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

9 March 2019

1 Acknowledgments

Contents

List of Figures

List of Tables

Abstract

This project aims to provide 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.

2 Introduction

2.1 Background

Since Shor first introduced a polynomial time algorithm for prime factorisation and discrete logarithms [4], 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 [5] - 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 [6]. At present, the largest functional quantum computer, named Bristlecone, has 72 Qubits and is made by Google [7].

2.2 Appliances of Quantum Computing

Quantum computers have been shown to be faster than classical computers at solving certain problems [8], 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 [9], 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 [10], however the largest number factorised on a quantum computer is 56,153, which required 4 qubits [11]. 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) [12], 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 [12]. 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* [13] 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 and removing 0:

$$\mathbb{Z}_p = \mathbb{Z}_p \setminus \{0\}$$

These types of fields are also typically called Prime Galois Fields and can be denoted as $GF(p)$. We can define a cyclic subgroup from this field: $g \in \mathbb{Z}_p$ which means 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 $\alpha, \mod p$. 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 [14]. For example

Algorithm 1 Square and Multiply Algorithm

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

```
1: function F($\alpha, X$)
2:     if $X = 1$ then
3:         return $X \mod p$
4:     else if $X \mod 2 = 0$ then
5:         return $F(\alpha^2 \mod p, X/2)$
6:     else if $X \mod 2 = 1$ then
7:         return $\alpha^2 \times F(\alpha, X-1) \mod p$
```

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 [15]. 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 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, 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. 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

To begin understanding Hyperelliptic Curve Cryptography, it may be of help to briefly describe Elliptic Curve Cryptography (ECC). 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

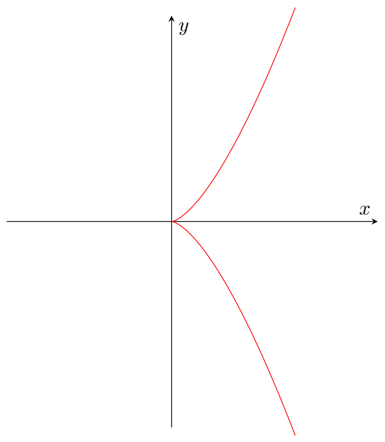

Figure 1: Example of cusp on graph $y^2 = x^3$

algebraically by making sure that the 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.

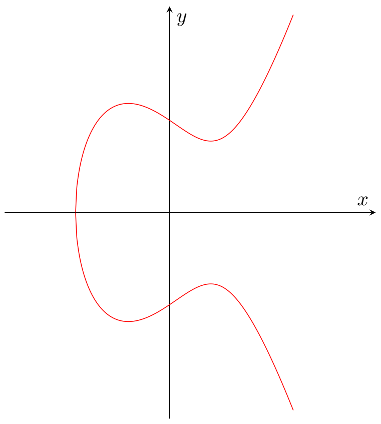

Figure 2: Elliptic Curve satisfying $y^2 = x^3 - 3x + 5$

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

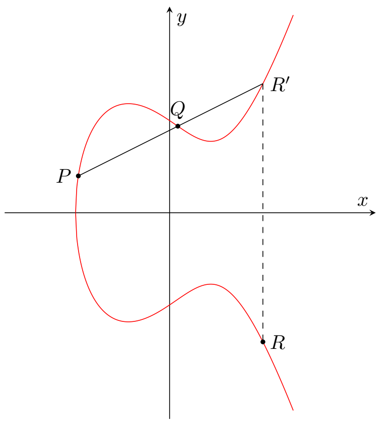

Figure 3: Dot operation on two points P and Q to give final point R

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.

