Concrete quantum-cryptanalysis of binary elliptic curves

van Hoof, Iggy

Award date:
2019

Concrete quantum-cryptanalysis of binary elliptic curves

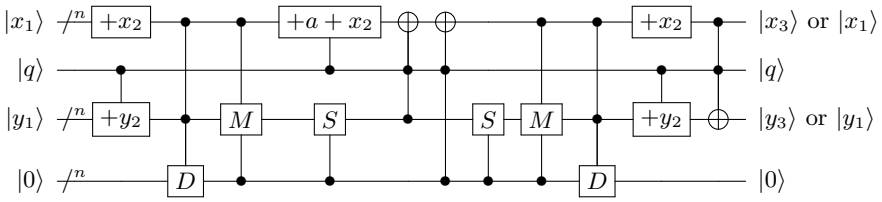
Iggy van Hoof

Technische Universiteit Eindhoven

Department of Mathematics and Computer Science

Supervisors:
Prof. dr. Tanja Lange
Dr. Gustavo Banegas

Committee:
Prof. dr. Tanja Lange
Dr. Gustavo Banegas
Dr. Jesper Nederlof



I Abstract

In this thesis we look at quantum algorithms for computing binary elliptic curve discrete logarithms. Elliptic curve cryptography relies on the hardness of the elliptic curve discrete logarithm, but quantum computers can break it in polynomial time. We give quantum algorithms for binary finite field operations, with the purpose of giving concrete estimates of the hardness of the binary elliptic curve discrete logarithm problem on quantum computers. These algorithms include translating a recent optimized classical inversion algorithm to a quantum setting, resulting in a low number of qubits required for division. They also include a new Karatsuba-based multiplication algorithm, which is optimal (for any Karatsuba-based multiplication algorithm) in space and number of Toffoli gates.

The number of logical qubits required for solving the binary elliptic curve discrete logarithm is reduced to $7n + \lfloor \log(n) \rfloor + 9$. This is at least $2n$ qubits lower than comparable previous work. The number of Toffoli gates required is $48n^3 + 8n^{\log(3)+1} + 352n^2 \log(n) + 512n^2 + O(n^{\log(3)})$ and the number of CNOT gates $O(n^3)$ with exact numbers given in this thesis for elliptic curves currently used for cryptography. While the Toffoli and CNOT gate count is high, the number of qubits required is minimal for the given division algorithm and coordinate system.

II Acknowledgements

This thesis is the accumulation of a lot of the work I have done so far on this university. I have never been alone so I would like to thank the people who have made this possible. I would like to thank Tanja Lange not just for her work as a teacher, but also for being a mentor during the last year of my bachelor degree and all of my master's degree. Tanja has helped me meet many people in this field and without her lectures, advice and encouragement I would have never been as interested in cryptography and computer security as I am now.

I have only properly worked with Gustavo Banegas during my master thesis (which, to be fair, is quite a while) but he has been nothing but kind, helpful, smart and understanding. He also got his PhD a month before my final presentation, congratulations. I would also like to thank Jesper Nederlof for being a member of my graduation committee, being open to communication even on vacation. I also want to thank Daniel J. Bernstein for advice on classical algorithms and quantum computing and generally having a great understanding of every subject relevant to my thesis.

I would also like to thank my friends at this university. I have met so many great people, not just fellow students I took courses with, worked with and partied with during my studies but also the amazing group of Coding and Crypto, and Security people, as well as the many great people I have met through the study association GEWIS. I will never regret going to Eindhoven University of Technology.

Finally, I would like to thank my family for their support, love and understanding with not just my studies but my life in general. I have had my fair share of personal difficulty, but always having a family to rely on has made this possible.

And of course I would like to thank everyone I forgot to thank in my acknowledgements.

Table of Contents

1 Introduction

Current cryptographic systems used on the internet rely on Diffie-Hellman key exchange, a way to exchange secret keys over a public channel. One of the most common Diffie-Hellman variants uses operations on elliptic curves (ECC). The key-exchange schemes rely on problems that are hard to solve with a classical computer. However, a quantum computer has advantages against these problems and can solve them exponentially faster.

Real quantum computers are becoming increasingly powerful. While still being relatively small compared to classical computers, a time will soon (in the next few decades) come where quantum computers can threaten computer security. This thesis looks at a specific instance of a currently used cryptographic system and makes estimates how large a quantum computer would have to be to quickly break it.

1.1 Previous work

When looking at previous work, we compare the resources for elliptic curve operations against both a paper by Amento, Rötteler and Steinwandt [1] which looks at the same cryptographic system (binary ECC) and a paper by Rötteler, Naehrig, Svore and Lauter [20] which looks at a different case (prime field ECC). Both make optimizations in a single area, in both cases division, as well as providing an overview of techniques. These papers differ in their cost metrics: the paper on binary ECC uses depth (runtime) as its singular metric, making sacrifices in the space that this thesis is not making. The second paper, on prime field ECC, uses space and gate count as its primary metrics, which is the same as this thesis, further detailed in Section 3.5. As such, we are using some strategies which that paper also uses, modifying and improving where possible for the specific case of binary ECC.

Binary ECC uses, among other steps, a multiplication of binary polynomials. For this specific part, we base our space-efficient variant on classic space-efficient algorithms. Roche [19] describes a classic time-efficient multiplier for polynomials in $O(\log n)$ space, later expanded by Cheng [5] to also work for integers. With this as a basis we can implement a quantum Karatsuba-based multiplier with as little space as possible.

Another important step is division of binary polynomials. The division algorithm in [20] uses a method based on greatest common divisor algorithms, which is common for division in finite fields. A recent (2019) paper by Bernstein and Yang [4] shows how to do finite field division in constant classical time, which makes it very suitable for translation to quantum computing. The division in [1] uses exponentiation and multiplication to get the inverse finite field element, using a strategy from Itoh and Tsuji from 1988 [11] which we will also show how to implement on quantum computers.

1.2 Our contributions

This thesis makes 3 primary contributions:

- A complete overview of quantum point addition on binary elliptic curves with a concrete count of space and time: $7n + \lfloor \log(n) \rfloor + 9$ qubits and $48n^3 + 8n^{\log(3)+1} + 352n^2 \log(n) + 512n^2 + O(n^{\log(3)})$ Toffoli gates.
- An improved Karatsuba multiplier. Previous instances of binary quantum Karatsuba multiplication traded space for time or time for space. The algorithm in this thesis has minimal space and the same number of quantum bit multiplications (Toffoli gates) as bit multiplications in classical Karatsuba. The algorithm can use some refining in the number of CNOT gates, but is optimal for our purposes. I have made this contribution available as a standalone paper, which is currently available on arXiv preprint [25].
- A concrete comparison of Fermat's little theorem-based division algorithms versus extended greatest common divisor-based algorithms (also known as Euclid's algorithm). This includes a quantum implementation of Bernstein and Yang's recent gcd-based variant that runs in constant, low classical time.

1.3 Overview

Sections 2 and 3 consist of background on elliptic curves and quantum computing respectively, while clarifying notation and goals. Section 4 details Shor's algorithm, the general quantum algorithm we use to solve discrete logarithm problems. Section 5 introduces basic operations like addition and constant multiplication. Section 6 details a binary multiplication algorithm which is used in Section 7 to achieve binary finite field multiplication with no extra space and minimal time. Section 8 details and compares two methods to do division: a new algorithm using extended greatest common divisor and an algorithm using Fermat's little theorem. In Section 9 we put this together to achieve point addition on binary elliptic curves. The resulting resource count and a comparison to other work is given in Section 10. Finally, Section 11 draws a conclusion and details future work.

2 Binary elliptic curve discrete logarithm

This section contains a very brief introduction into binary elliptic curve cryptography, the primary application of this thesis.

2.1 Elliptic curves over the reals

Standard elliptic curves are non-singular curves over $\mathbb{R}^2$ with an equation of the form $y^2 = x^3 + ax + b$. A curve is non-singular if $a, b \in \mathbb{R}$ with $4a^3 + 27b^2 = 0$ [2]. Figure 1 is an example of an elliptic curve. For our purposes singular points on a curve are points where the curve intersects itself, or sharp corners. We can define a group operation on this curve: when we have two different points on the curve, P_1 and P_2, we can add these points together. If we draw a line through P_1 and P_2 we get a third point R on the curve. We say $P_1 + P_2 = -R$, the point

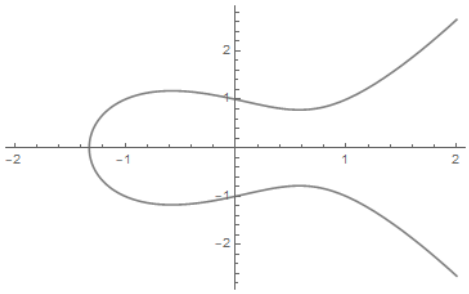


Figure 1: $y^2 = x^3 - x + 1$

R reflected through the x-axis (every elliptic curve over $\mathbb{R}^2$ is symmetrical with respect to the x-axis).

If P_1 and P_2 share an x-coordinate but have the opposite y-coordinate, adding them together does not result in a point on the curve. In that case, we say that the addition result is a new point, O, the point at infinity. It is above and below every point, $P + O = P$ for any P on the curve. We can also double points, notation $[2]P$. Doubling uses the tangent line at the point rather than the line through two points, with $O + O = O$. Curves over the reals have additional properties we do not explore here.

2.2 Elliptic curves over a finite field

The points on a curve in a finite field (for example, the integers modulo a prime) taken together with O form a group under point addition. In this group, each point P has an order $\text{ord}(P)$, which is the smallest positive integer such that $[\text{ord}(P)]P = O$. The number of points and thus the order is bounded by Hasse's bound [10], which states that the number of points on an elliptic curve is bounded by the field size with a margin up to twice the square root of the field size. We look at curves specifically over finite fields with characteristic 2.

2.3 Binary elliptic curves

Binary elliptic curves are curves over binary finite fields. We refer to the field of 2^n elements as $\mathbb{F}_{2^n}$ or $GF(2^n)$. A binary polynomial is a polynomial with only 1 and 0 as its coefficients, for example $x^{10} + x^3 + 1$. The sum of two polynomials $f(x), g(x)$ takes the coefficient of each term of $f(x)$ and adds it modulo 2 to the coefficient of the same term of $g(x)$, for example $(x^{10} + x^3 + 1) + (x^{10} + x^2) = x^3 + x^2 + 1$. Multiplication works as expected, with the final coefficients being taken modulo 2, for example $(x + 1) \cdot (x + 1) = x^2 + 1$. In a binary field, we take these polynomials modulo an irreducible polynomial $m(x)$, where n is the degree of $m(x)$. For example, $x^3 + x \mod x^2 + x + 1 = x + 1$. The finite field of binary polynomials modulo $m(x)$ has 2^n different elements (note all polynomials of degree less than n can be represented as n-length bitstrings), which means

that by Hasse's bound, the order of a point is at most $2^n + 2^{\frac{n}{2}} + 1$ (this is smaller than 2^{n+1} for $n > 2$) on a binary elliptic curve over $\mathbb{F}_{2^n}$.

