, Xiao-Qi Zhou, and Jeremy L. O'Brien. Experimental realization of Shor's quantum factoring algorithm using qubit recycling. *Nature Photonics*, 6, Oct 2012.

[12] Nikesh S Dattani and Nathaniel Bryans. Quantum factorization of 56153 with only 4 qubits. *arXiv preprint arXiv:1411.6758*, 2014.

[13] John Proos and Christof Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *arXiv preprint quant-ph/0301141*, 2003.

[14] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006.

[15] D. E. KNUTH. Seminumerical algorithms. *The Art of Computer Programming*, 2, 1969.

[16] Michael T McClellan and Jack Minker. The art of computer programming, vol. 3: sorting and searching, 1974.

[17] Steven D. Galbraith and Pierrick Gaudry. Recent progress on the elliptic curve discrete logarithm problem. *Designs, Codes and Cryptography*, 78(1):51–72, Jan 2016.

[18] Pierrick Gaudry and David Lubicz. The arithmetic of characteristic 2 kummer surfaces and of elliptic kummer lines. *Finite Fields and Their Applications*, 15(2):246–260, 2009.

[19] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of elliptic and hyperelliptic curve cryptography*. CRC press, 2005.

[20] David G Cantor. Computing in the jacobian of a hyperelliptic curve. *Mathematics of computation*, 48(177):95–101, 1987.

[21] Jan Pelzl, Thomas Wollinger, Jorge Guajardo, and Christof Paar. Hyperelliptic curve cryptosystems: Closing the performance gap to elliptic curves. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 351–365. Springer, 2003.

[22] Joppe W. Bos, Craig Costello, Huseyin Hisil, and Kristin Lauter. Fast cryptography in genus 2. Cryptology ePrint Archive, Report 2012/670, 2012. https://eprint.iacr.org/2012/670.

[23] Nicolas Thériault. Index calculus attack for hyperelliptic curves of small genus. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 75–92. Springer, 2003.

[24] Jasper Scholten and Frederik Vercauteren. An introduction to elliptic and hyperelliptic curve cryptography and the ntru cryptosystem. *State of the Art in Applied Cryptography, COSIC*, 3, 2003.

[25] Jon Rowe Iain Styles. Introduction to molecular and quantum computation. http://www.cs.bham.ac.uk/internal/courses/intro-mqc/current/, 2008.

[26] Gary J Mooney, Charles D Hill, and Lloyd CL Hollenberg. Entanglement in a 20-qubit superconducting quantum computer. *arXiv preprint arXiv:1903.11747*, 2019.

[27] Steven A Tretter. *Introduction to discrete-time signal processing*. Wiley New York, 1976.

[28] Vivien M Kendon and William J Munro. Entanglement and its role in shor's algorithm. *arXiv preprint quant-ph/0412140*, 2004.

[29] Dave Wecker and Krysta M. Svore. LIQUi|>: A Software Design Architecture and Domain-Specific Language for Quantum Computing, 2014.

[30] Martin Ekerå. Modifying shor's algorithm to compute short discrete logarithms. *IACR Cryptology ePrint Archive*, 2016:1128, 2016.

[31] Sylvain Duquesne. Montgomery scalar multiplication for genus 2 curves. In *International Algorithmic Number Theory Symposium*, pages 153–168. Springer, 2004.