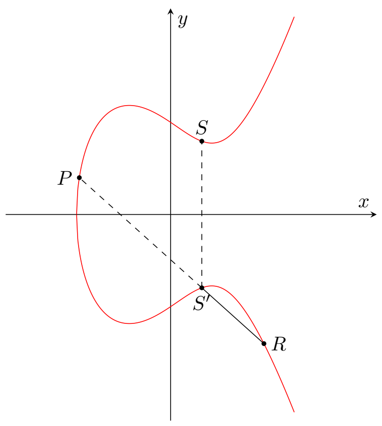

Figure 4: Dot operation on points R and P to give final point S

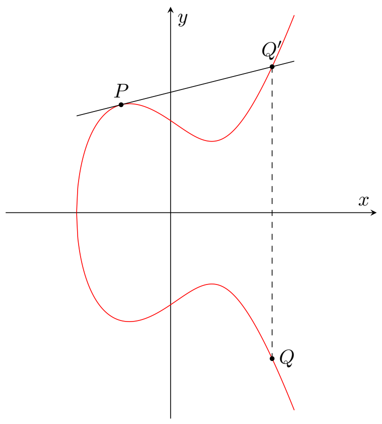

Figure 5: Point doubling operation on point P to give point $Q = 2 * P$

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

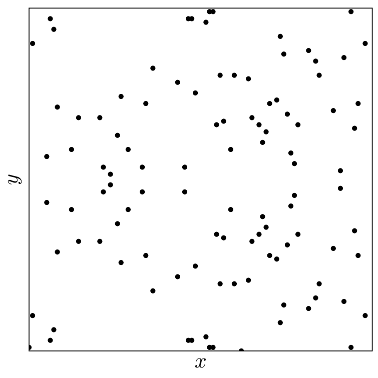

Figure 6: The elliptic curve $y^2 = x^3 - 3x + 5$ restricted to elements of the finite field GF(97)

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.

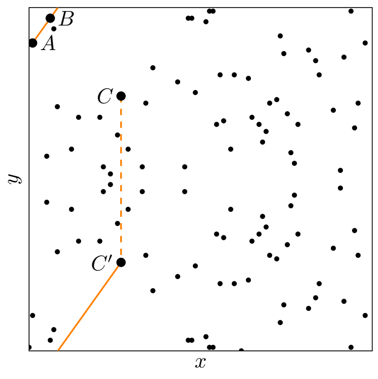

Figure 7: Dot operation on A and B when restricted to finite field GF(97)

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 [16]. 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 [2]. 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 [2]

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 [17]. 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.