Binary elliptic curve formulas look a little different from the real case:

$$y^2 + xy = x^3 + ax^2 + b$$

with $a, b \in \mathbb{F}_{2^n}$ and $b \neq 0$. As such, we have the following point addition formula for adding $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ with $x_1 \neq x_2$ to get $P_1 + P_2 = P_3 = (x_3, y_3)$:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$$

$$y_3 = (x_2 + x_3)\lambda + x_3 + y_2$$

with

$$\lambda = \frac{y_1 + y_2}{x_1 + x_2}.$$

There are additional formulas for point doubling:

$$x_3 = \lambda^2 + \lambda + a$$

$$y_3 = x_1^2 + (\lambda + 1)x_3$$

with

$$\lambda = x_1 + y_1/x_1.$$

When $x_1 = x_2$ but $y_1 \neq y_2$ (in which case $y_1 + y_2 = x_1$) they are inverses, so $P_3 = O$

2.4 Elliptic curve Diffie-Hellman

Elliptic curve Diffie-Hellman, the primary key exchange mechanism using elliptic curves, works as follows: Alice and Bob want to privately agree on a secret point on a public curve while communicating in a public space. To do this, each takes a secret integer α and β respectively. Publicly, they agree on a point P with a large prime order. Then, they calculate and tell each other $P_\alpha = [\alpha]P$ and $P_\beta = [\beta]P$. Finally, they calculate their shared point $P_{\alpha\beta} = [\alpha \cdot \beta]P = [\alpha]P_\beta = [\beta]P_\alpha$. Despite everyone knowing P, P_α and P_β, it is very hard to find α, β or $P_{\alpha\beta}$ with just a classical computer. This problem is called the elliptic curve discrete logarithm problem. With a quantum computer, algorithms exist which solve this in polynomial time. We will look at such algorithms.

3 Quantum background

This section contains a brief overview of quantum computing. This section was inspired by a Mastermath course on quantum computing, taught by Ronald de Wolf at the University of Amsterdam (UvA). Lecture notes of the most recent iteration of this course are available online [6].

3.1 Qubits

A classical bit can take 2 positions: 0 or 1, measuring that bit does nothing to it and using transistors we can have gates like AND or OR. In the quantum case we have quantum bits (**qubits**), for which these things are not true. The reader might be familiar with the double-slit experiment where light, which is made from tiny 'particles' called photons, behaves both as a particle and a wave (and the experiment is less cruel than Schrödinger's cat). By pointing a light source at a plate with 2 slits and a screen behind it, the pattern on the screen shows a pattern of interference that cannot be explained if photons were particles. Furthermore, by placing a light detector in the slit that should not alter the shape of the pattern, the pattern changes due to the photons being observed.

This displays that two of the introduced properties of bits do not hold in the quantum space: objects can be in two states at once, referred to as a **superposition** and **measuring** that object collapses the object to one of those states. In addition, like the **interference** of the waves of light, qubits interfering can cause the result to change in ways that would not be possible if qubits were not in superposition.

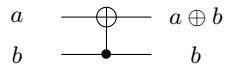
The state of n qubits is represented as a vector of length 2^n or a weighted sum of base states written using the ket notation. For example $(1,0)^T$ would be $|0\rangle$ and $(0,1)^T$ would be $|1\rangle$. Now if we were to apply a Hadamard gate (detailed later) to the $|1\rangle$ state we get $(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$ or $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. When measuring we can only measure one state. The chance to observe $|0\rangle$ is the absolute value of the square of the corresponding vector element or coefficient of the state, which in this case is equal to the chance of $|1\rangle$ occurring.

Note that qubits are not limited to positive or negative, and can also have imaginary states. In that case, vector elements are often written as their distance to 0, called the **amplitude**, times a rotation on the unit circle, called the **phase**. For example, $-\frac{1}{\sqrt{2}} = \frac{1}{\sqrt{2}}e^{i\pi}$. While measuring qubits, we only care about the amplitude. However, the phase is significant for interference. We apply that interference through gates.

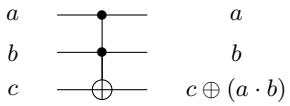
3.2 Reversible actions

Quantum computing uses reversible gates, because resetting qubits would require reading them and collapse (part of) the superposition and discarding them would quickly balloon the number of qubits required. Unlike classical gates like AND or XOR reversible gates are bijective (every input state corresponds to exactly one output state) and require an equal number of input and output qubits. In Sections 5 to 8 we state our algorithms only in terms of these gates applied to classic states, but the gates we use can be applied to superpositions of qubits in states $|1\rangle$ and $|0\rangle$. Each state then behaves as expected individually: applying a NOT-gate to $|0\rangle$ turns it into $|1\rangle$ no matter the phase or amplitude. For the purpose of binary elliptic curve point operations we need three gates to do reversible addition, multiplication and division:

- The NOT gate. It has one input and one output: if the input is $|0\rangle$, the output is $|1\rangle$ and vice versa. It is its own inverse.
- The CNOT, or Feynman, gate serves as the equivalent of XOR or $\mathbb{F}_2$-addition. This gate takes 2 qubits as inputs and adds one input to the other qubit and outputs the other qubit as itself: $(a, b) \to (a \oplus b, b)$. It is reversible and its own inverse: applying it twice results in $(a \oplus b \oplus b, b) = (a, b)$. In Circuit 1 an example has been drawn. In algorithms we write this as $a \leftarrow \text{CNOT}(a, b)$.
- The Toffoli (TOF) gate serves as the equivalent of AND or $\mathbb{F}_2$-multiplication in our case. This gate takes 3 qubits as inputs and adds the result of mulitplication of the first two qubits to the third qubit and outputs the other qubits as themselves: $(a, b, c) \to (a, b, c \oplus (a \cdot b))$. It is also its own inverse. In circuit 2 an example has been drawn. In algorithms we write this as $c \leftarrow \text{TOF}(a, b, c)$.

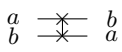


Circuit 1: The CNOT gate.



Circuit 2: The TOF gate.

In addition to these gates, we will also need to swap some qubits. Unlike the CNOT and TOF gates we do not build these in physical circuits. Rather, we change the index on some qubits: if we were to swap qubits a and b we would simply refer to qubit a as "b" and qubit b as "a" from that point on without counting any gates. In Circuit 3 an example has been drawn.

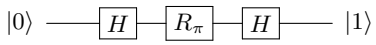


Circuit 3: The swap

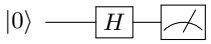
3.3 Quantum actions

We use 3 different actions that are purely quantum and not available in classic reversible computing:

- The Hadamard gate (H). This gate transforms $|0\rangle$ into $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ into $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Its primary use for us is putting zero states into uniformly random states, as well as differentiating between states with a positive and a negative phase. It is its own inverse.
- The phase shift gate (R_ϕ). This gate maintains $|0\rangle$ but transforms $|1\rangle$ into $e^{i\phi}|1\rangle$. We generally implement one specific $\phi = \pi/4$ (the T-gate) and use that (and the H and CNOT gates) to approximate any specific ϕ with arbitrary precision. The inverse of R_ϕ is $R_{-\phi}$. Circuit 4 shows a phase shift.
- Measurement. This collapses the quantum state of a qubit into a classical state. The chance to read any specific state is the absolute value of the square of its amplitude. Circuit 5 shows a measurement.

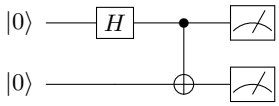


Circuit 4: A circuit that shows a phase shift.



Circuit 5: A circuit that applies a Hadamard gate and then measures. It has an equal chance of reading a $|1\rangle$ or a $|0\rangle$.

3.4 Entanglement



Circuit 6: A circuit that shows entanglement.

Quantum mechanics has a unique property called entanglement that is not present in the classical world. When 2 qubits interact, they become entangled. For example, in circuit 6 the chance to read $|1\rangle$ or $|0\rangle$ in the first qubit is still equal. However, when reading the second qubit it will always read the same as the first qubit. Even if you read them in opposite order, they will still be the same. The state before measuring is $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Using this entanglement we can make quantum algorithms.

3.5 Quantum Algorithms

Quantum algorithms consist of operations to registers of qubits. We divide those qubits into two types:

- Input and output qubits. These qubits contain the input and will contain the output after running the algorithm, potentially with some qubits being in the same state as before. For example, a Toffoli gate has 3 input and output qubits, but only 1 of them changes.
- Ancillary qubits. These qubits are used by the algorithm, but do not contain the input and output. For this thesis we restrict ancillary qubits to always start and end in a fixed state of $|0\rangle$.

Efficiency
There are several methods to measure the efficiency of algorithms:

- On the most basic level, we can compare the number of gates. However, quantum Toffoli gates are considered much more expensive than CNOT gates, with the exact difference depending on the physical realization of the quantum computer. As such, minimizing the number of Toffoli gates alone can be considered a better goal. The number of Toffoli gates will be an important concern in this thesis.
- Furthermore, the number of qubits an algorithm uses is something very relevant to implementations today. Actual quantum computers are slowly increasing their number of qubits. As such the space, or width, of an algorithm is also relevant. The lower this space, the sooner the algorithm can be implemented. Space will be the primary concern in this work.
- In addition to this, we can parallelize quantum circuits well: applying a circuit once on a set of qubits and once on a different set of qubits can be done twice as fast as applying that circuit twice on some of the same qubits. For example, $\text{CNOT}(a, b)$ and $\text{CNOT}(b, c)$ has to be done sequentially in 2 steps, while $\text{CNOT}(a, b)$ and $\text{CNOT}(c, d)$ can be done in one step. This measure of how many gates we need sequentially is called depth. In this work, depth will not be explored in-depth, but will be reported and left to future work.
- Finally, all of the above assumes quantum computers will not have errors. Precise quantum states are difficult to maintain and errors come quickly. Error correction has to be implemented to create what are called logical qubits, qubits on which operations can be performed with a reasonable degree of certainty. Error correction is not considered in this work and any mention of qubits refers to logical qubits.

An ideal analysis would take all of the above into account. In the interest of time and space only the first two are detailed, as those are what the author is most familiar with and has been a measure in previous work [20].

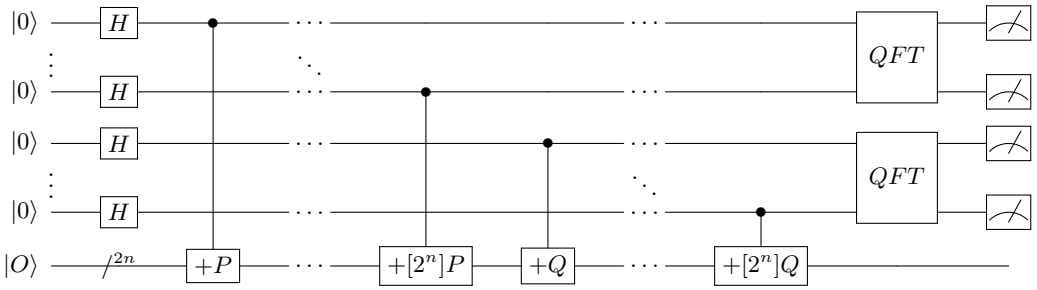
4 Shor's algorithm

In 1994, Peter Shor described how to use quantum computers to break traditional asymmetric cryptography [22]. While his primary example detailed how to break RSA by factoring integers in polynomial time on a quantum computer, he also showed how to extend his algorithm to any discrete logarithm problem. This includes ECC.

Shor's algorithm for solving discrete logarithms works as follows: we have two points $P, Q \in E(\mathbb{F}_{2^n})$ with $Q = [\mu]P$. We want to find μ. Take 2 registers of size $n+1$ in the $|0\rangle$ state and apply Hadamard gates to them to get them in a uniform superposition $\frac{1}{2^{n+1}} \sum_{k,\ell=0}^{2^{n+1}-1} |k, \ell\rangle$. Take another $2n$ qubits in a state representing $|O\rangle$. Conditional on the first 2 registers, we add classically precomputed points $[2^0]P, [2^1]P, ..., [2^n]P$ and $[2^0]Q, [2^1]Q, ..., [2^n]Q$ to the last $2n$ qubits and get

$$\frac{1}{2^{n+1}} \sum_{k,\ell=0}^{2^{n+1}-1} |k, \ell, [k]P + [\ell]Q\rangle.$$

A quantum Fourier transform (QFT), consisting of specific phase shift gates and Hadamard gates is applied to the first 2 registers[1]. Those two registers are then measured, and the measurement result can be used to compute μ classically [23]. When measuring the last $2n$ qubits, you would measure a point R, for which k, ℓ exist such that $[k]P + [\ell]Q = R$. Shor's algorithm finds the hidden period ν such that $[k + 1]P + [\ell + \nu]Q = R$, which you can use to find μ. In Circuit 7 the general algorithm is drawn. Note that it does not matter when the final $2n$ qubits are measured, so these can be measured when measuring the entire state or even after the result of the quantum Fourier transform is measured.

