# Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms

Martin Roetteler, Michael Naehrig, Krysta M. Svore, and Kristin Lauter

Microsoft Research, USA

**Abstract.** We give precise quantum resource estimates for Shor's algorithm to compute discrete logarithms on elliptic curves over prime fields. The estimates are derived from a simulation of a Toffoli gate network for controlled elliptic curve point addition, implemented within the framework of the quantum computing software tool suite LIQ$Ui|\rangle$. We determine circuit implementations for reversible modular arithmetic, including modular addition, multiplication and inversion, as well as reversible elliptic curve point addition. We conclude that elliptic curve discrete logarithms on an elliptic curve defined over an $n$-bit prime field can be computed on a quantum computer with at most $9n + 2\lceil \log_2(n) \rceil + 10$ qubits using a quantum circuit of at most $448n^3 \log_2(n) + 4090n^3$ Toffoli gates. We are able to classically simulate the Toffoli networks corresponding to the controlled elliptic curve point addition as the core piece of Shor's algorithm for the NIST standard curves P-192, P-224, P-256, P-384 and P-521. Our approach allows gate-level comparisons to recent resource estimates for Shor's factoring algorithm. The results also confirm estimates given earlier by Proos and Zalka and indicate that, for current parameters at comparable classical security levels, the number of qubits required to tackle elliptic curves is less than for attacking RSA, suggesting that indeed ECC is an easier target than RSA.

**Keywords:** Quantum cryptanalysis, elliptic curve cryptography, elliptic curve discrete logarithm problem.

## 1 Introduction

**Elliptic curve cryptography (ECC).** Elliptic curves are a fundamental building block of today's cryptographic landscape. Thirty years after their introduction to cryptography [27, 24], they are used to instantiate public key mechanisms such as key exchange [10] and digital signatures [15, 20] that are widely deployed in various cryptographic systems. Elliptic curves are used in applications such as transport layer security [9, 4], secure shell [42], the Bitcoin digital currency system [29], in national ID cards [19], the Tor anonymity network [11], and the WhatsApp messaging app [48], just to name a few. Hence, they play a significant role in securing our data and communications.

Different standards (e.g., [7, 45]) and standardization efforts (e.g., [12, 31]) have identified elliptic curves of different sizes targeting different levels of security. Notable curves with widespread use are the NIST curves P-256, P-384, P-521, which are curves in Weierstrass form over special primes of size 256, 384, and 521 bits respectively, the Bitcoin curve `secp256k1` from the SEC2 [7] standard and the Brainpool curves [12]. More recently, Bernstein's Curve25519 [49], a Montgomery curve over a 255-bit prime field, has seen more and more deployment, and it has been recommended to be used in the next version of the TLS protocol [25] along with another even more recent curve proposed by Hamburg called Goldilocks [17].

The security of elliptic curve cryptography relies on the hardness of computing discrete logarithms in elliptic curve groups, i.e. the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP). Elliptic curves have the advantage of relatively small parameter and key sizes when compared to other cryptographic schemes, such as those based on RSA [36] or finite field discrete logarithms [10], when compared at the same security level. For example, according to NIST recommendations from 2016, a 256-bit elliptic curve provides a similar resistance against classical

attackers as an RSA modulus of size 3072 bits[1]. This advantage arises from the fact that the currently known best algorithms to compute elliptic curve discrete logarithms are exponential in the size of the input parameters[2], whereas there exist subexponential algorithms for factoring [26, 8] and finite field discrete logarithms [16, 21].

**The quantum computer threat.** In his famous paper [39], Peter Shor presented two polynomial-time quantum algorithms, one for integer factorization and another one for computing discrete logarithms in a finite field of prime order. Shor notes that the latter algorithm can be generalized to other fields. It also generalizes to the case of elliptic curves. Hence, given the prerequisite that a large enough general purpose quantum computer can be built, the algorithms in Shor's paper completely break all current crypto systems based on the difficulty of factoring or computing discrete logarithms. Scaling up the parameters for such schemes to sizes for which Shor's algorithm becomes practically infeasible will most likely lead to highly impractical instantiations.

Recent years have witnessed significant advances in the state of quantum computing hardware. Companies have invested in the development of qubits, and the field has seen an emergence of startups, with some focusing on quantum hardware, others on software for controlling quantum computers, and still others offering consulting services to ready for the quantum future. The predominant approach to quantum hardware is focused around a digital, programmable, and universal quantum computer. With the amount of investment in quantum computing hardware, the pace of scaling is increasing and underscoring the need to understand the scaling of the difficulty of ECDLP.

**Language-Integrated Quantum Operations: LIQ$Ui|\rangle$.** As quantum hardware advances towards larger-scale systems of upwards of tens to hundreds of qubits, there is a critical need for a software architecture to program and control the device. We use the LIQ$Ui|\rangle$ software architecture [47] to determine the resource costs of solving the ECDLP. LIQ$Ui|\rangle$ is a high-level programming language for quantum algorithms embedded in F#, a compilation stack to translate and compile quantum algorithms into quantum circuits, and a simulator to test and run quantum circuits[3]. LIQ$Ui|\rangle$ can simulate roughly 32 qubits in 32GB RAM, however, we make use of the fact that reversible circuits can be simulated efficiently on classical input states for thousands of qubits.

**Gate sets and Toffoli gate networks.** The basic underlying fault-tolerant architecture and coding scheme of a quantum computer determine the universal gate set, and hence by extension also the synthesis problems that have to be solved in order to compile high-level, large-scale algorithms into a sequence of operations that an actual physical quantum computer can then execute. A gate set that arises frequently and that has been studied often in the literature, but by no means the only conceivable gate set, is the so-called Clifford+$T$ gate set [30]. This gate set consists of the Hadamard gate $H = \frac{1}{\sqrt{2}} \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$, the phase gate $P = \mathrm{diag}(1, i)$, and the controlled NOT (CNOT) gate which maps $(x, y) \mapsto (x, x \oplus y)$ as generators of the Clifford group, along with the $T$ gate given by $T = \mathrm{diag}(1, \exp(\pi i/4))$. The Clifford+$T$ gate set is known to be universal [30]. This means that it can be used to approximate any given target unitary single qubit operation to within precision $\varepsilon$ using sequences of length $4 \log_2(1/\varepsilon)$ [38, 23], and using an entangling gate such as the CNOT gate, the Clifford+$T$ gate set can approximate any unitary operation. When assessing the complexity of a quantum circuit built from Clifford+$T$ gates, often only $T$-gates are counted as many fault-tolerant implementations of the Clifford+$T$ gate set at the logical gate level require much more resources for $T$-gates than for Clifford gates [13].

[1] Opinions about such statements of equivalent security levels differ, for an overview see `https://www.keylength.com`. There is consensus about the fact that elliptic curve parameters can be an order of magnitude smaller than parameters for RSA or finite field discrete logarithm systems to provide similar security.

[2] For a recent survey, see [14].

[3] See `http://stationq.github.io/Liquid/` and `https://github.com/StationQ/Liquid`.

In this paper, we base reversible computations entirely on the Toffoli gate. The Toffoli gate $|x, y, z\rangle \mapsto |x, y, z \oplus xy\rangle$ is known to be universal for reversible computing [30] and can be implemented exactly over the Clifford+$T$ gate set, see [37] for a $T$-depth 1 implementation using a total of 7 qubits and [1] for a $T$-depth 3 realization using a total of 3 qubits. As discussed in [18, Section V], there are two main reasons for focusing on Toffoli gate networks as our preferred realization of quantum circuits. The first is that because the Toffoli gate can be implemented exactly over the Clifford+$T$ gate set, Toffoli networks do not have gate synthesis overhead. The second is testability and debugging. Toffoli gate networks can be simulated using classical reversible simulators. While a fully functional simulation of a quantum circuit could be deemed feasible for circuits on up to 50 qubits, classical simulation of Toffoli gate-based circuits can deal with a lot more qubits. Also, for implementations on actual quantum hardware, Toffoli gate circuits can be debugged efficiently, where faults can be localized through binary search.

**Estimating quantum resources for Shor's ECDLP algorithm.** Understanding the concrete requirements for a quantum computer that is able to run Shor's algorithm helps to put experimental progress in quantum computing into perspective. Although it is clear that the polynomial runtime asymptotically breaks ECC, constant factors can make an important difference when actually implementing the algorithm.