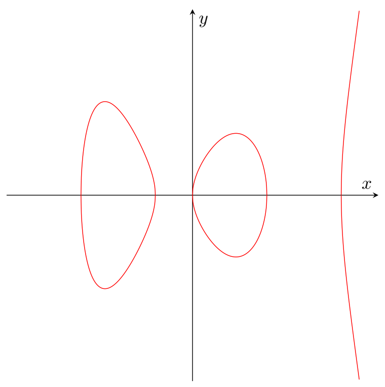

Figure 8: Hyperelliptic Curve satisfying $y^2 = x^5 - 5x^3 + 4x + 1$

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* [18]. 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.* [19]. 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 [18]. 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 $\bar{D} = \bar{D_1} \oplus \bar{D_2}$

```
1:  $d_1 \longleftarrow \gcd(u_1, u_2)$
2:  $d \longleftarrow \gcd(d_1, v_1 + v_2 + h)$
3:  $s_1 \longleftarrow c_1e_1, s_2 \longleftarrow c_2e_2$ and $s_3 \longleftarrow c_2$
4:  $u \longleftarrow \frac{u_1u_2}{d^2}$ and $v \longleftarrow \frac{s_1u_1v_2+s_2u_2v_1+s_3(v_1v_2+f)}{d}$ $mod\ u$
5:  repeat
6:      $u' \longleftarrow \frac{f-vh-v^2}{u}$ and $v' \longleftarrow (-h-v)$ $mod\ u'$
7:      $u \deg u'$ and $v \longleftarrow v'$
8:  until $deg(u) \leq g$
9:  make $u$ monic by dividing through by value of first coefficient
10: 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.

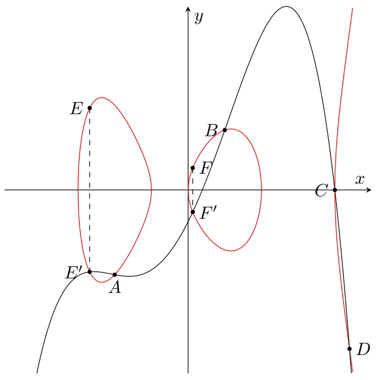

Figure 9: Visual representation of adding two reduced divisors: $(A + B) \oplus (C + D) = (E + F)$

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 [20] 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 [21] 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 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 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 specifically 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 [22]. For this reason, the genus of a curve is suggested to be less than 3 to ensure security against this attack [23].

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.

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, consisting of some 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 $m \times m$ 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 Entanglement

If we have two independent qubits in the states $|\psi_0\rangle = \frac{1}{\sqrt{3}}|0\rangle + \sqrt{\frac{2}{3}}|1\rangle$ and $|\psi_1\rangle = \frac{1}{\sqrt{5}}|0\rangle + \frac{2}{\sqrt{5}}|1\rangle$, then we can create a composite system from these as such:

$$|\psi\rangle = \frac{1}{\sqrt{3}}\frac{1}{\sqrt{5}}|00\rangle + \frac{1}{\sqrt{3}}\frac{2}{\sqrt{5}}|01\rangle + \sqrt{\frac{2}{3}}\frac{1}{\sqrt{5}}|10\rangle + \sqrt{\frac{2}{3}}\frac{2}{\sqrt{5}}|11\rangle$$

$$|\psi\rangle = \frac{1}{\sqrt{15}}|00\rangle + \frac{2}{\sqrt{15}}|01\rangle + \sqrt{\frac{2}{15}}|10\rangle + \frac{2\sqrt{2}}{\sqrt{15}}|11\rangle$$

These two qubits are then said to be entangled. Entangled qubits have a multitude of useful properties when designing quantum circuits, which will be discussed more in depth in a later section.

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

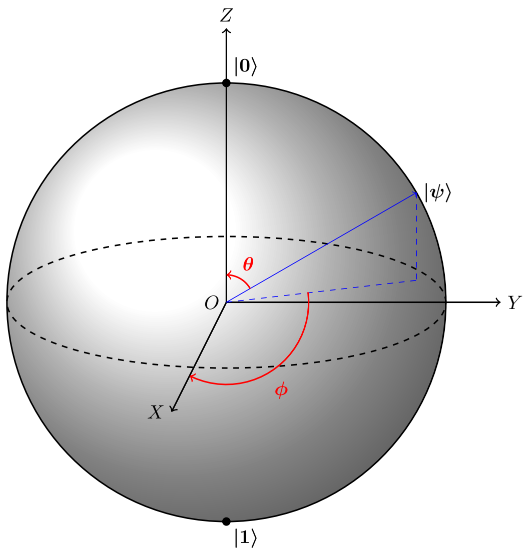

Figure 10: Vector Representation of Qubit Position