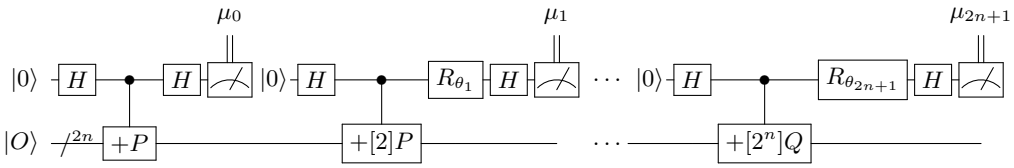


Circuit 7: Shor's algorithm for finding elliptic curve discrete logarithm.

By taking measurements after every step, we can compress the quantum Fourier transform on the first $2n+2$ qubits into a single qubit [9]. The phase shift

[1] Quantum Fourier transforms are not detailed in this thesis as the primary focus is on the elliptic curve operations.

after every step depends on the previous measurement outcomes $\mu_0, ..., \mu_{2n+1}$ with $\theta_k = -\pi \sum_{j=0}^{k-1} 2^{k-j} \mu_j$. In Circuit 8 the algorithm has been drawn.



Circuit 8: Shor's algorithm for finding elliptic curve logarithms with a semiclassical Fourier transform.

5 Basic Arithmetic

In this section we discuss reversible in-place algorithms for the basic arithmetic of binary polynomials modulo a field polynomial $m(x)$, i.e. elements of $\mathbb{F}_{2^n}$.

5.1 Addition and binary shift

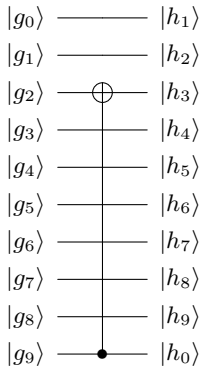
The first operation we consider, addition, can easily be implemented for binary polynomials. Individual additions can be done with a CNOT gate, the addition of two polynomials of degree at most $n-1$ takes n CNOT gates with depth 1. This operation uses no ancillary qubits and the result of the addition replaces either of the inputs. Since addition is component-wise, addition for polynomials over $\mathbb{F}_2$ is the same as addition for elements of the field $\mathbb{F}_{2^n}$.

Binary shifts are straightforward: they correspond to multiplying or dividing by x. This requires no quantum computation by doing a series of swaps.

Finally, if we have a fixed n, a polynomial $g(x)$ of degree at most $n-1$ and want to do a multiplication by x followed by a modular reduction by a fixed weight-ω (for our purposes ω will always be 3 or 5) and degree-n polynomial $m(x)$ that has coefficient 1 for x^0, we can do this in 2 steps. To describe this, we represent $m(x)$ as M where M is an ordered list of length ω that contains the degrees of the nonzero terms in descending order, for example if $m(x) = 1 + x^3 + x^{10}$ we get $M = [10, 3, 0]$. Let $g(x) = \sum_{i=0}^{n-1} g_i x^i$:

- Step 1: For every qubit g_i change its index so that it represents the coefficient of $x^{i+1 \bmod n}$. Let h_i be the coefficients of the relabeled polynomial, i.e. $h_{i+1 \bmod n} = g_i$.
- Step 2: Apply CNOT controlled by the x^0 term h_0 (g_{n-1} before Step 1) to h_j, with $j = M_1, \ldots, M_{\omega-2}$. In the example of $1 + x^3 + x^{10}$ we would apply 1 CNOT to h_3 controlled by h_0.

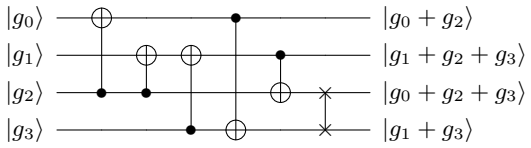
See Circuit 9 for an example. After a multiplication by x without reduction the coefficient of x^0 is always 0. As $m(x)$ is irreducible, it always has coefficient 1



Circuit 9: Binary shift circuit for $\mathbb{F}_{2^{10}}$ with $g_0 + \cdots + g_9x^9$ as the input and $h_0 + \cdots + h_9x^9 = g_9 + g_0x + g_1x^2 + (g_2 + g_9)x^3 + g_3x^4 + \cdots + g_9x^9$ as the output.

for x^0, so after a reduction by $m(x)$ that qubit will be 1 and if no reduction takes place that qubit will be 0, which means our modular shift algorithm is reversible. This results in a total of $\omega - 2$ CNOT gates for a modular reduction, with depth $\omega - 2$ and we do not use ancillary qubits. Running this circuit in reverse corresponds to dividing by x modulo $m(x)$.

5.2 Multiplication by a constant polynomial



Circuit 10: Multiplication of g by $1+x^2$ modulo $1+x+x^4$. Depth 4 and 5 CNOT gates.

Multiplication by a constant non-zero polynomial in a fixed binary field is $\mathbb{F}_2$-linear: as the field polynomial is irreducible, every input corresponds to exactly one output. We can see that any such multiplication can be represented as a matrix, which we can turn into a circuit using an LUP-decomposition, an algorithm also used by Amento, Rötteler and Steinwandt [1]. For example, multiplication by $1 + x^2$ modulo $1 + x + x^4$ can be represented by a matrix Γ. Using the decomposition $\Gamma = P^{-1}LU$ we get an upper and lower triangular matrix which we translate into a circuit. Any 1 not on the diagonal in U and L is a CNOT controlled by the column number on the row number. In cases of conflict[2], for

[2] Conflicts exist if according to the triangular matrix a CNOT would both have to be applied on and controlled by a qubit. By doing the controlled operation first and

U CNOT gates should be performed top row first, second row second and so on and for L CNOT gates from the bottom row up. P represents a series of swaps, and can be represented either as a permutation matrix or an ordered list with all elements from 0 to $n-1$.

$$\Gamma = \begin{pmatrix} 1&0&1&0 \\ 0&1&1&1 \\ 1&0&1&1 \\ 0&1&0&1 \end{pmatrix} = P^{-1}LU = \begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{pmatrix} \begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&1&1&0 \\ 1&0&0&1 \end{pmatrix} \begin{pmatrix} 1&0&1&0 \\ 0&1&1&1 \\ 0&0&1&0 \\ 0&0&0&1 \end{pmatrix}$$

Circuit 10 shows how we translate Γ. According to [1] this costs up to n^2+n CNOT gates with depth up to $2n$. We can improve this count by noting L and U are each size n by n and can have up to $(n^2-n)/2$ non-diagonal non-zero entries, giving us up to n^2-n CNOT gates. Note that the LUP-decomposition is precomputed and for any fixed polynomial and field we can give an exact CNOT gate count and depth.

Since this algorithm is introduced in [1] without correctness proof and we will use it later for a bigger algorithm, we will write an explicit implementation and go over the correctness of this algorithm. Note that since we are working with reversible gates, multiplying by constant $f(x)$ modulo $m(x)$ is the same as doing the reverse of multiplying by constant $f(x)^{-1}$ modulo $m(x)$.

Theorem 1. *Algorithm 1 correctly describes multiplication by a non-zero constant polynomial in a fixed binary field.*

Proof. Since multiplication by a non-zero constant in a finite field is a linear map, an invertible matrix Γ to represent this linear map must exist. Since Γ is invertible, its decomposition L, U, P^{-1} must also consist of invertible linear maps. Since we are working in a binary field and U is an invertible upper-triangular matrix, the diagonal of U is all-one. If we look at lines 1 through 4 of the algorithm, we can see they correspond to applying linear map U to g, as it results in $g_i = \sum_{j=0}^{n-1} u_{i,j} g_j$ for $i = 0, .., n-1$. Analogously the same is true for L in lines 5 through 8. We can also see that if P^{-1} is a row-permutation of the identity matrix, lines 9 through 13 will apply it correctly. Since $P^{-1}LU = \Gamma$ we have correctly applied the linear map Γ. □

Note that the algorithm is not optimized for depth, for example in circuit 10 the first and second CNOT could be swapped so the depth would be 3 rather than 4. The LUP-decomposition is generated automatically using primitive Gaussian elimination.

Choice of field polynomials

When doing operations in a finite binary field we can choose what representation we use, as long as the polynomial $m(x)$ is irreducible. Our goal is to make the

applying the operation on it afterwards, we ensure that the matrix multiplication is correctly translated.

Algorithm 1: $\text{MULT}_{f(x)}$, from [1]. Reversible algorithm for in-place multiplication by a nonzero constant polynomial $f(x)$ in $\mathbb{F}_2[x]/m(x)$ with $m(x)$ an irreducible polynomial.

Fixed input : A binary LUP-decomposition L, U, P^{-1} for a binary n by n matrix that corresponds to multiplication by the constant polynomial $f(x)$ in the field $\mathbb{F}_2[x]/m(x)$.
Quantum input: A binary polynomial $g(x)$ of degree up to $n-1$ stored in an array G.
Result: G as $f \cdot g$ in the field $\mathbb{F}_2/m(x)$.

```
for i = 0..n − 1                                          // U · G
do
    for j = i + 1..n − 1 do
        if U[i, j] = 1 then
            G[i] ← CNOT(G[i], G[j])
for i = n − 1..0                                          // L · UG
do
    for j = i − 1..0 do
        if L[i, j] = 1 then
            G[i] ← CNOT(G[i], G[j])
for i = 0..n                                              // P^{-1} · LUG
do
    for j = i + 1..n − 1 do
        if P^{-1}[i, j] = 1 then
            SWAP(G[i], G[j])
            SWAP column i and j of P^{-1}
```

matrices L and U as sparse as possible. For this purpose we also want our Γ to be as sparse as possible, which can be achieved in two steps: pick irreducible polynomials with as few non-zero coefficients as possible, i.e. trinomials when available and pentanomials otherwise, and pick irreducible polynomials where the second highest non-constant term has the lowest possible degree. For example, the pentanomial $1 + x^3 + x^4 + x^{19} + x^{20}$ requires 108 CNOT gates, the pentanomial $1 + x^3 + x^5 + x^9 + x^{20}$ requires 55 CNOT gates, while the trinomial $1 + x^3 + x^{20}$ requires only 27. All 3 polynomials are irreducible. In Table 1 we can see some examples of gate counts for various choices of n. The depth count is an upper bound without accounting for swapping gates.

5.3 Squaring

Squaring in $\mathbb{F}_{2^n}$ is a lot easier than in the general case since:

$$\left(\sum_{i=0}^{n-1} a_i x^i\right)^2 = \sum_{i=0}^{n-1} a_i \cdot x^{2 \cdot i} \bmod m(x)$$

Degree	Irreducible polynomial	Source	CNOT gates	Depth upper bound
4	$[4, 1, 0]$	[2]	5	4
8	$[8, 4, 3, 1, 0]$	[2]	20	14
16	$[16, 5, 3, 1, 0]$	[2]	47	30
32	$[32, 7, 3, 2, 0]$	[2]	133	93
64	$[64, 4, 3, 1, 0]$	[2]	264	182
127	$[127, 1, 0]$	[2]	396	293
128	$[128, 7, 2, 1, 0]$	[2]	626	443
163	$[163, 7, 6, 3, 0]$	[14]	740	975
163	$[163, 89, 74, 15, 0]$	[3]	1885	1646
233	$[233, 74, 0]$	[14]	3319	2976
256	$[256, 10, 5, 2, 0]$	[2]	1401	1030
283	$[283, 12, 7, 5, 0]$	[14]	2117	1700
283	$[283, 160, 123, 37, 0]$	[3]	6785	6368
571	$[571, 10, 5, 2, 0]$	[14]	4027	3177
571	$[571, 353, 218, 135, 0]$	[3]	33182	32331
1024	$[1024, 19, 6, 1, 0]$	[21]	8147	6624

Table 1: Comparison of the CNOT gates required for various instances of Algorithm 1. Source is the source of the polynomial.