In [34], Proos and Zalka describe how Shor's algorithm can be implemented for the case of elliptic curve groups. They conclude with a table of resource estimates for the number of logical qubits and time (measured in "1-qubit additions") depending on the bitsize of the elliptic curve. Furthermore, they compare these estimates to those for Shor's factoring algorithm and argue that computing elliptic curve discrete logarithms is significantly easier than factoring RSA moduli at comparable classical security levels. However, some questions remained unanswered by [34], the most poignant of which being whether it is actually possible to construct and simulate the circuits to perform elliptic curve point addition in order to get confidence in their correctness. Another question that remained open is whether it is possible to determine constants that were left in terms of asymptotic scaling and whether some of the proposed circuit constructions to compress registers and to synchronize computations can actually be implemented in code that can then be automatically generated for arbitrary input curves.

Here we build on their work and fully program and simulate the underlying arithmetic. We verify the correctness of our algorithms and obtain concrete resource costs measured by the overall number of logical qubits, the number of Toffoli gates and the depth of a quantum circuit for implementing Shor's algorithm.

**Contributions.** In this paper, we present precise resource estimates for quantum circuits that implement Shor's algorithm to solve the ECDLP. In particular, our contributions are as follows:

- We describe reversible algorithms for modular quantum arithmetic. This includes modular addition, subtraction, negation and doubling of integers held in quantum registers, modular multiplication, squaring and modular inversion.
- For modular multiplication, we consider two different approaches, besides an algorithm based on modular doublings and modular additions, we also give a circuit for Montgomery multiplication.
- Based on our implementations it transpired that using Montgomery arithmetic is beneficial as the cost for the multiplication can be seen to be lower than that of the double-and-add method. The latter requires less ancillas, however, in the given algorithm there are always enough ancillas available as overall a relatively large number of ancillas must be provided.
- Our modular inversion algorithm is a reversible implementation of the Montgomery inverse via the binary extended Euclidean (binary GCD) algorithm. To realize this algorithm as a circuit, we introduce tools that might be of independent interest for other reversible algorithms.
- We describe a quantum circuit for elliptic curve point addition in affine coordinates and describe how it can be used to implement scalar multiplication to be used in Shor's algorithm.

– We have implemented all of the above algorithms in F# within the framework of the quantum computing software tool suite LIQ$Ui|\rangle$ [47] and have simulated and tested all of these algorithms for real-world parameters of up to 521 bits[4].
– Derived from our implementation, we present concrete resource estimates for the total number of qubits, the number of Toffoli gates and the depth of the Toffoli gate networks to realize Shor's algorithm and its subroutines. We compare the quantum resources for solving the ECDLP to those required in Shor's factoring algorithm that were obtained in the recent work [18].

**Results.** Our implementation realizes a reversible circuit for controlled elliptic curve point addition on an elliptic curve defined over a field of prime order with $n$ bits and needs at most $9n + 2\lceil \log_2(n) \rceil + 10$ qubits. An interpolation of the data points for the number of Toffoli gates shows that the quantum circuit can be implemented with at most roughly $224n^2 \log_2(n) + 2045n^2$ Toffoli gates. For Shor's full algorithm, the point addition needs to be run $2n$ times sequentially and does not need additional qubits. The overall number of Toffoli gates is thus about $448n^3 \log_2(n) + 4090n^3$. For example, our simulation of the point addition quantum circuit for the NIST standardized curve P-256 needs 2330 logical qubits and the full Shor algorithm would need about $1.26 \cdot 10^{11}$ Toffoli gates. In comparison, Shor's factoring algorithm for a 3072-bit modulus needs 6146 qubits and $1.5 \cdot 10^{14}$ Toffoli gates[5], which confirms results by Proos and Zalka showing that it is easier to break ECC than RSA at comparable classical security.

Our estimates provide a data point that allows a better understanding of the requirements to run Shor's quantum ECDLP algorithm and we hope that they will serve as a basis to make better predictions about the time horizon until which elliptic curve cryptography can still be considered secure. Besides helping to gain a better understanding of the post-quantum (in-) security of elliptic curve cryptosystems, we hope that our reversible algorithms (and their LIQ$Ui|\rangle$ implementations) for modular arithmetic and the elliptic curve group law are of independent interest to some, and might serve as building blocks for other quantum algorithms.

## 2 Elliptic curves and Shor's algorithm

This section provides some background on elliptic curves over finite fields, the elliptic curve discrete logarithm problem (ECDLP) and Shor's quantum algorithm to solve the ECDLP. Throughout, we restrict to the case of curves defined over prime fields of large characteristic.

### 2.1 Elliptic curves and the ECDLP

Let $p > 3$ be a prime. Denote by $\mathbb{F}_p$ the finite field with $p$ elements. An elliptic curve over $\mathbb{F}_p$ is a projective, non-singular curve of genus 1 with a specified base point. It can be given by an affine Weierstrass model, i.e. it can be viewed as the set of all solutions $(x, y)$ to the equation $E : y^2 = x^3 + ax + b$ with two curve constants $a, b \in \mathbb{F}_p$, together with a point at infinity $\mathcal{O}$. The set of $\mathbb{F}_p$-rational points consists of $\mathcal{O}$ and all solutions $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ and is denoted by $E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p \mid y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$. The set $E(\mathbb{F}_p)$ is an abelian group with respect to a group operation "+" that is defined via rational functions in the point coordinates with $\mathcal{O}$ as the neutral element. Similarly, for a field extension $\mathbb{F} \supseteq \mathbb{F}_p$, one similarly defines the group of $\mathbb{F}$-rational points $E(\mathbb{F})$ and if $\mathbb{F}$ is an algebraic closure of $\mathbb{F}_p$, we simply denote $E = E(\mathbb{F})$. For an extensive treatment of elliptic curves, we refer the reader to [41].

The elliptic curve group law on an affine Weierstrass curve can be computed as follows. Let $P_1, P_2 \in E$ and let $P_3 = P_1 + P_2$. If $P_1 = \mathcal{O}$ then $P_3 = P_2$ and if $P_2 = \mathcal{O}$, then $P_3 = P_1$. Now let $P_1 \neq \mathcal{O} \neq P_2$ and write $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ for $x_1, y_1, x_2, y_2 \in \mathbb{F}$. If $P_2 = -P_1$, then $x_1 = x_2$, $y_2 = -y_1$ and $P_3 = \mathcal{O}$. If neither of the previous cases occurs, then $P_3 = (x_3, y_3)$ is an affine point and can be computed as

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = (x_1 - x_3)\lambda - y_1,$$

[4] Our code will be made publicly available.
[5] These estimates are interpolated from the results in [18].

where $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ if $P_1 \neq P_2$, i.e. $x_1 \neq x_2$, and $\lambda = \frac{3x_1^2 + a}{2y_1}$ if $P_1 = P_2$. For a positive integer $m$, denote by $[m]P$ the $m$-fold sum of $P$, i.e. $[m]P = P + \cdots + P$, where $P$ occurs $m$ times. Extended to all $m \in \mathbb{Z}$ by $[0]P = \mathcal{O}$ and $[-m]P = [m](-P)$, the map $[m] : E \to E, P \mapsto [m]P$ is called the multiplication-by-$m$ map or simply scalar multiplication by $m$. Scalar multiplication (or group exponentiation in the multiplicative setting) is one of the main ingredients for discrete-logarithm-based cryptographic protocols. It is also an essential operation in Shor's ECDLP algorithm. The order $\mathrm{ord}(P)$ of a point $P$ is the smallest positive integer $r$ such that $[r]P = \mathcal{O}$.

Curves that are most widely used in cryptography are defined over large prime fields. One works in a cyclic subgroup of $E(\mathbb{F}_p)$ of large prime order $r$, where $\#E(\mathbb{F}_p) = h \cdot r$. The group order can be written as $\#E(\mathbb{F}_p) = p + 1 - t$, where $t$ is called the trace of Frobenius and the Hasse bound ensures that $|t| \leq 2\sqrt{p}$. Thus $\#E(\mathbb{F}_p)$ and $p$ are of roughly the same size. The most efficient instantiations of ECC are achieved for small cofactors $h$. For example, the above mentioned NIST curves have prime order, i.e. $h = 1$, and Curve25519 has cofactor $h = 8$. Let $P \in E(\mathbb{F}_p)$ be an $\mathbb{F}_p$-rational point on $E$ of order $r$ and let $Q \in \langle P \rangle$ be an element of the cyclic subgroup generated by $P$. The Elliptic Curve Discrete Logarithm Problem (ECDLP) is the problem to find the integer $m \in \mathbb{Z}/r\mathbb{Z}$ such that

$$Q = [m]P.$$

The bit security of an elliptic curve is estimated by extrapolating the runtime of the most efficient algorithms for the ECDLP.