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.

4.2.3 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. More commonly in quantum circuits, we use a controlled version

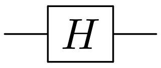

Figure 11: Quantum circuit describing Hadamard gate

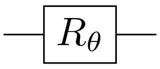

Figure 12: Quantum circuit describing phase shift gate by arbitrary angle θ

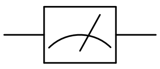

Figure 13: Quantum circuit describing measurement gate

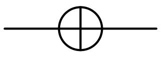

Figure 14: Quantum circuit describing NOT gate

of gates, which means that the gate takes two inputs, one of which controls the operation of the gate, for example the controlled not (CNOT) 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 concept of control gates can be extended to any unitary transformation. The controlled not gate is an example of

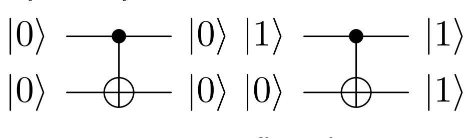

Figure 15: Quantum circuits describing the effect of controlled not gate on different inputs

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

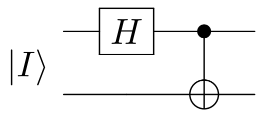

Figure 16: Quantum circuit describing how to create entangled states

what is happening in this circuit, we can split the circuit up, as described in Figure 17 When the

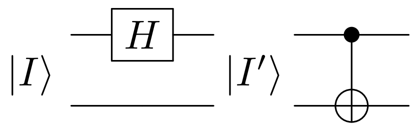

Figure 17: Quantum circuit for entanglement split into components

circuit is split up in this way, we can describe what transformations are 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

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

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

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

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:

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

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 $ac, bd = \frac{1}{\sqrt{2}}$. This is impossible in conventional practice, but since the quantum state does not obey the laws of classical mechanics, it is possible.

4.3 Quantum Fourier Transform

Another key example of a quantum logic circuit which is used in many quantum algorithms is the Quantum Fourier transform. This is a quantum version of the classical discrete Fourier transform, 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.

4.3.1 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 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 samples of this function's amplitude at

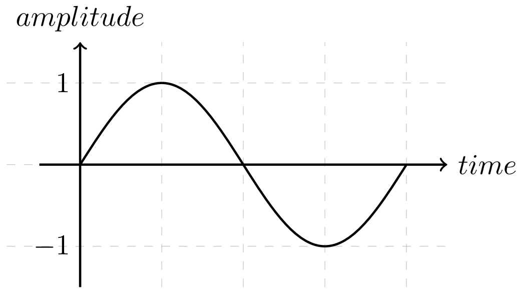

Figure 18: Sine wave with frequency 1Hz and amplitude 1

equally spaced points. For this example we use a sampling frequency of 8 Hz: The eight values

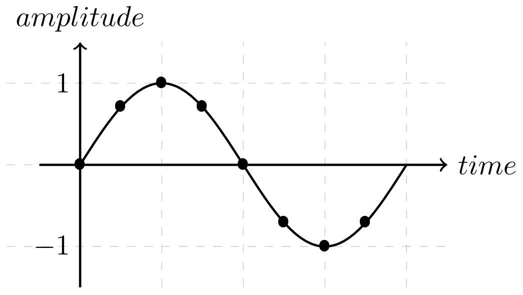

Figure 19: Sine wave sampled at 8Hz