If we do not consider the mod operation, this would be 'free,' as we just need to shuffle zeroes between our registers. We can see two approaches for squaring in $\mathbb{F}_{2^n}$: a circuit that takes the result of squaring a polynomial of degree at most $n-1$ and stores it in n separate qubits, or a circuit that replaces the input with the result. The second approach is only possible for finite fields with 2^n elements since every square is unique.

Squaring and replacing the input: Muñoz-Coreaz and Thapliyal [16] propose a design which uses a small number of gates for reversible squaring by shuffling the qubits cleverly. The number of CNOT gates saved for their squaring is equal to n, and they use no ancillary qubits. Their algorithm as proposed, however, does not take into account cases where qubits in the upper $\lfloor \frac{n}{2} \rfloor$ registers have to interact[3]. Instead, we can use another LUP-decomposition since squaring is also a linear map.

Squaring and storing the result separately: For this approach, we can take schoolbook squaring mod $m(x)$: for every i from 0 to $n-1$ add $a_i x^{2i}$ mod $m(x)$

[3] For example, if $n = 8$ and $m(x) = x^8+x^4+x^3+x+1$, we have $x^{6\cdot 2} = x^7+x^5+x^3+x+1$. This means the qubit corresponding to x^6 in the input has to be added to qubits that also have to add themselves to the qubit corresponding to x^6 in the input, regardless of which output qubit you use to represent input qubits x^4, x^5, x^7. Solutions (like the LUP-decomposition) exist, as squaring in $\mathbb{F}_{2^n}$ is a linear map, but these solutions are not provided by the authors of [16] in the paper.

to the output qubits which start in state 0^n. Since $m(x)$ is known we can exactly compute the number of CNOT gates required depending on it. For example, squaring modulo $1 + x^3 + x^{10}$ requires 16 CNOT gates, with $O(n)$ CNOT gates in general.

6 Quantum Multiplication for binary polynomials

This section details schoolbook multiplication and we present our new Karatsuba algorithm.

6.1 Quantum Schoolbook Multiplication

The simplest way to multiply is schoolbook multiplication. For two polynomials of degree at most $n-1$ that takes n^2 Toffoli gates, the number of pairs of qubits from the first and second polynomial. While the computation does not use ancillary qubits, the result needs to be stored separately from the input in $2n-1$ qubits; unlike the previous circuits we cannot replace either of the inputs with the result since the Toffoli gate requires a separate output. If we want to apply modular reduction by a weight-ω and degree-n odd polynomial, this adds $(n-1)\cdot(\omega-2)$ CNOT gates and uses no ancillary qubits (by using the modular shift algorithm after every n multiplications). The result is stored in n qubits.

6.2 Classic Karatsuba multiplication in binary polynomial rings

Rather than using schoolbook multiplication, methods like Karatsuba multiplication [12] can speed up multiplication of high-degree polynomials in the classical case. We can look at in-place multiplication in the classical case for ideas [19]. As input we take two polynomials of degree up to $n-1$, $f(x)$ and $g(x)$ as well as a polynomial of size $2n-1$: $h(x)$. As output we want to calculate $h + f \cdot g$. For some k such that $\frac{n-1}{2} \leq k < n-1$ (we will always use $k = \lceil \frac{n-1}{2} \rceil$) we can split each polynomial as follows: $f = f_0 + f_1 x^k$, $g = g_0 + g_1 x^k$ and $h = h_0 + h_1 x^k + h_2 x^{2k} + h_3 x^{3k}$.

We compute intermediate products $\alpha = f_0 \cdot g_0$, $\beta = f_1 \cdot g_1$ and $\gamma = (f_0 + f_1) \cdot (g_0 + g_1)$. Finally, we add these in the right way for Karatsuba multiplication:

$$h + f \cdot g = h + \alpha + (\gamma + \alpha + \beta)x^k + \beta x^{2k}.$$

For cleanliness, we can split up our α, β, γ in the same way as f and g to get a result with no overlap, which is useful for checking correctness:

$$h+f\cdot g = (h_0+\alpha_0)+(h_1+\alpha_0+\alpha_1+\beta_0+\gamma_0)x^k+(h_2+\alpha_1+\beta_0+\beta_1+\gamma_1)x^{2k}+(h_3+\beta_1)x^{3k}.$$

Alternatively, we can rewrite this another way that will prove useful:

$$h + f \cdot g = h + (1 + x^k)\alpha + x^k\gamma + x^k(1 + x^k)\beta.$$

6.3 Reversible Karatsuba multiplication in binary polynomial rings

Based on these equations we can split our multiplication algorithm into 2 parts: given $f(x), g(x), h(x)$ calculate $h + f \cdot g$ and given $k, f(x), g(x), h(x)$ with $k > \max(\deg(f), \deg(g))$ calculate $h + (1 + x^k)f \cdot g$. We will look at our algorithms for the 2 parts, which can then be used recursively to provide a significant improvement to the schoolbook algorithm in terms of Toffoli gate count.

Algorithm 2: MULT1x$_k$. Reversible algorithm for multiplication by the polynomial $1 + x^k$.

Fixed input : A constant integer $k > 0$ to indicate part size as well as an integer $n \leq k$ to indicate polynomial size.
$\ell = \max(0, 2n - 1 - k)$ is the size of h_2 and $(fg)_1$. In the case of Karatsuba we will have either $n = k$ or $n = k - 1$.

Quantum input: Two binary polynomials $f(x), g(x)$ of degree up to $n - 1$ stored in arrays A and B respectively of size n. A binary polynomial $h(x)$ of degree up to $k + 2n - 2$ stored in array C of size $2k + \ell$.

Result: A and B as input, C as $h + (1 + x^k)fg$

1 **if** $n > 1$ **then**
2 $\quad C[k..k + \ell - 1] \leftarrow \text{CNOT}(C[k..k + \ell - 1], C[2k..2k + \ell - 1])$
3 $\quad C[0..k - 1] \leftarrow \text{CNOT}(C[0..k - 1], C[k..2k - 1])$
4 $\quad C[k..2k + \ell - 1] \leftarrow \text{KMULT}(A[0..n - 1], B[0..n - 1], C[k..2k + \ell - 1])$
5 $\quad C[0..k - 1] \leftarrow \text{CNOT}(C[0..k - 1], C[k..2k - 1])$
6 $\quad C[k..k + \ell - 1] \leftarrow \text{CNOT}(C[k..k + \ell - 1], C[2k..2k + \ell - 1])$
7 **else**
8 $\quad C[0] \leftarrow \text{CNOT}(C[0], C[k])$
9 $\quad C[k] \leftarrow \text{TOF}(A[0], B[0], C[k])$
10 $\quad C[0] \leftarrow \text{CNOT}(C[0], C[k])$

Line	C in MULT1x$_k$		
	$C[0..k-1]$	$C[k..2k-1]$	$C[2k..2k+\ell-1]$
1	h_0	h_1	h_2
2	h_0	$h_1 + h_2$	h_2
3	$h_0 + h_1 + h_2$	$h_1 + h_2$	h_2
4	$h_0 + h_1 + h_2$	$h_1 + h_2 + (fg)_0$	$h_2 + (fg)_1$
5	$h_0 + (fg)_0$	$h_1 + h_2 + (fg)_0$	$h_2 + (fg)_1$
6	$h_0 + (fg)_0$	$h_1 + (fg)_0 + (fg)_1$	$h_2 + (fg)_1$

Table 2: Step by step calculation of Algorithm 2.

Lemma 1. *Given polynomials f, g of degree up to $n-1$ with $n > 1$, polynomial h of degree up to $k + 2n - 2$ with some $k \geq n$ and assuming Algorithm 3 correctly calculates $h + fg$ with degrees of f, g and h bounded as above, Algorithm 2 correctly calculates $h + (1 + x^k)fg$ in $\mathbb{F}_2[x]$ without altering the values of f and g.*

Proof. Let $\ell = \max(0, 2n - 1 - k)$. Table 2 gives the result of each step on array C, split into 3 parts of size k, k and $\ell - 1$ respectively: $h = h_0 + h_1 x^k + h_2 x^{2k}$. The final result corresponds to $h_0 + (fg)_0 + (h_1 + (fg)_0 + (fg)_1)x^k + (h_2 + (fg)_1)x^{2k} = h_0 + h_1 x^k + h_2 x^{2k} + fg + fgx^k = h + (1 + x^k)fg$, where $(fg)_0$ is the first k terms of $f \cdot g$ and $(fg)_1$ is the last up to ℓ terms.

f and g do not have their values altered because arrays A and B remain unchanged. □

Algorithm 2 computes $h + (1 + x^k)fg$ with at most $2k + 2\ell \geq 2k + 2(2n - 1 - k) = 4n - 2$ CNOT gates, at a depth of 4 per layer and 1 call to Algorithm 3 for an n-by-n multiplication. For $n = 1$ both the depth and number of gates is 2 CNOT and 1 TOF gates.

Algorithm 3: KMULT. Reversible algorithm for multiplication of 2 polynomials.

Fixed input : A constant integer n to indicate polynomial size and an integer $k < n \leq 2k$ with $k = \lceil \frac{n}{2} \rceil$ for $n > 1$ and $k = 0$ for $n = 1$, to indicate upper and lower half.

Quantum input: Two binary polynomial f, g of degree up to $n - 1$ stored in arrays A and B respectively of size n. A binary polynomial h of degree up to $2n - 2$ stored in array C of size $2n - 1$.

Result: A and B as input, C as $h + fg$

if $n > 1$ **then**
- $C[0..3k-2] \leftarrow \text{MULT1x}_k(A[0..k-1], B[0..k-1], C[0..3k-2])$
- $C[k..2n-2] \leftarrow \text{MULT1x}_k(A[k..n-1], B[k..n-1], C[k..2n-2])$
- $A[0..n-k-1] \leftarrow \text{CNOT}(A[0..n-k-1], A[k..n-1])$
- $B[0..n-k-1] \leftarrow \text{CNOT}(B[0..n-k-1], B[k..n-1])$
- $C[k..3k-2] \leftarrow \text{KMULT}(A[0..k-1], B[0..k-1], C[k..3k-2])$
- $B[0..n-k-1] \leftarrow \text{CNOT}(B[0..n-k-1], B[k..n-1])$
- $A[0..n-k-1] \leftarrow \text{CNOT}(A[0..n-k-1], A[k..n-1])$

else
- $C[0] \leftarrow \text{TOF}(A[0], B[0], C[0])$

Lemma 2. *Let $k = \lceil \frac{n}{2} \rceil$. Given polynomials f, g of degree up to $n - 1$ with $n > 1$ and h of degree up to $2n - 2$. Assuming Algorithm 2 correctly calculates $h' + (1 + x^k)f'g'$ for f', g' up to degree $k - 1$ and h' up to degree $3k - 2$, and Algorithm 3 correctly calculates $h'' + f''g''$ with f'', g'' of degree $k - 1$ and h'' of degree $2k - 2$ without altering the values of f'' and g''. Then Algorithm 3*

correctly calculates $h + fg$ in $\mathbb{F}_2[x]$. The values of f and g are the same after the algorithm as they were before.