The currently best known classical algorithms to solve the ECDLP are based on parallelized versions of Pollard's rho algorithm [32, 46, 33]. When working in a group of order $n$, the expected running time for solving a single ECDLP is $(\sqrt{\pi/2} + o(1))\sqrt{n}$ group operations based on the birthday paradox. This is exponential in the input size $\log(n)$. See [14] for further details and [5] for a concrete, implementation-based security assessment.

### 2.2 Shor's quantum algorithm for solving the ECDLP

In [39], Shor presented two polynomial time quantum algorithms, one for factoring integers, the other for computing discrete logarithms in finite fields. The second one can naturally be applied for computing discrete logarithms in the group of points on an elliptic curve defined over a finite field.

We are given an instance of the ECDLP as described above. Let $P \in E(\mathbb{F}_p)$ be a fixed generator of a cyclic subgroup of $E(\mathbb{F}_p)$ of known order $\mathrm{ord}(P) = r$, let $Q \in \langle P \rangle$ be a fixed element in the subgroup generated by $P$; our goal is to find the unique integer $m \in \{1, \ldots, r\}$ such that $Q = [m]P$. Shor's algorithm proceeds as follows. First, two registers of length $n + 1$ qubits[6] are created and each qubit is initialized in the $|0\rangle$ state. Then a Hadamard transform $H$ is applied to each qubit, resulting in the state $\frac{1}{2^{n+1}} \sum_{k,\ell=0}^{2^{n+1}-1} |k, \ell\rangle$. Next, conditioned on the content of the register holding the label $k$ or $\ell$, we add the corresponding multiple of $P$ and $Q$, respectively, i. e., we implement the map

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

Hereafter, the third register is discarded and a quantum Fourier transform $\mathrm{QFT}_{2^{2\cdot(n+1)}}$ on $2(n+1)$ qubits is computed. Finally, the state of the first two registers—which hold a total of $2(n + 1)$ qubits—is measured. As shown in [40], the discrete logarithm $m$ can be computed from this measurement data via classical post-processing. The corresponding quantum circuit is shown in Figure 1.

Using Kitaev's phase estimation framework [30], Beauregard [2] obtained a quantum algorithm for factoring an integer $N$ from a circuit that performs a conditional multiplication of the form $x \mapsto ax \mod N$, where $a \in \mathbb{Z}_N$ is a random constant integer modulo $N$. The circuit uses only $2n + 3$ qubits, where $n$ is the bitlength of the integer to be factored. An implementation of this

[6] Hasse's bound guarantees that the order of $P$ can be represented with $n + 1$ bits.

algorithm on $2n+2$ qubits, using Toffoli-gate-based modular multiplication is described in [18]. In analogy to this algorithm, one can modify Shor's ECDLP algorithm, resulting in the circuit shown in Figure 2. The phase shift matrices $R_i = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta_k} \end{pmatrix}$, $\theta_k = -\pi \sum_{j=0}^{k-1} 2^{k-j} \mu_j$, depend on all previous measurement outcomes $\mu_j \in \{0,1\}$, $j \in \{0, \ldots, k-1\}$.

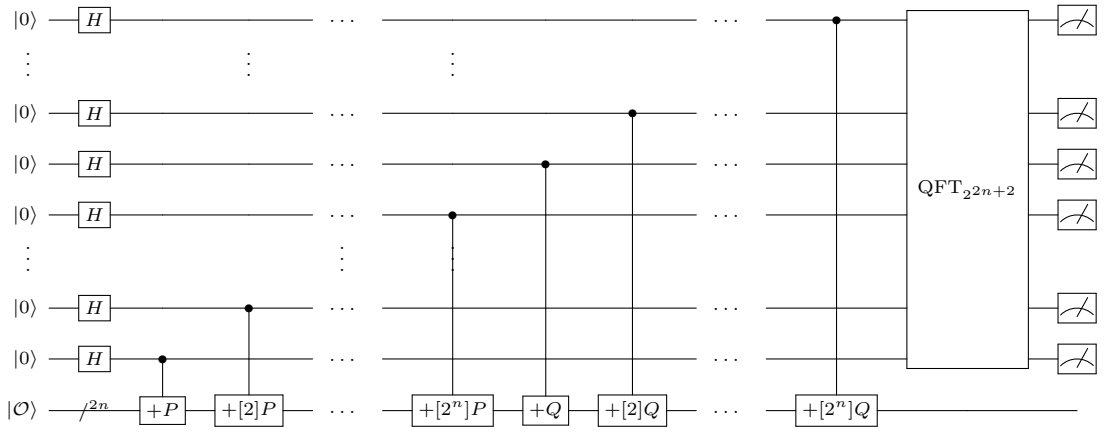


Fig. 1: Shor's algorithm to compute the discrete logarithm in the subgroup of an elliptic curve generated by a point $P$. The input to the problem is a point $Q$, and the task is to find $m \in \{1, \ldots, \mathrm{ord}(P)\}$ such that $Q = [m]P$. The circuit naturally decomposes into three parts, namely (i) the Hadamard layer on the left, (ii) a double scalar multiplication (in this figure implemented as a cascade of conditional point additions), and (iii) the quantum Fourier transform QFT and subsequent measurement in the standard basis which is performed at the end.

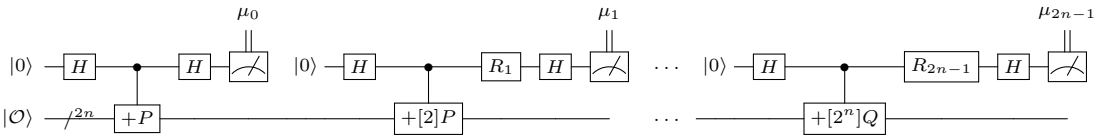


Fig. 2: Shor's algorithm to compute the discrete logarithm in the subgroup of an elliptic curve generated by a point $P$, analogous to the algorithms from [2] and [18]. The gates $R_k$ are phase shift gates given by $\mathrm{diag}(1, e^{i\theta_k})$, where $\theta_k = -\pi \sum_{j=0}^{k-1} 2^{k-j} \mu_j$ and the sum runs over all previous measurements $j$ with outcome $\mu_j \in \{0,1\}$. In contrast to the circuit in Figure 1 only one additional qubit is needed besides qubits required to represent and add the elliptic curve points.

## 3 Reversible modular arithmetic

Shor's algorithm for factoring actually only requires modular multiplication of a quantum integer with classically known constants. In contrast, the elliptic curve discrete logarithm algorithm requires elliptic curve scalar multiplications to compute $[k]P + [\ell]Q$ for a superposition of values for the scalars $k$ and $\ell$. These scalar multiplications are comprised of elliptic curve point additions, which in turn consist of a sequence of modular operations on the coordinates of the elliptic curve points. This requires the implementation of full modular arithmetic, which means that one needs to add and multiply two integers held in quantum registers modulo the constant integer modulus $p$.

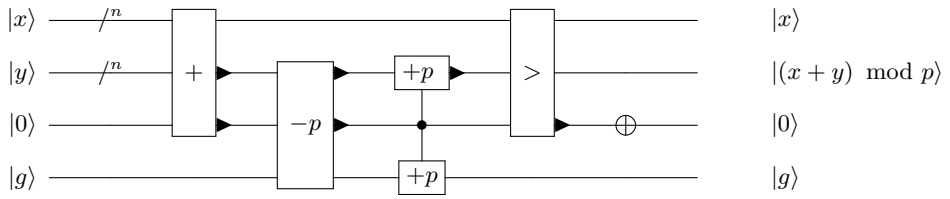


Fig. 3: `add_modp`: Quantum circuit for in-place modular addition $|x\rangle\,|y\rangle \mapsto |x\rangle\,|(x+y) \mod p\rangle$. The registers $|x\rangle$, $|y\rangle$ consist of $n$ logical qubits each. The ancilla qubit $|c\rangle$ is a single logical qubit. The circuit uses integer addition $+$, addition $+p$ and subtraction $-p$ of the constant modulus $p$, and strict comparison of two $n$-bit integers in the registers $|x\rangle$ and $|y\rangle$, where the output bit flips the carry qubit in the last register. The constant adders use an ancilla qubit in an unknown state $|g\rangle$, which is returned to the same state at the end of the circuit. To implement controlled modular addition `ctrl_add_modp`, one simply controls all operations in this circuit.