for amplitude we get are displayed in Table 2 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 X_k as shown in Table 3 From this we can see that the only non-zero values are those for X_1 and 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

	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 2: Amplitudes of samples

	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 3: Fourier Coefficients of samples

as a frequency against magnitude graph as shown in Figure 20 The Nyquist limit is the highest

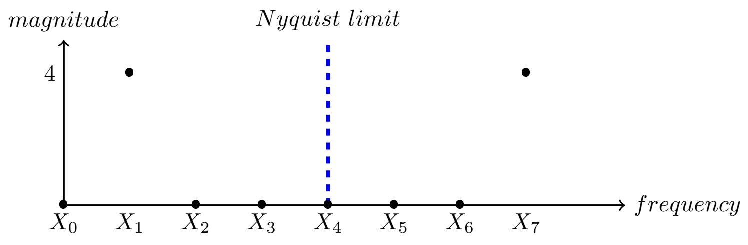

Figure 20: Frequency against magnitude plot for sample

frequency that can be coded at a given sampling rate to be able to fully reconstruct the original signal, and it is equal to $\frac{N}{2}$ [24]. 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 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 21 The angle θ is equal to $\frac{3\pi}{2}$, which is our phase shift from a normal

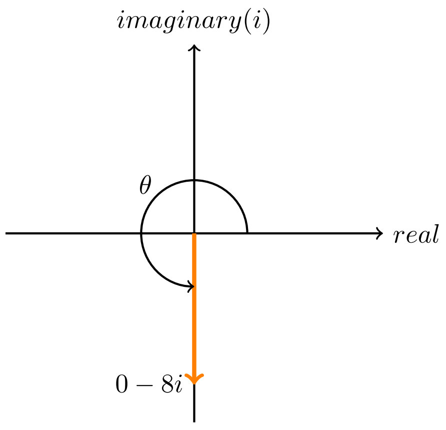

Figure 21: Angle between positive real axis and sample

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.

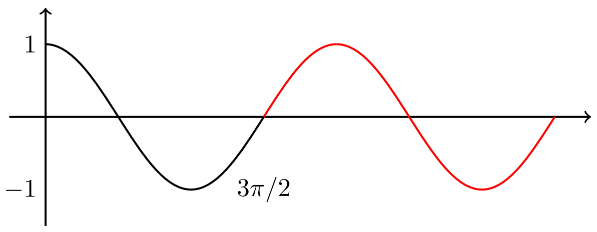

Figure 22: Phase shift of sample from cosine wave

4.3.2 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:

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

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:

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

4.3.3 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 23. The gate at the end is called a *swap* gate, and simply swaps the order of the qubits. This

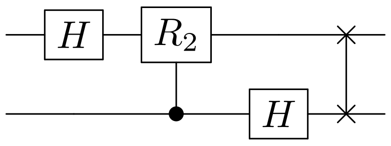

Figure 23: Quantum Fourier Transform for a 2-qubit circuit

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 24, 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

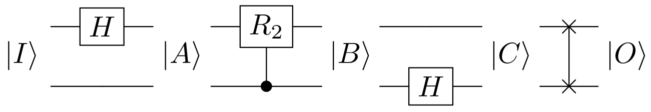

Figure 24: Quantum Fourier Transform for a 2-qubit circuit split into components

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 25, and we can denote it in a larger circuit as a single gate to make things simpler using the notation in Figure 26.

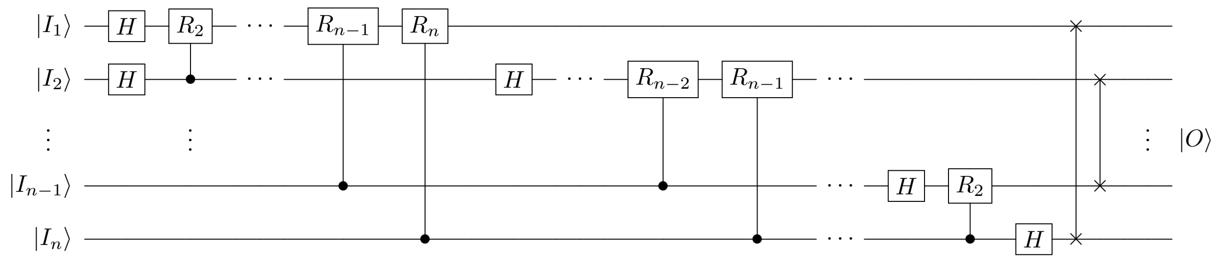

Figure 25: Quantum Fourier Transform for a n-qubit circuit

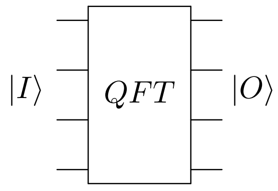

Figure 26: Quantum Fourier Transform gate

4.4 Shor's Algorithms

In his paper: *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer* [4], 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.

4.4.1 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 $f(x) = a^x \mod N$.

4.4.2 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 $U|\psi\rangle \equiv e^{2\pi i\phi}|\psi\rangle$, $0 \le \phi < 1$ [3]. 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 27. This circuit involves two registers of qubits, R_1 and R_2. Register 1 contains n qubits which

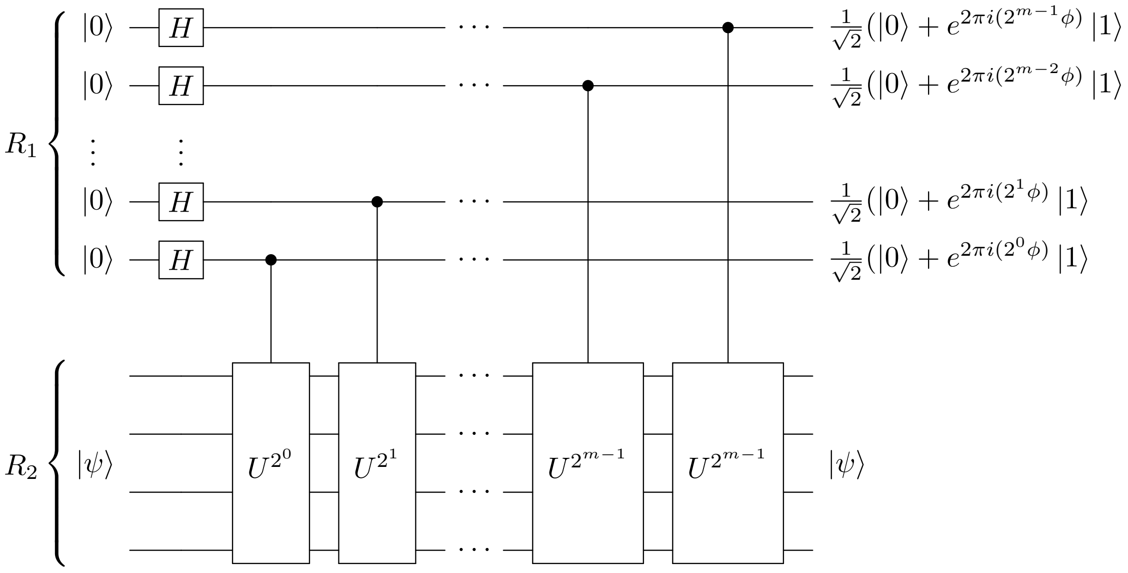
R_1
$|0\rangle$
H
$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i(2^{m-1}\phi)}|1\rangle$