Proof. Table 3 gives the result of each line on array C, split into 4 parts of size k, k, k and $2n-1-3k$ respectively: $h = h_0 + h_1x^k + h_2x^{2k} + h_3x^{3k}$. As can be seen in the table, the final result corresponds to $(h_0+\alpha_0)+(h_1+\alpha_0+\alpha_1+\beta_0+\gamma_0)x^k + (h_2+\alpha_1+\beta_0+\beta_1+\gamma_1)x^{2k} + (h_3+\beta_1)x^{3k} = h + f \cdot g$ as discussed in Section 6.2. Lines 7 and 8 are the inverses of lines 4 and 5 so return A and B to their original states. □

Line	C in KMULT			
	$C[0..k-1]$	$C[k..2k-1]$	$C[2k..3k-1]$	$C[3k..2n-2]$
1	h_0	h_1	h_2	h_3
2	$h_0+\alpha_0$	$h_1+\alpha_0+\alpha_1$	$h_2+\alpha_1$	h_3
3-5	$h_0+\alpha_0$	$h_1+\alpha_0+\alpha_1+\beta_0$	$h_2+\alpha_1+\beta_0+\beta_1$	$h_3+\beta_1$
6-8	$h_0+\alpha_0$	$h_1+\alpha_0+\alpha_1+\beta_0+\gamma_0$	$h_2+\alpha_1+\beta_0+\beta_1+\gamma_1$	$h_3+\beta_1$

Table 3: Step by step calculation of Algorithm 3.

Algorithm 3 computes $h+fg$ with $4(n-k)$ CNOT gates, at a depth of 4, 1 call to itself for a k-by-k multiplication, 1 call to Algorithm 2 for a k-by-k multiplication and 1 call to Algorithm 2 for an $(n-k)$-by-$(n-k)$ multiplication. For $n = 1$ we just have a single TOF gate.

Theorem 2. *Given polynomials f, g of degree up to $n-1$ and h of degree up to $2n-2$, Algorithm 3 correctly calculates $h + fg$. The values of f and g are the same after the algorithm as they were before.*

Proof. We use proof by induction. For $n = 1$ line 10 of Algorithm 3 correctly calculates $h + fg$ without altering f or g.

For $n = 2$ two calls are made to Algorithm 2 and one call to Algorithm 3 with $n' = 1$ and $k' = 1$. Lines 7-9 of Algorithm 2 correctly calculate $h' + (1 + x^k)f'g'$.

For $n > 2$ we use lemmas 1 and 2 as our inductive steps. Every time Algorithm 3 is called recursively to calculate $h' + f'g'$ with f', g' of degree $n'-1$, it is with either $n' = \lceil \frac{n}{2} \rceil$ or $n' = n - \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$.

The series $\lceil \frac{n}{2} \rceil, \lceil \frac{\lceil \frac{n}{2} \rceil}{2} \rceil, \lceil \frac{\lceil \frac{\frac{n}{2}}{2} \rceil}{2} \rceil, \ldots$ reaches 1 in $O(\log n)$ steps and $\lfloor \frac{n}{2} \rfloor \leq \lceil \frac{n}{2} \rceil$. From this we can see that we reach $n' = 1$ or 2 in a finite number of steps. By induction Algorithm 3 correctly calculates $h + fg$ and returns f and g to their original values. □

7 Reversible Karatsuba multiplication in binary finite fields

With the multiplication methods from the previous section, we can move on to the modular multiplication. We will need Algorithm 1, which we will also run in

Algorithm 4: MODMULT. Reversible algorithm for multiplication of 2 polynomials in $\mathbb{F}_2[x]/m(x)$ with $m(x)$ an irreducible polynomial.

Fixed input : A constant integer n to indicate field size, $k = \lceil \frac{n}{2} \rceil$. $m(x)$ of degree n as the field polynomial. The LUP-decomposition precomputed for multiplication by $1 + x^k$ modulo $m(x)$.

Quantum input: Three binary polynomials $f(x), g(x), h(x)$ of degree up to $n-1$ stored in arrays A, B, C respectively of size n.

Result: A and B as input, C as $h + f \cdot g \mod m$.

for $i = 0..k-1$ **do**
  $C[0..n-1] \leftarrow \text{MODSHIFT}^{-1}(C[0..n-1])$
$A[0..n-k-1] \leftarrow \text{CNOT}(A[0..n-k-1], A[k..n-1])$
$B[0..n-k-1] \leftarrow \text{CNOT}(B[0..n-k-1], B[k..n-1])$
$C[0..n-1] \leftarrow \text{KMULT}(A[0..k-1], B[0..k-1], C[0..n-1])$
$B[0..n-k-1] \leftarrow \text{CNOT}(B[0..n-k-1], B[k..n-1])$
$A[0..n-k-1] \leftarrow \text{CNOT}(A[0..n-k-1], A[k..n-1])$
$C[0..n-1] \leftarrow \text{MULT}^{-1}_{1+x^k}(C[0..n-1])$
$C[0..n-1] \leftarrow \text{KMULT}(A[k..n-1], B[k..n-1], C[0..n-1])$
for $i = 0..k-1$ **do**
  $C[0..n-1] \leftarrow \text{MODSHIFT}(C[0..n-1])$
$C[0..n-1] \leftarrow \text{KMULT}(A[0..k-1], B[0..k-1], C[0..n-1])$
$C[0..n-1] \leftarrow \text{MULT}_{1+x^k}(C[0..n-1])$

reverse for multiplication by an inverse of the polynomial, and the binary shifts from Section 5.1, which we will refer to as MODSHIFT (MODSHIFT^{-1} for the reverse), as well as the previous Karatsuba algorithms. We can see in Algorithm 4 the number of operations we use:

- 3 calls to Algorithm 3: twice for k-by-k multiplication and once for $(n-k)$-by-$(n-k)$ multiplication.
- 2 calls to Algorithm 1 (once in reverse), each time for multiplication by the same polynomial $1 + x^k$.
- $2k$ calls to MODSHIFT.
- 4 times $(n-k)$ CNOT gates, half of which can be performed at the same time.

Note that Algorithm 3 can multiply two polynomials f and g of degree at most $\lceil \frac{n}{2} \rceil - 1$ while needing n qubits for the output polynomial h, which has degree $n-1$ at most in the case that n is odd. We make recursive calls to Algorithm 3 rather than Algorithm 4 because it uses significantly fewer CNOT operations and fits in the required space.

In total, this algorithm uses $n^{\log 3} + O(1)$ TOF gates, with $O(1)$ being zero if n is a power of two. The exact number of TOF and CNOT gates are in Table 5. Section 10 details the comparison to other multiplication algorithms.

Theorem 3. *Algorithm 4 correctly calculates fg in a field $\mathbb{F}_2[x]/m(x)$ and the values of f and g are the same after the algorithm as they were before.*

Line	C in MODMULT
1	h
2-4	$x^{-k}h \mod m$
5-7	$x^{-k}h + \gamma \mod m$
8	$(1+x^k)^{-1}(x^{-k}h + \gamma) \mod m$
9	$(1+x^k)^{-1}(x^{-k}h + \gamma) + \beta \mod m$
11	$(1+x^k)^{-1}(h + x^k\gamma) + x^k\beta \mod m$
12	$(1+x^k)^{-1}h + \alpha + x^k(1+x^k)^{-1}\gamma + x^k\beta \mod m$
13	$h + (1+x^k)\alpha + x^k\gamma + x^k(1+x^k)\beta \mod m$

Table 4: Step-by-step calculation of Algorithm 4.

Proof. Table 4 gives the result of each line on array C. As can be seen in the table, the final result corresponds to $h + (1+x^k)\alpha + x^k\gamma + x^k(1+x^k)\beta \mod m$. Lines 6 and 7 are the inverses of lines 3 and 4 so return A and B to their original states. □

Degree	schoolbook TOF gates	Algorithm 4 TOF gates	CNOT gates	Depth upper bound
2	4	3	9	9
4	16	9	44	32
8	64	27	200	124
16	256	81	678	365
32	1,024	243	2,238	1,110
64	4,096	729	6,896	3,129
127	16,129	2,185	20,632	8,769
128	16,384	2,187	21,272	9,142
163	26,569	4,387	37,168	17,906
233	54,289	6,323	63,655	29,530
256	65,536	6,561	64,706	26,725
283	80,089	10,273	89,620	41,548
571	326,041	31,171	270,940	121,821
1024	1,048,576	59,049	591,942	234,053

Table 5: CNOT and TOF gate count and depth upper bounds for various instances of Algorithm 4 as well as TOF gate count for schoolbook multiplication. Field polynomials used are the same as in Table 1, with the irreducible polynomial chosen that has the lowest CNOT count.

8 Inversion and division in binary finite fields

The most computationally intensive step is the division step. For the purpose of this paper we treat division as multiplication by the inverse of a polynomial.

Usually there are two different ways these inverses are calculated, which we compare in this section.

8.1 Inversion using extended GCD

The first variant we look at is using the extended greatest common divisor (GCD) or Euler's algorithm. Roetteler, Naehrig, Svore and Lauter [20] propose a straightforward variant using Kaliski's binary GCD algorithm for inversion in $\mathbb{F}_p$. In the quantum setting this has a problem because Kaliski's algorithm terminates in a number of steps dependent on the input polynomial. To circumvent this, a qubit stores whether the algorithm has terminated and $\log(n)$ qubits store how long ago the algorithm terminated. This ends up introducing a rather large number of conditional CNOT and conditional Toffoli gates at each step, which balloons the total Toffoli gate cost. This algorithm ends up having $32n^2 \log(n)$ Toffoli gates while using only $8n + 2\lceil \log(n) \rceil + 9$ qubits.

Instead, we use a constant time gcd variant introduced by Bernstein and Yang [4] as the basis of our new quantum algorithm. They introduce a classical gcd-based inversion algorithm which runs in constant time. Our translation to a reversible algorithm runs in fewer Toffoli gates than a naive implementation of a greatest common divisor algorithm. In order to help with the simplification of some of the steps, we base our algorithm on an optimized variant from section 7 of the paper. Specifically, it allows us to restrict the sizes of 2 variables to $n+1$ qubits rather than the less optimized variant from section 6 of the paper, which use up to $2n$ qubits, and gets rid of 2 variables which are not relevant for the calculation of the inverse.

Using these strategies, we arrive at Algorithm 5. The loop is repeated $2n-1$ times uses the following actions:

- RIGHTSHIFT and LEFTSHIFT shift the contents using only swap gates. We see these as free.
- a is the qubit used to decide whether to swap or not. Since v is always odd after a swap takes place and even if no swap has taken place, we can uncompute it directly. Unfortunately, v is always even before the swap takes place and whether r is odd depends on g, so keeping track of the sign is necessary.
- δ is a number from 1 to $2n$. In [4] this is a number from $-n$ to n but by making it a positive number we get an important advantage: by making the number 1 in the original equal to $2^{\lfloor \log(n) \rfloor + 1}$ the most significant qubit of δ will correspond to the sign and changing the sign is done by a series of CNOT gates. This will also increment δ, which is why δ is only incremented with the incrementer circuit if a is 0.
- CSWAP is a conditional swap using 2 CNOT and 1 TOF gate to swap 2 qubits based on a.
- g_0 is not possible to uncompute within a single step. In [20] a similar value, called m_i, is stored. We reduce some of the space by observing that f and g start to decrease in size after n steps but at step n the registers v, r, f, g, g_0

Algorithm 5: GCD_DIV. Reversible algorithm for division using inversion with an extended GCD algorithm. CNOT$(\delta[0,...,\lfloor\log(n)\rfloor+1],a)$ is shorthand for CNOT$(\delta[0],a),...,$CNOT$(\delta[\lfloor\log(n)\rfloor+1],a)$ and similar shorthand is used for TOF gates.