This section presents quantum circuits for reversible modular arithmetic on $n$-bit integers that are held in quantum registers. We provide circuit diagrams for the modular operations, in which black triangles on the right side of gate symbols indicate qubit registers that are modified and hold the result of the computation. Essential tools for implementing modular arithmetic are integer addition and bit shift operations on integers, which we describe first.

### 3.1 Integer addition and binary shifts

The algorithms for elliptic curve point addition as described below need integer addition and subtraction in different variants: standard integer addition and subtraction of two $n$-bit integers, addition and subtraction of a classical constant integer, as well as controlled versions of those.

For adding two integers, we take the quantum circuit described by Takahashi et al. [44]. The circuit works on two registers holding the input integers, the first of size $n$ qubits and the second of size $n+1$ qubits. It operates in place, i.e. the contents of the second register are replaced to hold the sum of the inputs storing a possible carry bit in the additionally available qubit. To obtain a subtraction circuit, we implement an inverse version of this circuit. The carry bit in this case indicates whether the result of the subtraction is negative. Controlled versions of these circuits can be obtained by using partial reflection symmetry to save controls, which compares favorably to a generic version where simply all gates are controlled. For the constant addition circuits, we take the algorithms described in [18]. Binary doubling and halving circuits are needed for the Montgomery multiplication and inversion algorithms. They are implemented essentially as cyclic bit shifts realized by sequences of symmetric bit swap operations built from CNOT gates.

### 3.2 Modular addition and doubling

We now turn to modular arithmetic. The circuit shown in Figure 3 computes a modular addition of two integers $x$ and $y$ held in $n$-qubit quantum registers $|x\rangle$ and $|y\rangle$, modulo the constant integer modulus $p$. It performs the operation in place $|x\rangle\,|y\rangle \mapsto |x\rangle\,|(x+y) \mod p\rangle$ and replaces the second input with the result. It uses quantum circuits for plain integer addition and constant addition and subtraction of the modulus. It uses two auxiliary qubits, one of which is used as an ancilla qubit in the constant addition and subtraction and can be in an unknown state to which it will be returned at the end of the circuit. The other qubit stores the bit that determines whether a modular reduction in form of a modulus subtraction actually needs to be performed or not. It is uncomputed at the end by a strict comparison circuit between the result and the first input. Modular subtraction is implemented by reversing the circuit.

The modular doubling circuit for a constant odd integer modulus $p$ in Figure 4 follows the same principle. There are two changes that make it more efficient than the addition circuit. First of all it works in place on only one $n$-qubit input integer $|x\rangle$, it computes $|x\rangle \mapsto |2x \mod p\rangle$. Therefore it uses only $n+2$ qubits. The first integer addition in the modular adder circuit is replaced by a

more efficient multiplication by 2 implemented via a cyclic bit shift as described in the previous subsection. Since we assume that the modulus $p$ is odd in this circuit, the auxiliary reduction qubit can be uncomputed by checking whether the least significant bit of the result is 0 or 1. A subtraction of the modulus has taken place if, and only if this bit is 1.

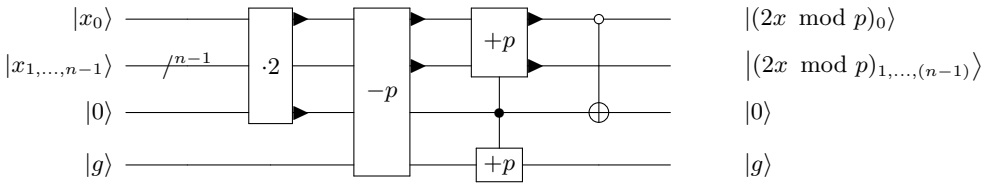


Fig. 4: `dbl_modp`: Quantum circuit for in-place modular doubling $|x\rangle \mapsto |2x \mod p\rangle$ for an odd constant modulus $p$. The registers $|x\rangle$ consists of $n$ logical qubits, the circuit diagram represents the least significant bit separately. The circuit uses a binary doubling operation $\cdot 2$ and addition $+p$ and subtraction $-p$ of the constant modulus $p$. The constant adders use an ancilla qubit in an unknown state $|g\rangle$, which is returned to the same state at the end of the circuit.

For adding a classical constant to a quantum register modulo a classical constant modulus, we use the in-place modular addition circuit described in [18, Section II]. The circuit operates on the $n$-bit input and requires only 1 ancilla qubit initialized in the state $|0\rangle$ and $n-1$ dirty ancillas that are given in an unknown state and will be returned in the same state at the end of the computation.

### 3.3 Modular multiplication

**Multiplication by modular doubling and addition.** Modular multiplication can be computed by repeated modular doublings and conditional modular additions. Figure 5 shows a circuit that computes the product $z = x \cdot y \mod p$ for constant modulus $p$ as described by Proos and Zalka [34, Section 4.3.2] by using a simple expansion of the product along a binary decomposition of the first multiplicand, i.e.

$$x \cdot y = \sum_{i=0}^{n-1} x_i 2^i \cdot y = x_0 y + 2(x_1 y + 2(x_2 y + \cdots + 2(x_{n-2} y + 2(x_{n-1} y)) \ldots)).$$

The circuit runs on $3n+2$ qubits, $2n$ of which are used to store the inputs, $n$ to accumulate the result and 2 ancilla qubits are needed for the modular addition and doubling operations, one of which can be dirty. The latter could be taken to be one of the $x_i$, for example $x_0$ except in the last step, when the modular addition gate is conditioned on $x_0$. For simplicity, we assume it to be a separate qubit.

Figure 6 shows the corresponding specialization to compute a square $z = x^2 \mod p$. It uses $2n+3$ qubits by removing the $n$ qubits for the second input multiplicand, and adding one ancilla qubit, which is used in round $i$ to copy out the current bit $x_i$ of the input in order to add $x$ to the accumulator conditioned on the value of $x_i$.

**Montgomery multiplication** In classical applications, Montgomery multiplication [28] is often the most efficient choice for modular multiplication if the modulus does not have a special shape such as being close to a power of 2. Here, we explore it as an alternative to the algorithm using modular doubling and addition as described above.

In [28], Montgomery introduced a representation for an integer modulo $p$ he called a $p$-residue that is now called the Montgomery representation. Let $R$ be an integer radix coprime to $p$. An integer $a$ modulo $p$ is represented by the Montgomery representation $aR \mod p$. The Montgomery reduction algorithm takes as input an integer $0 \leq c < Rp$ and computes $cR^{-1} \mod p$. Thus given

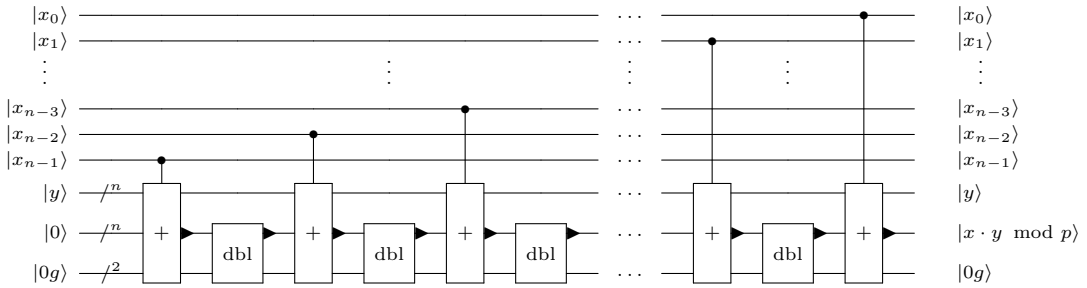


Fig. 5: `mul_modp`: Quantum circuit for modular multiplication $|x\rangle|y\rangle|0\rangle \mapsto |x\rangle|y\rangle|x \cdot y \mod p\rangle$ built from modular doublings dbl ← `dbl_modp` and controlled modular additions + ← `ctrl_add_modp`. The registers $|x_i\rangle$ hold single logical qubits, $|y\rangle$ and $|0\rangle$ hold $n$ logical qubits. The two ancilla qubits $|0g\rangle$ are the ones needed in the modular addition and doubling circuits, the second one can be in an unknown state to which it will be returned.