$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i(2^{m-2}\phi)}|1\rangle$
$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i(2^{1}\phi)}|1\rangle$
$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i(2^{0}\phi)}|1\rangle$
R_2
$|\psi\rangle$
U^{2^0}
U^{2^1}
$U^{2^{m-1}}$
$U^{2^{m-1}}$
$|\psi\rangle$

Figure 27: Phase Estimation Circuit [1]

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:

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

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

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

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

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 [25].

4.4.3 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 $e^{2\pi is/r}$ [3], 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 28. The next section will describe a practical example of how we can obtain the factors of a number using Shor's algorithm.

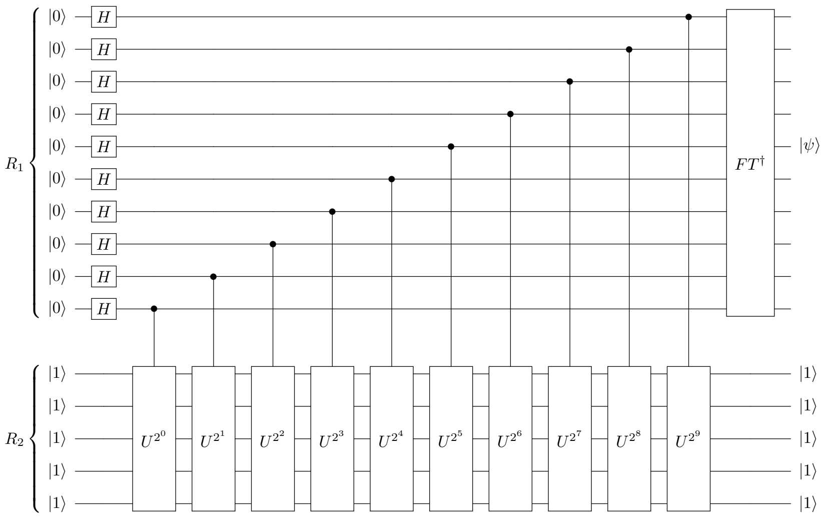

Figure 28: Quantum Circuit to Factorise 21

4.4.4 Example of Factorisation Using Shor's Algorithm

Using the software LIQUi|> [26], 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 [3], 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 4. Equation 5.3 in Shor's paper describes how we

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

Table 4: Values for d and r obtained from continued fraction algorithm [Box 5.3] [3]

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:

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

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.

4.4.5 Shor's Algorithm for Discrete Logarithms

Shor's paper also describes a quantum routine for solving the discrete logarithm problem for a prime group $\mathbb{F}p$

Describe Shor's algorithm for discrete log briefly, then describe how it is adapated to any cyclic group using source 1 and 2

4.4.6 Quantum Circuit for Discrete Logarithms

4.5 Shor's Algorithm Adapted for the Elliptic and Hyperelliptic Curve Discrete Logarithm Problem

5 Methods

6 Results

7 Discussion

8 Conclusions

9 Recommendations

10 References

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] 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.

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

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

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

[6] 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.

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

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

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

[10] 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.

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

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

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

11 Appendices