Fixed input : A constant field polynomial m of degree $n>0$ as an array M as in Subsection 5.1, $\Lambda=\min(2n-2-\ell,n)$ and $\lambda=\min(\ell+1,n)$.

Quantum input:

- A non-zero binary polynomial $R_1(x)$ of degree up to $n-1$ stored in array g of size n to invert.
- A binary polynomial $R_2(x)$ of degree up to $n-1$ to multiply with the inverse stored in array B.
- A binary polynomial $R_3(x)$ of degree up to $n-1$ for the result stored in array C.
- 4 arrays of size $n+1$: f, r, v, g_0 initialized to an all-$|0\rangle$ state.
- 1 array of size $\lceil\log(n)\rceil+2$ initialized to an all-$|0\rangle$ state: δ, which will be treated as an integer.
- 2 qubits to store ancillary qubits $a, g[n]$ initialized to $|0\rangle$.
- Refer to $g[n], g[n-1], ..., g[3]$ as $g_0[n+1], g_0[n+2], ..., g_0[2n-2]$ when applicable.
- Refer to $\delta[\lfloor\log(n)\rfloor+1]$ as sign with sign $=1$ if $\delta\geq 2^{\lfloor\log(n)\rfloor+1}$ and 0 otherwise.

Result: Everything except C the same as their input, C as R_3+R_2/R_1

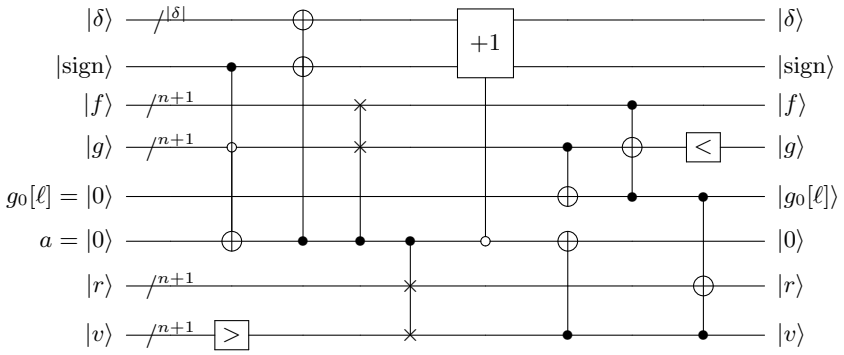
1 **for** i *in* M **do**
2 $\quad f[n-i]\leftarrow|1\rangle$ `// Reverse` m
3 sign $\leftarrow|1\rangle$
4 $r[0]\leftarrow|1\rangle$
5 **for** $i=0,...,\lfloor\frac{n}{2}\rfloor-1$ **do**
6 $\quad$ SWAP$(g[i],g[n-1-i])$ `// Reverse` g
7 **for** $\ell=0,...,2n-2$ **do**
8 $\quad v[0,...,n]\leftarrow$ RIGHTSHIFT$(v[0,...,n])$
9 $\quad a\leftarrow$ TOF$(\text{sign},g[0],a)$
10 $\quad \delta[0,...,\lfloor\log(n)\rfloor+1]\leftarrow$ CNOT$(\delta[0,...,\lfloor\log(n)\rfloor+1],a)$
11 $\quad$ CSWAP$_a(f[0,...,\Lambda],g[0,...,\Lambda])$ `//` $\Lambda=\min(2n-2-\ell,n)$
12 $\quad$ CSWAP$_a(r[0,...,\lambda],v[0,...,\lambda])$ `//` $\lambda=\min(\ell+1,n)$
13 $\quad \delta[0,...,\lfloor\log(n)\rfloor+1]\leftarrow$ INC$_{1+a}(\delta[0,...,\lfloor\log(n)\rfloor+1])$
14 $\quad a\leftarrow$ CNOT$(a,v[0])$ `// Uncompute` a
15 $\quad g_0[\ell]\leftarrow$ CNOT$(g_0[\ell],g[0])$
16 $\quad g[0,...,\Lambda]\leftarrow$ TOF$(f[0,...,\Lambda],g_0[\ell],g[0,...,\Lambda])$ `//` $\Lambda+1$ `TOF gates`
17 $\quad r[0,...,\lambda]\leftarrow$ TOF$(v[0,...,\lambda],g_0[\ell],r[0,...,\lambda])$ `//` $\lambda+1$ `TOF gates`
18 $\quad g[0,...,\Lambda]\leftarrow$ LEFTSHIFT$(g[0,...,\Lambda])$
19 **for** $i=0,...,\lfloor\frac{n}{2}\rfloor-1$ **do**
20 $\quad$ SWAP$(v[i],v[n-1-i])$
21 $C[0,...,n-1]\leftarrow$ MODMULT$(v[0,...,n-1],B[0,...,n-1],C[0,...,n-1])$
22 UNCOMPUTE lines 1-20

all need mostly full $n+1$ qubit arrays. This means the number of qubits for these arrays is $5n + O(1)$ at least.

- INC_{1+a} is a controlled incrementing algorithm. Using the n borrowed bits design from [7] (we easily have $\log(n)$ qubits laying around for borrowing), we turn the CNOT gates into TOF and TOF into 3 TOF gates using ancillary qubit $g_0[\ell]$ at step ℓ which is still zero at this point. This leads to $22\lfloor\log(n)\rfloor + 26$ TOF gates and $2\lfloor\log(n)\rfloor + 3$ CNOT gates.
- In total we get $2(\Lambda + \lambda) + 5$ TOF gates at step ℓ and $4(+\lambda) + 3$ CNOT gates in addition to the gates from INC, with $\Lambda = \min(2n - 2 - \ell, n)$ and $\lambda = \min(\ell + 1, n)$.

By keeping track of the maximum sizes of f, g, v, r we get two distinct benefits: the CSWAP and TOF steps take fewer gates and we free up some space to store some of the decisional qubits. On average, both Λ and λ are size $\frac{3}{4}n + O(1)$ since we have $n - 1$ steps of size n and n steps where the size is increasing or decreasing by 1 per step.

We need to do the loop $4n-2$ times in total: $2n-1$ for computing and $2n-1$ for uncomputing. Not including the multiplication, this gives us $12n^2 + (88n - 44)\lfloor\log(n)\rfloor + 116n - 62$ TOF gates and $24n^2 + 8n\lfloor\log(n)\rfloor + O(n)$ CNOT gates while using $4n + \lfloor\log(n)\rfloor + 8$ ancillary qubits plus $3n$ qubits for the input and output qubits.



Circuit 11: Step ℓ of Algorithm 5. $|\delta| = \lfloor\log(n)\rfloor + 1$.

8.2 Inversion using FLT

Fermat's little theorem (FLT) states $x^p = x \mod p$. This can be extended for binary finite fields to $f^{2^n-2} = f^{-1} \mod m(x)$ where n is the degree of $m(x)$. By using squarings we can compute this in n multiplications and $n-1$ squarings: $f^{2^n-2} = f^2 \cdot f^{2^2} \cdot f^{2^3} \cdot \ldots \cdot f^{2^{n-1}}$. However, improvements to this straightforward

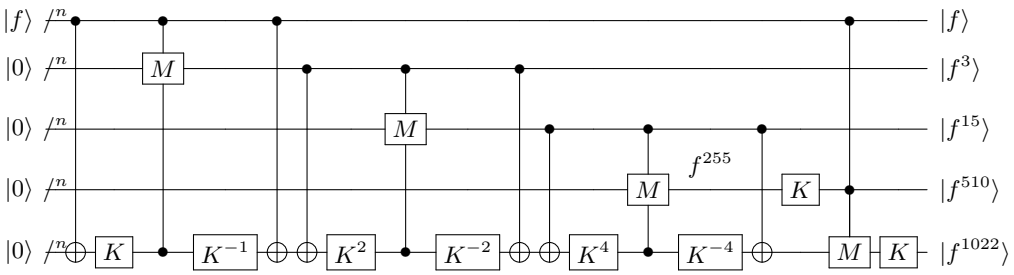
method exist. Itoh and Tsuji[4] [11] give three improved variants. We use the second variant (theorem 2 in [11]) since the third variant, despite giving better results, requires n to be a product of two integers, meaning it cannot be used for n prime like the NIST curves [14] use.

This algorithm uses two observations:

- $f^{2^n-2} = (f^{2^{n-1}-1})^2$
- $f^{2^{2^t}-1} = (f^{2^{2^{t-1}}-1})^{2^{2^{t-1}}}(f^{2^{2^{t-1}}-1})$

to reduce the cost to below $2\log(n)$ multiplications and to $n-1$ squarings. This algorithm works as follows:

0. Write $n-1$ as $[k_1, ..., k_t]$ with $\sum_{s=1}^{t} 2^{k_s} = n-1$ and $k_1 > k_2 > ... > k_t \geq 0$. Note t is the Hamming weight of $n-1$ in binary and $t \leq \lfloor \log(n-1) \rfloor + 1$ and $k_1 = \lfloor \log(n-1) \rfloor$.
1. Calculate $f^{2^{2^{k_1}}-1}$ with k_1 multiplications using the second observation, save the intermediate results $f^{2^{2^{k_t}}-1}, f^{2^{2^{k_{t-1}}}-1}, ..., f^{2^{2^{k_1}}-1}$.
2. Calculate $f^{2^{n-1}-1} = \{...\{(f^{2^{2^{k_1}}-1})^{2^{2^{k_2}}}(f^{2^{2^{k_2}}-1})\}^{2^{2^{k_3}}}...\}^{2^{2^{k_t}}}(f^{2^{2^{k_t}}-1})$ using $t-1$ multiplications.
3. Square the result to get f^{-1}.



Circuit 12: Step 1-3 of Algorithm 6 for $n = 10$. K is the squaring circuit using a LUP-decomposition and M is MODMULT. $[k_1, k_2] = [3, 0]$, $2^{2^1} - 1 = 3$, $2^{2^2} - 1 = 15$, $2^{2^3} - 1 = 255$.

In total we have $k_1 + t - 1$ multiplications, which in the quantum case translates to $2n^{\log(3)}(k_1 + t - \frac{1}{2})$ TOF gates and $n \cdot \max(k_1 + t - 1, k_1 + 1)$ ancillary qubits. The classic algorithm uses $n-1$ squarings, while we have to use up to $4n-4$. We use $O(n^2)$ CNOT gates per squaring as explained in Section 5.3, but we cannot be more accurate about the number of CNOT gates for general n due to the variance in the squaring algorithm. We can get the exact number of CNOT gates using an LUP-decomposition. A full division algorithm is given

[4] They cite an unpublished manuscript by Scott A. Vanstone as having found a similar algorithm for the second theorem independently a year earlier, 1987.

Algorithm 6: FLT_DIV. Reversible algorithm for division using inversion with Fermat's little theorem.

Fixed input : A constant field polynomial $m(x)$ of degree $n > 0$.
$k_1 > k_2 > ... > k_t \geq 0$ such that $\sum_{s=1}^{t} 2^{k_s} = n - 1$.
$k = \max(k_1 + t - 1, k_1 + 1)$.

Quantum input:

- A non-zero binary polynomials $R_1(x)$ of degree up to $n-1$ stored in array f_0 of size n to invert.
- A binary polynomial $R_2(x)$ of degree $n-1$ to multiply with the inverse stored in array B.
- A binary polynomial $R_3(x)$ of degree $n-1$ for the result stored in array C.
- k zero arrays of size n initialized to an all-$|0\rangle$ state: $f_1, ..., f_k$.