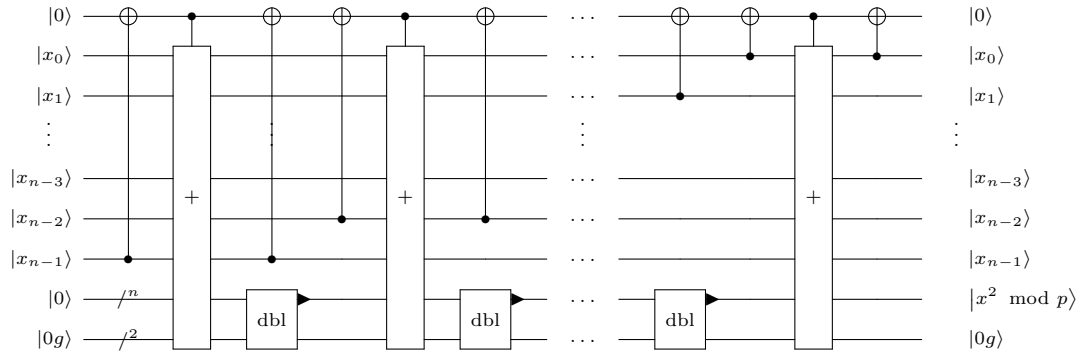


Fig. 6: `squ_modp`: Quantum circuit for modular squaring $|x\rangle|0\rangle \mapsto |x\rangle|x^2 \mod p\rangle$ built from modular doublings dbl ← `dbl_modp` and controlled modular additions + ← `ctrl_add_modp`. The registers $|x_i\rangle$ hold single logical qubits, $|0\rangle$ holds $n$ logical qubits. The two ancilla qubits $|0g\rangle$ are the ones needed in the modular addition and doubling circuits, the second one can be in an unknown state to which it will be returned.

two integers $aR \mod p$ and $bR \mod p$ in Montgomery representation, applying the Montgomery reduction to their product yields the Montgomery representation $(ab)R \mod p$ of the product. If $R$ is a power of 2, one can interleave the Montgomery reduction with school-book multiplication, obtaining a combined Montgomery multiplication algorithm. The division operations usually needed for computing remainders are replaced by binary shifts in each round of the multiplication algorithm.

The multiplication circuit using modular doubling and addition operations described in the previous subsection contains two modular reductions in each round of the algorithm. Each of those is realized here by at least two integer additions. In contrast, the Montgomery algorithm shown in Figure 7 avoids these and uses only one integer addition per round. This reduces the circuit depth in comparison to the double-and-add approach. However, it comes at the cost of requiring more qubits. The main issue is that the algorithm stores the information for each round, whether the odd modulus $p$ had to be added to the intermediate result to make it even or not. This is done to allow divisions by 2 through a simple bit shift of an even number. These bits are still set at the end of the circuit shown in Figure 7. To uncompute these values, we copy the result to another $n$-qubit register, and run the algorithm backwards, which essentially doubles the depth of the algorithm. But this still leads to a lower overall depth than the one of the double-and-add algorithm. Hence, switching to Montgomery multiplication presents a trade-off between the required number of qubits and the multiplication circuit depth.

The same optimization as shown in the previous section allows to save $n - 1$ qubits when implementing a Montgomery squaring circuit that computes $z = x^2 \mod p$.

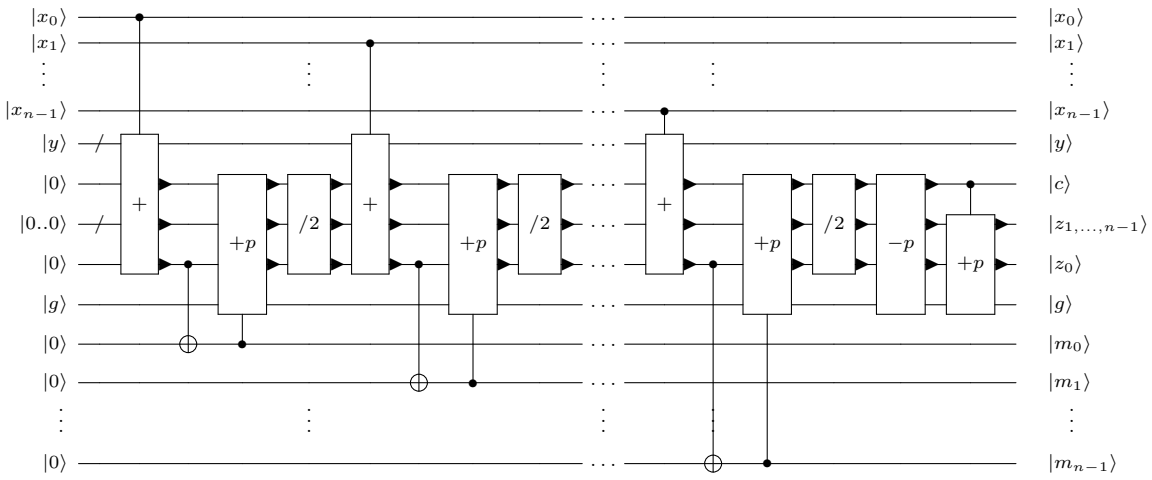


Fig. 7: `mul_modp`: Quantum circuit for the forward Montgomery modular multiplication $|x\rangle\,|y\rangle\,|0\rangle \mapsto |x\rangle\,|y\rangle\,|z = x \cdot y \mod p\rangle$. The register $|y\rangle$ holds $n$ logical qubits and $|0..0\rangle$ holds $n-1$. All others are single qubits. The qubit $|g\rangle$ is a dirty ancilla qubit in an unknown state. The qubit labeled $|m_i\rangle$ holds the information whether the intermediate result in round $i$ was odd and thus whether $p$ was added to it. The circuit uses integer addition $+$, integer addition $+p$ and subtraction $-p$ of the constant modulus $p$ and a halving circuit $/2$ that performs a cyclic qubit shift. The last two gates reflect the standard conditional subtraction of $p$. To uncompute the qubits $|m_i\rangle$, one copies out the result $z$ and runs the circuit backwards.

### 3.4 Modular inversion

Performing the modular inversion on a quantum computer is by far the most costly operation required in order to implement the affine group law of an elliptic curve. We use a reversible circuit for the extended binary greatest common divisor algorithm [43] that implements Kaliski's algorithm [22] for inverting a number $xR \mod p$ given in Montgomery representation for $R = 2^n$; i.e. an algorithm (i) which only uses elementary reversible operations such as Toffoli gates, (ii) whose sequence of instructions does not depend on the given input $x2^n \mod p$, and (iii) whose output is again in Montgomery form $x^{-1}2^n \mod p$.

We use the extended binary GCD algorithm to compute the representation of the greatest common divisor between $x$ and $p$, with the added requirement to ensure property (ii), namely to make sure that the sequence of operations that carries out the Euclidean algorithm is the same, independent of $x$. In particular, an issue is that for different inputs $x \neq x'$ the usual, irreversible Euclidean algorithm can terminate in a different number of steps. To fix this, we include a counter register which is incremented upon termination of the loop to ensure the algorithm is run for $2n$ rounds, which is the worst-case runtime.

In the following algorithm to compute the Montgomery inverse the inputs are a prime $p$ and a value $x$ where $0 \leq x < p$. The output is $x^{-1}2^n \bmod p$. In functional programming style (here, using F# syntax), Kaliski's algorithm is described as follows:

```
let MGinverse p x =
  let rec xmg u v r s k =
    match u, v, r, s with
      | _,0,r,_                -> r
      | u,_,_,_ when u\%2=0 -> xmg (u >>> 1) v r (s <<< 1) (k+1)
      | _,v,_,_ when v\%2=0 -> xmg u (v >>> 1) (r <<< 1) s (k+1)
      | u,v,_,d when u > v  -> xmg ((u-v) >>> 1) v (r+s) (s <<< 1) (k+1)
      | _,_,_,_                -> xmg u ((v-u) >>> 1) (r <<< 1) (r+s) (k+1)
  xmg p x 0 1 0
```

The algorithm actually computes only the so-called "almost inverse" which is of the form $x^{-1}2^k$, i.e., there is a secondary step necessary to convert to the correct form (not shown here). Two example executions are shown in Figure 8.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $u$ | 11 | 11 | 11 | 11 | 5 | 2 | 1 | 1 | 1 |
| $v$ | 8 | 4 | 2 | 1 | 1 | 1 | 1 | 0 | 0 |
| $r$ | 0 | 0 | 0 | 0 | 1 | 3 | 3 | 6 | 6 |
| $s$ | 1 | 1 | 1 | 1 | 2 | 4 | 8 | 11 | 11 |
| $k$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 7 |
| $\ell$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

(a)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $u$ | 11 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $v$ | 7 | 7 | 7 | 3 | 1 | 0 | 0 | 0 | 0 |
| $r$ | 0 | 1 | 1 | 2 | 4 | 8 | 8 | 8 | 8 |
| $s$ | 1 | 2 | 4 | 5 | 7 | 11 | 11 | 11 | 11 |
| $k$ | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 |
| $\ell$ | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 |

(b)

Fig. 8: Two example runs of the reversible extended binary Euclidean algorithm to compute the Montgomery inverse modulo $p = 11$. Shown in (a) is the execution for input $x = 8$ which leads to termination of the usual irreversible algorithm after $k = 7$ steps. The algorithm is always executed for $2n$ rounds, where $n$ is the bit-size of $p$ which is an upper bound on the maximum number of steps required for general input $x$. Once the final step $v = 0$ has been reached, a counter register $\ell$ is incremented. Shown in (b) is the execution for input $x = 7$ which leads to termination after 5 steps after which the counter is incremented three times.

As shown in Fig. 8, the actual number of steps that need to be executed until the gcd is obtained, depends on the actual input $x$: in the first example the usual Kaliski algorithm terminates after $k = 7$ steps, whereas in the second example the usual algorithm would terminate after $k = 5$ steps. To make the algorithm reversible, we must find an implementation that carries out the same operations, irrespective of the input. The two main ingredients to obtain such an implementation are a) an upper bound of $2n$ steps that Kaliski's algorithm can take in the worst case [22] and b) the introduction of a counter that ensures that either the computation is propagated forward or, in case the usual Kaliski algorithm has terminated, the counter is incremented. Such a counter can be implemented using $O(\log(n))$ qubits.

The circuit shown in Fig. 9 implements the Kaliski algorithm in a reversible fashion. We next describe the various registers used in this circuit and explain why this algorithm actually computes the same output as the Kaliski algorithm. The algorithm uses $n$-bit registers for inputs $u$ and $v$, where $u$ is initially set to the underlying prime $p$. As $p$ is constant, the register can be prepared using bit flips corresponding to the binary representation of $p$. Moreover, $v$ is initially set to the input $x$ of which we would like to compute the inverse. Moving downward from the top, the next line represents a single ancilla qubit which is used to store an intermediate value which is the result of a comparison. Next, is an $n+1$-bit register for $r$ and likewise an $n+1$-bit register for $s$, so that the loop invariant $p = ru + sv$ holds at each stage of the algorithm. Eventually, when $v = 0$ is reached, register $r$ will hold the almost inverse and register $s$ will be equal to $p$. The next 2 lines represent ancilla qubits which are used as scratch space to store an intermediate computation. The technically most interesting part is the next register which consists of a single qubit labeled $m_i$. This indicates that in round $i$, where $1 \leq i \leq 2n$ a fresh qubit is introduced, then acted upon by the circuit and then kept around.

After the maximum number of $2n$ rounds is executed, hence $2n$ qubits have been introduced and entangled in this way. The purpose of the qubit $m_i$ is to remember which of the 4 branches in Kalski's algorithm was taken in step $i$. As there are 4 branches, this choice could be naively encoded into 2 qubits, which however would lead to a space overhead of $4n$ instead of $2n$. The fact that one of these two qubits is actually redundant is shown below. The next qubit, labeled $f$ in the figure, is part of a mechanism to unroll the entire algorithm which drives precisely one of two processes forward: either the Kaliski algorithm itself, or a counter, here represented as the "INC" operation. The flag $f$ starts out in state 1 which indicates that the algorithm is in Kaliski-mode. Once the terminating condition $v = 0$ is reached, the flag switches to 0, indicating that the algorithm is in counter-mode. Finally, the register $k$ holds the state of the counter. As the counter can take values between $n$ and $2n$ only [22], it can be implemented using $\lceil \log_2(n) + 1 \rceil$ qubits only.

Having covered all registers that are part of the circuit, we next explain how the circuit is actually unraveled to compute the almost inverse. Shown in Fig. 9 is only one round. The circuit is applied over and over to the same set of qubit registers, with the sole exception of qubit $m_i$

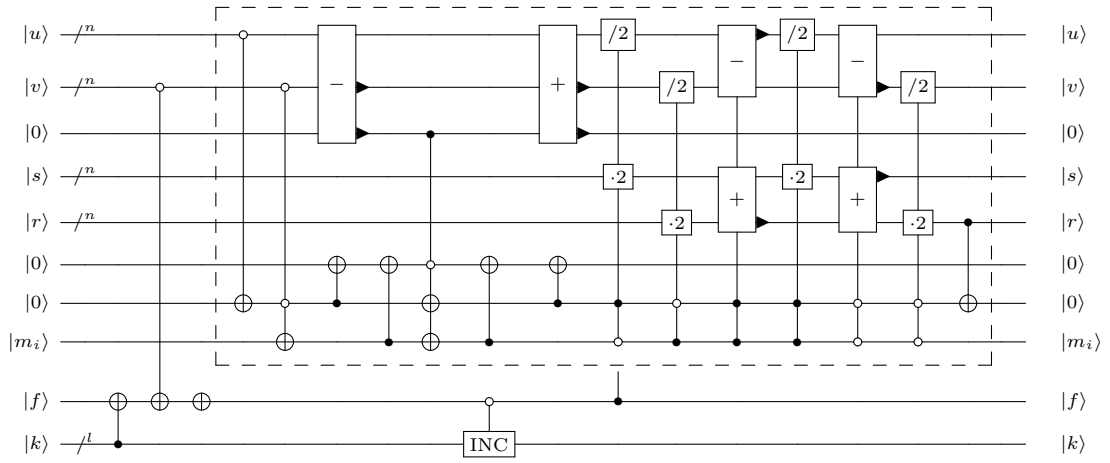


Fig. 9: Quantum circuit for the Montgomery-Kaliski round function. The function is repeated $2n$ times to obtain a reversible modular inversion algorithm. The $n$-qubit registers $|u\rangle$, $|v\rangle$, $|r\rangle$, $|s\rangle$ represent the usual coefficients in the binary Euclidean algorithm. The circuit uses integer subtraction $-$ and addition $+$, as well as multiplication and division by 2 functions $\cdot 2$ and $/2$ and an incrementer circuit INC. The circuits $\cdot 2$ and $/2$ are implemented as cyclic qubit shifts.

which depends on round $i$ and which is initialized, acted upon, and then stored. In each round there are 4 possible branches. These are dispatched using the gates inside the dashed box. The first gate is a controlled NOT that acts only on the least significant bit of $u$, checking whether $u$ is even. The next gate does the same for $v$, which flips the target bit in case $u$ was odd and $v$ was even. If both $u$ and $v$ are odd, the difference $u - v$ respectively $v - u$ has to be computed, depending on whether $u > v$ or $u \leq v$. To figure out which case actually holds, we use a subtractor and store the most significant qubit in the mentioned ancilla. The sequences of 5 gates underneath the two subtractors/adders serve as an encoder that prepares the following correspondence: '10' for the case $u$ even, '01' for the case $u$ odd, $v$ even, '11' for the case both odd and $u > v$, and '00' for the case both odd and $u \leq v$. Denote the two bits involved in this encoding as '$ab$' we see that $b$ is the round qubit $m_i$. The fact that $a$ can be immediately uncomputed is a consequence of the following observation.

In each step of Kaliski's algorithm, precisely one of $r$ and $s$ is even and the other is odd. If the updated value of $r$ is even, then the branch must be the result of either the case $v$ even or the case both $u$ and $v$ odd and $u \leq v$. Correspondingly, if the updated value of $s$ is even, then the branch must have been the result of either the case $u$ even or the case both $u$ and $v$ odd and $u > v$. Indeed, an even value of $r$ arises only from the mentioned two branches $v$ even or $u$ and $v$ both odd and $u \leq v$. Similarly, the other statement is obtained for $s$. The invariant $p = ru + sv$ implies inductively that precisely one of $r$ and $s$ is even and the other henceforth must be odd.

Coming back to the dashed circuit, the next block of 6 gates is to dispatch the appropriate case, depending on the 2 bits $a$ and $b$, which corresponds to the 4 branches in the match statement. Finally, the last CNOT gate between the least significant bit of $r$ (indicating whether $r$ is even) is used to uncompute 'a'.