Result: Everything except C as input, C as $R_3 + R_2/R_1$

1 **for** $i = 1, ..., k_1$ **do**
2 $\quad f_k[0, ..., n-1] \leftarrow \text{CNOT}(f_k[0, ..., n-1], f_{i-1}[0, ..., n-1])$ `// Step 1`
3 $\quad$ **for** $j = 1, ..., 2^{i-1}$ **do**
4 $\quad\quad f_k[0, ..., n-1] \leftarrow \text{SQUARE}(f_k[0, ..., n-1])$
5 $\quad f_i[0, ..., n-1] \leftarrow \text{MODMULT}(f_{i-1}[0, ..., n-1], f_k[0, ..., n-1], f_i[0, ..., n-1])$
6 $\quad$ **for** $j = 1, ..., 2^{i-1}$ **do**
7 $\quad\quad f_k[0, ..., n-1] \leftarrow \text{SQUARE}^{-1}(f_k[0, ..., n-1])$
8 $\quad f_k[0, ..., n-1] \leftarrow \text{CNOT}(f_k[0, ..., n-1], f_{i-1}[0, ..., n-1])$
9 **for** $s = 1, ..., t-1$ `// Step 2`
10 **do**
11 $\quad$ **for** $i = 1, ..., 2^{k_{s+1}}$ **do**
12 $\quad\quad f_{k_1+s-1}[0, ..., n-1] \leftarrow \text{SQUARE}(f_{k_1+s-1}[0, ..., n-1])$
13 $\quad f_{k_1+s}[0, ..., n-1] \leftarrow$
$\quad\quad \text{MODMULT}(f_{k_1+s-1}[0, ..., n-1], f_{k_{s+1}}[0, ..., n-1], f_{k_1+s}[0, ..., n-1])$
14 **if** $t = 1$ **then**
15 $\quad \text{SWAP}(f_{k_1}, f_k)$
16 $f_k[0, ..., n-1] \leftarrow \text{SQUARE}(f_k[0, ..., n-1])$ `// Step 3`
17 $C[0, ..., n-1] \leftarrow \text{MODMULT}(f_k, B[0, ..., n-1], C[0, ..., n-1])$
18 UNCOMPUTE lines 1-16

in Algorithm 6. We can save up to $n(k_1 - t)$ qubits by doing additional multiplications to uncompute intermediate results, at the cost of a significant number of Toffoli gates. We leave it to future work how many qubits we can save for specific fields.

8.3 Comparison of the two division algorithms

In the division branch of [24] we have given a full implementation of both division algorithms for the purpose of comparison. As can be seen in Table 6 the algorithms have different strengths. For small n ($n < 12$ or $n = 13$) the FLT-based algorithm performs better in both number of qubits and Toffoli gate count, for larger n the GCD-based algorithm performs better in number of qubits. For any n the GCD-based algorithm performs better in CNOT gate count, with roughly half the gate count of the FLT-based algorithm. The FLT-based algorithm uses roughly a fifth of the Toffoli gates used by the GCD-based algorithm while using roughly twice the number of qubits. Due to the lower space requirement of the GCD-based algorithm we use it in the remainder of the work despite the larger Toffoli gate cost.

n	GCD				FLT			
	TOF	CNOT	qubits	depth	TOF	CNOT	qubits	depth
8	3,641	1,516	67	4113	243	2,212	56	1314
16	10,403	5,072	124	12,145	1,053	10,814	144	5968
127	277,195	227,902	903	378,843	50,255	502,870	1,778	203,500
163	442,161	375,738	1,156	612,331	83,353	906,170	1,956	451,408
233	827,977	743,136	1,646	1,172,733	132,783	1,486,464	3,029	640,266
283	1,202,987	1,088,400	1,997	1,708,863	236,279	2,708,404	3,962	1,434,686
571	4,461,673	4,266,438	4,014	6,494,306	814,617	10,941,536	9,136	6,151,999

Table 6: Comparison of various instances of division Algorithms 5 and 6. Field polynomials from Table 1. Depths and gate count are upper bounds since a generic algorithm is used rather than optimizing for specific fields.

9 Point addition

With every type of basic operation covered we now describe how to do point addition on binary elliptic curves.

9.1 Classic point addition

Consider the following case from Section 2.3: we have two non-zero points on our elliptic curve, $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ with $x_1 \neq x_2$. We want to find $P_1 + P_2 = P_3 = (x_3, y_3)$. Point addition uses $\lambda = \frac{y_1+y_2}{x_1+x_2}$ to get $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$ and $y_3 = (x_2 + x_3)\lambda + x_3 + y_2$.

9.2 Reversible point addition

Looking at these formulas, we seem to need at least $6n$ qubits: n for every x and every y. However, in the case of Shor's algorithm we want something different: we have a size $2n$ register containing a superposition of points P_1. Given this P_1, a control qubit q and a fixed P_2, change P_1 into $P_3 = P_1 + P_2$ if $q = 1$ and let it remain P_1 otherwise. Considering division needs at least $3n$ input and output qubits, a minimal implementation of one step would need $2n + 1$ qubits for the input, output and control qubits as well as the ancillary qubits from the division algorithm, and n qubits for the division result. Indeed, modifying Algorithm 1 from Roetteler, Naehrig, Svore and Lauter [20] for the binary case gives us exactly this number of qubits. The modified algorithm for a single step is Algorithm 7 with Table 7 providing a step-by-step breakdown and it is drawn in Circuit 13.

Algorithm 7: Point addition for binary elliptic curves.

Fixed input : A constant field polynomial m of degree $n > 0$. A fixed constant a from the elliptic curve formula. A fixed point $P_2 = (x_2, y_2)$.

Quantum input: A control qubit q. An elliptic curve point P_1 represented as two binary polynomials x_1, y_1 stored in x, y of size n. A size-n array λ initialized to an all-$|0\rangle$ state. Ancillary qubits for division.

Result: (x, y) as $P_1 + P_2 = P_3 = (x_3, y_3)$ if $q = 1$, P_1 if $q = 0$, λ and ancillary qubits same as input $\lambda = 0$.

1	$x \leftarrow \text{const\_ADD}(x, x_2)$	// $x = x_1 + x_2$
2	$y \leftarrow \text{ctrl\_const\_ADD}_q(y, y_2)$	// $y = y_1 + q \cdot y_2$
3	$\lambda \leftarrow \text{DIV}(x, y, \lambda)$	// $\lambda = y/x$
4	$y \leftarrow \text{MODMULT}(x, \lambda, y)$	// $y = y + x \cdot (y/x) = 0$
5	$y \leftarrow \text{SQUARE}(\lambda, y)$	// $y = \lambda^2$
6	$x \leftarrow \text{ctrl\_ADD}_q(x, \lambda)$	// $x = x_1 + x_2 + q \cdot \lambda$
7	$x \leftarrow \text{ctrl\_ADD}_q(x, y)$	// $x = x_1 + x_2 + q(\lambda + \lambda^2)$
8	$x \leftarrow \text{ctrl\_const\_ADD}_q(x, a + x_2)$	// $x = x_1 + x_2 + q(\lambda + \lambda^2 + a + x_2)$
9	$y \leftarrow \text{SQUARE}(\lambda, y)$	// $y = \lambda^2 + \lambda^2 = 0$
10	$y \leftarrow \text{MODMULT}(x, \lambda, y)$	// $y = x \cdot \lambda$
11	$\lambda \leftarrow \text{DIV}(x, y, \lambda)$	// $\lambda = \lambda + (x \cdot \lambda)/x = 0$
12	$x \leftarrow \text{const\_ADD}(x, x_2)$	// $x = x_1 + q(\lambda + \lambda^2 + a + x_2)$
13	$y \leftarrow \text{ctrl\_ADD}_q(y, x)$	// $y = y + q \cdot x_3$
14	$y \leftarrow \text{ctrl\_const\_ADD}_q(y, y_2)$	// $y = y + q \cdot y_2$

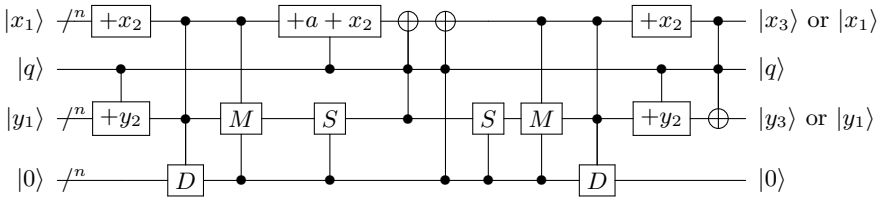
- const_ADD adds x_2 to x. Since this is a constant addition. we use up to n NOT gates with an average of $n/2$, assuming a uniformly random x_2.
- Similarly ctrl_const_ADD applies a CNOT gate from q onto another qubit in x or y at each space the constant polynomial has one as coefficient. Again up to n CNOT gates with an average of $n/2$.

– For DIV we use GCD_DIV as it uses fewer ancillary qubits.
– SQUARE is squaring with separate output, SQUARE(λ, y) computes $y+\lambda^2$. This takes roughly $O(n)$ CNOT gates with the exact number depending on the shape of the polynomial. For certain polynomials this can be more than $O(n)$ if the exponent of the second non-zero term is very high: for example $x^{127}+x^{126}+1$ has a very high number of CNOT gates. A design where we replace the input works equally well but using a LUP-decomposition requires $O(n^2)$ CNOT gates.
– ctrl_ADD applies n TOF gates controlled by q.

Algorithm 7 uses $3n$ TOF gates, up to $3n$ CNOT gates with an average of $1.5n$ and 2 calls to SQUARE, GCD_DIV and MODMULT each. The depth of the algorithm can be reduced by making up to n copies of q and doing the controlled actions simultaneously, but currently the majority of the depth is due to the division algorithm.

step	$q=1$	$q=0$
1	$x = x_1 + x_2$	$x = x_1 + x_2$
2	$y = y_1 + y_2$	$y = y_1$
3	$\lambda = \frac{y_1+y_2}{x_1+x_2}$	$\lambda = \frac{y_1}{x_1+x_2}$
4	$y = 0$	$y = 0$
5	$y = \lambda^2$	$y = \lambda^2$
6-8	$x = x_2 + x_3$	$x = x_1 + x_2$
9	$y = 0$	$y = 0$
10	$y = (x_2 + x_3)\lambda$	$y = y_1$
11	$\lambda = 0$	$\lambda = 0$
12	$x = x_3$	$x = x_1$
13, 14	$y = y_3$	$y = y_1$

Table 7: Steps of Algorithm 7.



Circuit 13: Algorithm 7. M is MODMULT, S is squaring with separate output, D is division.

9.3 Addition of points in special cases

When adding points, the constraints that both points are not the points at infinity or $x_1 \neq x_2$ cannot always be met. Proos and Zalka [18] proposed ignoring these special cases by taking a random ν, taken uniformly random as an integer above 0 and below the order of P, and starting with $[\nu]P$ instead of O. This does not affect the classical computations or quantum Fourier transform. As stated by Proos and Zalka and proven by Rötteler, Naehrig, Svore and Lauter [20], this only affects $n/2^n$ of the state[5]. We leave as future work to find out how to do this in the binary case, and how efficient reversible addition of any two points is.