The shown circuit is then applied precisely $2n$ times. At this juncture, the computation of the almost inverse will have stopped after $k$ steps where $n \leq k \leq 2n$ and the counter INC will have been advanced precisely $2n - k$ times. The counter INC could be implemented using a simple increment $x \mapsto x + 1$, however in our implementation we chose a finite state machine that has a transition function requiring less Toffoli gates.

Next, the register $r$ which is known to hold $-x^{-1}2^k$ is converted to $x^{-1}2^n$. This is done by performing precisely $n-k$ controlled modular doublings and a sign flip. Finally, the result is copied out into another register and the entire circuit is run backwards.

## 4 Reversible elliptic curve operations

Based on the reversible algorithms for modular arithmetic from the previous section, we now turn to implementing a reversible algorithm for adding two points on an elliptic curve. Next, we describe a reversible point addition in the generic case in which none of the exceptional cases of the simple affine Weierstrass group law occurs. After that, we describe a reversible algorithm for computing a scalar multiplication $[m]P$.

### 4.1 Point addition

The reversible point addition we implement is very similar to the one described in Section 4.3 of [34]. It uses affine coordinates. As was also mentioned in [34], it is enough to consider the generic case of an addition. This means that we assume the following situation. Let $P_1, P_2 \in E(\mathbb{F}_p)$, $P_1, P_2 \neq \mathcal{O}$ such that $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$. Furthermore let, $x_1 \neq x_2$ which means that $P_1 \neq \pm P_2$. Recall that then $P_3 = P_1 + P_2 \neq \mathcal{O}$ and it is given by $P_3 = (x_3, y_3)$, where $x_3 = \lambda^2 - x_1 - x_2$ and $y_3 = \lambda(x_1 - x_3) + y_1$ for $\lambda = (y_1 - y_2)/(x_1 - x_2)$.

As explained in [34], for computing the sum $P_3$ reversibly and in place (replacing the input point $P_1$ by the sum), the algorithm makes essential use of the fact that the slope $\lambda$ can be re-computed from the result $P_3$ via the point addition $P_3 + (-P_2)$ independent of $P_1$ using the equation

$$\frac{y_1 - y_2}{x_1 - x_2} = -\frac{y_3 + y_2}{x_3 - x_2}.$$

Algorithm 1 depicts our algorithm for computing a controlled point addition. As input it takes the four point coordinates for $P_1$ and $P_2$, a control bit ctrl, and replaces the coordinates holding $P_1$ with the result $P_3 = (x_3, y_3)$. Note that we assume $P_2$ to be a constant point that has been classically precomputed, because we compute scalar multiples of the input points $P$ and $Q$ to Shor's algorithm by conditionally adding together precomputed 2-power multiples of these points as shown in Figure 1 above. The point $P_2$ will thus always be one of these values. Therefore, operations involving the coordinates $x_2$ and $y_2$ are implemented as constant operations. Algorithm 1 uses two additional temporary variables $\lambda$ and $t_0$. All the point coordinates and the temporary variables fit in $n$-bit registers and thus the algorithm can be implemented with a circuit on a quantum register $|x_1\ y_1\ \text{ctrl}\ \lambda\ t_0\ \text{tmp}\rangle$, where the register tmp holds auxiliary registers that are needed by the modular arithmetic operations used in Algorithm 1 as described in Section 3.

The algorithm is given as a straight line program of (controlled) arithmetic operations on the point coefficients and auxiliary variables. The comments at the end of the line after each operation show the current values held in the variable that is possibly changed. The notation $[\cdot]_1$ shows the value of the variable in case the control bit is ctrl $= 1$, if it is ctrl $= 0$ instead, the value is shown with $[\cdot]_0$. In the latter case, it is easy to check that the algorithm indeed returns the original state of the register.

The functions in the algorithm all use the fact that the modulus $p$ is known as a classical constant. They relate to the algorithms described in Section 3 as follows:

- `add_const_modp` is a modular addition of a constant from a quantum state, `sub_const_modp` is its reverse, a modular subtraction of a constant.
- `ctrl_add_const_modp` is single qubit controlled modular addition of a constant to a qubit register, i.e. the controlled version of the above. Its reverse is the controlled modular subtraction `ctrl_sub_const_modp`.
- `ctrl_sub_modp` is a single qubit controlled modular subtraction on two qubit registers, implemented as the reverse of the corresponding modular addition.
- `ctrl_neg_modp` is a single qubit controlled modular negation on a qubit register.
- `mul_modp`, `squ_modp`, `inv_modp` are the out-of-place modular multiplication, squaring and inversion algorithms on two input qubit registers, respectively.

**Algorithm 1** Reversible, controlled elliptic curve point addition. This algorithm operates on a quantum register holding the point $P_1 = (x_1, y_1)$, a control bit ctrl, and two auxiliary values $\lambda$ and $t_0$. In addition it needs auxiliary registers for the functions that are called as described for those functions. The second point $P_2 = (x_2, y_2)$ is assumed to be a precomputed classical constant. For $P_1, P_2 \neq \mathcal{O}$, $P_1 \neq \pm P_2$, if ctrl $= 1$, the algorithm correctly computes $P_1 \leftarrow P_1 + P_2$ in the register holding $P_1$; if ctrl $= 0$ it returns the register in the same state as it was received.

| | | |
|---|---|---|
| 1: | `sub_const_modp` $x_1$ $x_2$; | // $x_1 \leftarrow x_1 - x_2$ |
| 2: | `ctrl_sub_const_modp` $y_1$ $y_2$ ctrl; | // $y_1 \leftarrow [y_1 - y_2]_1, [y_1]_0$ |
| 3: | `inv_modp` $x_1$ $t_0$; | // $t_0 \leftarrow 1/(x_1 - x_2)$t |
| 4: | `mul_modp` $y_1$ $t_0$ $\lambda$; | // $\lambda \leftarrow [\frac{y_1 - y_2}{x_1 - x_2}]_1, [\frac{y_1}{x_1 - x_2}]_0$ |
| 5: | `mul_modp` $\lambda$ $x_1$ $y_1$; | // $y_1 \leftarrow 0$ |
| 6: | `inv_modp` $x_1$ $t_0$; | // $t_0 \leftarrow 0$ |
| 7: | `squ_modp` $\lambda$ $t_0$; | // $t_0 \leftarrow \lambda^2$ |
| 8: | `ctrl_sub_modp` $x_1$ $t_0$ ctrl; | // $x_1 \leftarrow [x_1 - x_2 - \lambda^2]_1, [x_1 - x_2]_0$ |
| 9: | `ctrl_add_const_modp` $x_1$ $3x_2$ ctrl; | // $x_1 \leftarrow [x_2 - x_3]_1, [x_1 - x_2]_0$ |
| 10: | `squ_modp` $\lambda$ $t_0$; | // $t_0 \leftarrow 0$ |
| 11: | `mul_modp` $\lambda$ $x_1$ $y_1$; | // $y_1 \leftarrow [y_3 + y_2]_1, [y_1]_0$ |
| 12: | `inv_modp` $x_1$ $t_0$; | // $t_0 \leftarrow [\frac{1}{x_2 - x_3}]_1, [\frac{1}{x_1 - x_2}]_0$ |
| 13: | `mul_modp` $t_0$ $y_1$ $\lambda$; | // $\lambda \leftarrow 0$ |
| 14: | `inv_modp` $x_1$ $t_0$; | // $t_0 \leftarrow 0$ |
| 15: | `ctrl_neg_modp` $x_1$ ctrl; | // $x_1 \leftarrow [x_3 - x_2]_1, [x_1 - x_2]_0$ |
| 16: | `ctrl_sub_const_modp` $y_1$ $y_2$ ctrl; | // $y_1 \leftarrow [y_3]_1, [y_1]_0$ |
| 17: | `add_const_modp` $x_1$ $x_2$; | // $x_1 \leftarrow [x_3]_1, [x_1]_0$ |

Figure 10 shows a quantum circuit that implements Algorithm 1. The quantum registers $|x_1\rangle, |y_1\rangle, |t_0\rangle, |\lambda\rangle$ all consist of $n$ logical qubits, whereas $|\text{ctrl}\rangle$ is a single logical qubit. For simplicity in the circuit diagram, we do not show the register $|\text{tmp}\rangle$ with the auxiliary qubits. These qubits are used as needed by the modular arithmetic operations and are returned to their original state after each operation. The largest amount of ancilla qubits is needed by the modular inversion algorithm, which determines that we require $5n$ qubits in the register $|\text{tmp}\rangle$. To avoid permuting the wires between gates, we have used a split gate notation for some modular operations. For all gates, the black triangles mark the output wire that contains the result. As described in Section 3, addition and subtraction gates carry out their operations in place, meaning that one of the input registers is overwritten with the result. Modular multiplication, squaring and inversion operate out of place and store the result in a separate output register.