10 Result

Since the only step with ancillary qubits is division, which has $3n$ input and output qubits, we only need the number of qubits for the divisions as well as 1 control qubit on which we perform the semi-classical Fourier transform. The number of qubits we need is

$$7n + \lfloor \log(n) \rfloor + 9$$

to perform Shor's algorithm on binary elliptic curves. Since we need to do $2n+2$ point additions, each step consisting of 2 divisions, 4 multiplications (including the 2 in division) and 3 controlled additions, we get the following number of Toffoli gates:

$$48n^3 + 8n^{\log(3)+1} + 352n^2 \log(n) + 512n^2 + O(n^{\log(3)}).$$

We do not give an exact number of CNOT gates due to our upper bound of the cost of multiplication, leaving the total count if CNOT gates at $O(n^3)$. In Table 8 several numerical examples are given. We used java to calculate a LUP-decomposition and then calculate the number of gates. The total number of TOF gates is simply the number of TOF gates for a single step multiplied by $(2n+2)$. Depth upper bound is calculated by keeping track of whether 2 or more gates can be executed at the same time, increasing the counter if they cannot. These algorithms are not optimized for depth, as such the depth is of the same order as the number of TOF gates.
We can see that the number of Toffoli gates is strongly dependent on the number of Toffoli gates in the division: $48n^3 + 352n^2 \log(n) + 512n^2$ is purely from the division, with the $\log(n)$ term coming specifically from the incrementer circuit.

10.1 Comparison to previous work

We look at previous work to compare our multiplication algorithm as well as the point addition.

[5] Because each point P still only has one negative $R = -P$ such that $P + R = O$, the fact that they work over prime field rather than binary elliptic curves does not invalidate their proof.

		Single step			Total
n	qubits	TOF gates	CNOT gates	depth upper bound	TOF gates
8	68	7,360	3,514	8,553	132,480
16	125	21,016	11,665	25,183	714,544
127	904	559,141	497,782	776,058	143,140,096
163	1,157	893,585	827,379	1,262,035	293,095,880
233	1,647	1,669,299	1,614,947	2,405,889	781,231,932
283	1,998	2,427,369	2,358,734	3,503,510	1,378,745,592
571	4,015	8,987,401	9,080,190	13,237,682	10,281,586,744

Table 8: Qubit and gate count for Shor's algorithm for binary elliptic curves. Field polynomials from Table 1.

Multiplication in binary finite fields

We compare our multiplication algorithm to two previous instances of multiplication: a variant by Kepley and Steinwandt [13] that optimizes TOF gate count and a variant by Maslov, Mathew, Cheung and Pradhan [15] that optimizes the number of qubits used but does not use Karatsuba. Other variants exist, such as a Karatsuba variant by Parent, Roetteler and Mosca [17], that are worse in terms of space or Toffoli gate count. Since Kepley and Steinwandt use Clifford and T-gates rather than CNOT and Toffoli, we translate 7 of their T-gates and 8 Clifford gates to 1 Toffoli gate, and translate any remaining Clifford gates to CNOT. The resulting comparison is in Table 9. We can see that although Algorithm 4 does not compare favorably in every regard, both the number of Toffoli gates and the number of qubits are best compared to the alternatives. These are considered the more important metrics than CNOT gate count, as Toffoli gates are significantly more expensive than CNOT gates.

Field size 2^n	Toffoli gates			CNOT gates			qubits		
$n =$	Here	[13]	[15]	Here	[13]	[15]	Here	[13]	[15]
4	9	9	16	44	22	3	12	17	12
16	81	81	256	678	376	45	48	113	48
127	2185	2185	16129	20632	13046	126	381	2433	381
256	6561	6561	65536	64706	57008	765	768	7073	768
n	$O(n^{\log_2 3})$	$O(n^{\log_2 3})$	n^2	$O(n^2)$	$O(n^{\log_2 3})$	$O(n)$	$3n$	$O(n^{\log_2 3})$	$3n$

Table 9: Comparison of this work with the works of Kepley and Steinwandt [13] and Maslov, Mathew, Cheung and Pradhan [15] in terms of Toffoli and CNOT gates as well as qubit count.

Greatest common divisor based inversion

The algorithm we used for inversion and division is an improvement over the inversion algorithm based on Kaliski's [20]. That algorithm uses a large number

of controlled Toffoli and CNOT gates, which are translated into 3 Toffoli gates and 1 Toffoli gate respectively. This causes a large increase in Toffoli gate count, with the prime field cases using $32n^2 \log(n)$ Toffoli gates. The binary case would not be much better: a comparison is required in Kaliski's algorithm which, even in the binary case, requires two integer additions. The four remaining integer additions however, would be binary polynomial additions.

In terms of space the difference is smaller. Our implementation of Bernstein and Yang's inversion algorithm [4] is better than the quantum variant of Kaliski's inversion algorithm by $n + \lfloor \log(n) \rfloor + 2$ ancillary qubits. We get this benefit by using part of the input to store decision qubits, saving n qubits, using an incrementer circuit that uses dirty qubits rather than clean ones, saving $\lfloor \log(n) \rfloor$ qubits, as well as using fewer individual ancillary qubits: we only use one, compared to the three required by [20], accounting for differences in binary and prime fields.

Prime field elliptic curve point addition

When comparing the formulas in this thesis to the finite field formulas from Rötteler, Naehrig, Svore and Lauter [20] we cannot compare the number of qubits or gates. As we work in the binary case, this would not be very meaningful and we would likely have better results even with schoolbook algorithms because our essential operations of multiplication and addition require no carry qubits. However, when comparing the algorithms, potential for improvement even in the prime field case shows.

The step function is essentially the same as ours, with extra space required in their algorithm for inversion output, as their algorithm requires separate steps for inversion and multiplication. We can see that a variant of the division algorithm we use likely improves gate count in the prime field case: Bernstein and Yang [4] propose a classical implementation of a constant time prime field division algorithm in addition to the division algorithm we use. Translating this to a quantum algorithm might be worthwhile. For multiplication the algorithm in [20] is not optimized either, with recent work by Gidney [8] giving similar results as the binary Karatsuba algorithm for multiplying integers in $\mathbb{Z}$.

Binary elliptic curve point addition

Comparing to the binary point addition formulas by Amento, Rötteler and Steinwandt [1], we have to note an important difference. Their formulas use projective coordinates to avoid divisions. However, the formulas used still use a lot of ancillary qubits and require separate input and output qubits, leading to $10n$ qubits after the first step. Their paper is unclear on whether they store intermediate results as separate registers, but even if we take this $10n$ as the number of qubits their result has worse space requirements than the $7n + \lfloor \log(n) \rfloor + 9$ we use. However, since they do not use division, they only need 26 multiplications every step (they report 13, but do not replace the input and we need to run their algorithm in reverse to clear the input), which would result in $52n^{\log(3)+1}$ as the leading

term in their Toffoli gate count if they used our new Karatsuba multiplication algorithm.

It is left as future work how much we can benefit from a similar change in coordinate system.

11 Conclusion

The results in table 8 show concrete numbers of logical qubits required to perform Shor's algorithm to solve the discrete logarithm problem on binary elliptic curves. We obtained these results by optimizing the multiplication and division circuits. The number of Toffoli gates is high due to choosing algorithms optimized for space. Using the alternative division algorithm 6 with cryptographic field sizes, the number of Toffoli gates for division could be cut by about 80% at the cost of doubling the number of qubits. Furthermore, optimizing for depth might result in a better depth count than the upper bounds given without changing the number of gates. Additionally, a change to projective coordinates might result in an even better trade-off.

The multiplication algorithm described in Section 7 provides equal or better results to existing binary multiplication algorithms in terms of of Toffoli gate count and space. It can still be optimized for depth or CNOT gate count and can potentially be used as a base for non-binary cases.

11.1 Future work

We suspect that a better algorithm exists for multiplication by $x^{\lceil \frac{n}{2} \rceil} + 1$. Depth so far has been an upper bound: both the multiplication and division algorithm could benefit from a further look at how to optimize it. The division algorithm specifically can also benefit from a better incrementer circuit. A look at projective coordinates might also result in valuable contributions: the algorithms in [1] could benefit from being optimized for space rather than depth. Finally, some of the work done in this thesis such as a quantum algorithm for constant time GCD-based inversion can potentially provide benefits in prime fields.

References

1. Amento, B., Rötteler, M., and Steinwandt, R. Efficient quantum circuits for binary elliptic curve arithmetic: reducing T-gate complexity. *Quantum Information & Computation 13*, 7-8 (2013), 631–644.
2. Avanzi, R., Cohen, H., Doche, C., Frey, G., Lange, T., Nguyen, K., and Vercauteren, F., Eds. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Chapman and Hall/CRC, 2005.
3. Banegas, G., Custódio, R., and Panario, D. A new class of irreducible pentanomials for polynomial-based multipliers in binary fields. *Journal of Cryptographic Engineering* (2018), 1–15.
4. Bernstein, D. J., and Yang, B. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2019*, 3 (2019), 340–398.

5. Cheng, Y. Space-Efficient Karatsuba Multiplication for Multi-Precision Integers. *CoRR abs/1605.06760* (2016).
6. de Wolf, R. Quantum Computing: Lecture Notes. *arXiv preprint quant-ph/1907.09415* (2019).
7. Gidney, C. Constructing Large Increment Gates, 2015. Last retrieved 7 Nov 2019 at `https://algassert.com/circuits/2015/06/12/Constructing-Large-Increment-Gates.html`.
8. Gidney, C. Asymptotically Efficient Quantum Karatsuba Multiplication. *arXiv preprint arXiv:1904.07356* (2019).
9. Griffiths, R. B., and Niu, C.-S. Semiclassical Fourier transform for quantum computation. *Physical Review Letters 76*, 17 (1996), 3228–3231.
10. Hasse, H. Zur Theorie der abstrakten elliptischen Funktionenkörper I, II, III. *Journal für die reine und angewandte Mathematik 175* (1936), 55–62, 69–88, 193–208.
11. Itoh, T., and Tsujii, S. A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases. *Inf. Comput. 78*, 3 (1988), 171–177.
12. Karatsuba, A. A., and Ofman, Y. P. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk* (1962), vol. 145, Russian Academy of Sciences, pp. 293–294.
13. Kepley, S., and Steinwandt, R. Quantum circuits for $\mathbb{F}_{2^n}$-multiplication with subquadratic gate count. *Quantum Information Processing 14*, 7 (2015), 2373–2386.
14. Kerry, C. F., and Gallagher, P. D. FIPS PUB 186-4 Digital Signature Standard (DSS). *National Institute of Standards and Technology* (2013), 92–101.
15. Maslov, D., Mathew, J., Cheung, D., and Pradhan, D. K. An $O(m^2)$-depth quantum algorithm for the elliptic curve discrete logarithm problem over $GF(2^m)^a$. *Quantum Information & Computation 9*, 7 (2009), 610–621.
16. Muñoz-Coreas, E., and Thapliyal, H. Design of Quantum Circuits for Galois Field Squaring and Exponentiation. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2017), IEEE, pp. 68–73.
17. Parent, A., Roetteler, M., and Mosca, M. Improved reversible and quantum circuits for Karatsuba-based integer multiplication. In *12th Conference on the Theory of Quantum Computation, Communication and Cryptography, TQC 2017, June 14-16, 2017, Paris, France* (2017), pp. 7:1–7:15.
18. Proos, J., and Zalka, C. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Information & Computation 3*, 4 (2003), 317–344.
19. Roche, D. S. Space- and Time-Efficient Polynomial Multiplication. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation* (2009), ACM, pp. 295–302.
20. Roetteler, M., Naehrig, M., Svore, K. M., and Lauter, K. E. Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II* (2017), pp. 241–270.
21. Seroussi, G. *Table of low-weight binary irreducible polynomials.* Hewlett-Packard Laboratories, 1998.
22. Shor, P. W. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994* (1994), pp. 124–134.
23. Shor, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput. 26*, 5 (1997), 1484–1509.

24. van Hoof, I. QMKMBP: Quantum modulo Karatsuba multiplier for binary polynomials. Github, 2019. `https://github.com/ikbenbeter/QMKMBP`.
25. van Hoof, I. Space-efficient quantum multiplication of polynomials for binary finite fields with sub-quadratic toffoli gate count. *arXiv preprint arXiv:1910.02849* (2019).