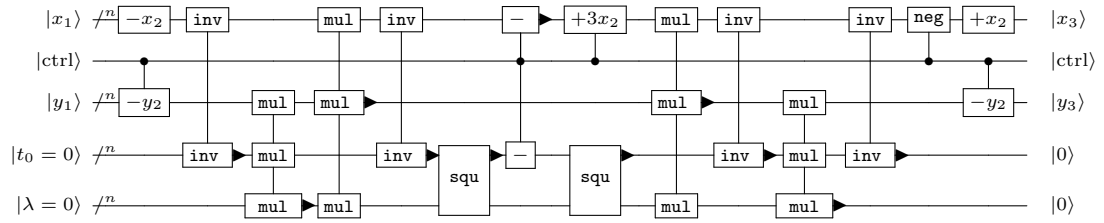


Fig. 10: Quantum circuit for controlled elliptic curve point addition. All operations are modulo $p$ and we use the abbreviations $+ \leftarrow$ `add_modp`, $- \leftarrow$ `sub_modp`, `mul` $\leftarrow$ `mul_modp`, `squ` $\leftarrow$ `squ_modp`, `inv` $\leftarrow$ `inv_modp`.

*Remark 1.* (Projective coordinates) As can be seen from Section 3, modular inversion is by far the most complex and resource consuming part of the elliptic curve point addition. The need for computing and uncomputing the slope $\lambda$ leads to four calls to the inversion in Algorithm 1. In accordance with the observations provided in [34], it accounts for the main cost of the algorithm.

Unsurprisingly, this situation resembles the one for classical modular arithmetic. For example, in elliptic curve cryptography, a modular inversion can be two orders of magnitudes more costly than a modular multiplication, depending on the specific prime field. A significant speed-up can be achieved by using some form of projective coordinates[7], which allow to avoid almost all modular inversions in cryptographic protocols by essentially multiplying through with all denominators. This comes at the relatively small cost of storing more coefficients and a moderate increase in addition and multiplication operations and has proved highly effective. It is thus a natural question to ask whether the use of projective coordinates can also make Shor's algorithm more efficient.

There are several obstacles that make it non-trivial to use projective coordinates for quantum algorithms, such as the fact that each point is represented by an equivalence class of coordinate vectors and the increased number of temporary variables, which need to be uncomputed. In this work, we thus refrained from investigating projective coordinate systems any further and leave it as an open problem to explore their benefits in the context of Shor's algorithm.

### 4.2 Scalar multiplication

In order to compute a scalar multiplication $[m]P$ of a known base point $P$, we also follow the approach outlined in [34]. Namely, by classically precomputing all $n$ 2-power multiples of $P$, the scalar multiple can be computed by a sequence of $n$ controlled additions of those constant points to an accumulator in a quantum register along the binary representation of the scalar. Namely, let $m = \sum_{i=0}^{n-1} m_i 2^i$, $m_i \in \{0, 1\}$, be the binary representation of the $n$-bit scalar $m$. Then,

$$[m]P = [\sum_{i=0}^{n-1} m_i 2^i]P = \sum_{i=0}^{n-1} m_i([2^i]P).$$

This has the advantage that all doubling operations can be carried out on a classical computer and the quantum circuit only requires the generic point addition, which simplifies the overall implementation.

The latter has been argued by Proos and Zalka in [34, Section 4.2]. They say that on average, for any addition step, the probability of an exceptional case is negligibly low, and hence this will only have a negligible influence on the fidelity of the algorithm. To prevent the addition with the point at infinity in the first step, they suggest to initialize the register with a non-zero multiple of the point $P$. For the purpose of estimating resources for Shor's algorithm, we follow the approach by Proos and Zalka and only consider the generic group law. We will have a closer look at the details next.

**Counting scalars with exceptional cases.** As explained in Section 2, Shor's algorithm involves generating a superposition over all possible pairs of $(n+1)$-bit strings $k$ and $\ell$, i.e. the state $\frac{1}{2^{n+1}} \sum_{k,\ell=0}^{2^{n+1}-1} |k, \ell\rangle$. Then over this superposition, involving two additional $n$-qubit registers to hold an elliptic curve point, one computes a double scalar multiplication $\frac{1}{2^{n+1}} \sum_{k,\ell=0}^{2^{n+1}-1} |k, \ell\rangle |[k]P + [\ell]Q\rangle$ of the input points given by the ECDLP instance.

Figure 1 depicts the additional elliptic curve point register to be initialized with a representation of the neutral element $\mathcal{O}$. But if we only consider the generic case of the group law, the first addition of $P$ would already involve an exceptional case due to one of the inputs being $\mathcal{O}$. Proos and Zalka [34] propose to solve this issue by instead initializing the register with a uniform random non-zero multiple of $P$, say $[a]P$ for a random $a \in \{1, 2, \ldots, r-1\}$. Recall that $r$ is the order of $P$ which we assume to be a large prime. Now, if $a \notin \{1, r-1\}$, the first point addition with $P$ works as a generic point addition. With high probability, this solves the issue of an exception in the first addition, but still exceptions occur along the way for many of the possibilities for bit strings $k$ and $\ell$. Whenever a bit string leads to an exceptional case in the group law, it produces a wrong

[7] A collection of possible coordinate systems and the corresponding formulas to carry out the group law is provided at `https://www.hyperelliptic.org/EFD/`.

result for the double scalar multiplication and pollutes the quantum register. We call such a scalar invalid. For Shor's algorithm to work, the overall number of such invalid scalars must be small enough. In the following, we count these scalars to confirm the reasoning in [34].

**Exceptional additions of a point to itself.** Let $a \in \{1, 2, \ldots, r-1\}$ be fixed and write $k = \sum_{i=0}^{n} k_i 2^i$, $k_i \in \{0, 1\}$. We first consider the exceptional case in which both input points are the same, which we call an exceptional doubling. If $a = 1$, this occurs in the first iteration for $k_0 = 1$, because we attempt to add $P$ to itself. This means that for $a = 1$, all scalars $k$ with $k_0 = 1$ lead to a wrong result and therefore half of the scalars are invalid, i.e. in total $2^n$.

For $a = 2$, the case $k_0 = 1$ is not a problem since the addition $[2]P + P$ is a generic addition, but $(k_0, k_1) = (0, 1)$ leads to an exceptional doubling operation in the second controlled addition. This means that all scalars $(0, 1, k_2, \ldots, k_n)$ are invalid. These are one quarter of all scalars, i.e. $2^{n-1}$.

For general $a$, assume that $k$ is a scalar such that the first $i-1$ additions, $i \in \{1, \ldots, n\}$, controlled on the bits $k_0, \ldots, k_{i-1}$ do not encounter any exceptional doubling cases. The $i$-th addition means the addition of $[2^i]P$ for $0 \leq i \leq n$. Then the $i$-th addition is an exceptional doubling if, and only if

$$a + (k_0 + k_1 \cdot 2 + \cdots + k_{i-1} \cdot 2^{i-1}) = 2^i \pmod{r}.$$

If $i$ is such that $2^i < r$. Then, the above condition is equivalent to the condition $a = 2^i - \sum_{j=0}^{i-1} k_j \cdot 2^j$ over the integers. This means that an $a$ can only lead to an exceptional doubling in the $i$-th addition if $a \in \{1, \ldots, 2^i\}$. Furthermore, if $i$ is the smallest integer, such that there exist $k_0, \ldots, k_{i-1}$ such that this equation holds, we can conclude that $a \in \{2^{i-1} + 1, \ldots, 2^i\}$ and $k_{i-1} = 0$. In that case, any scalar of the form $(k_0, \ldots, k_{i-2}, 0, 1, *, \ldots, *)$ is invalid. The number of such scalars is $2^{n-i}$.

If $i$ is instead such that $2^i \geq r$ and if $a \leq 2^i - \mu r$ for some positive integer $\mu \leq \lfloor 2^i / r \rfloor$, then in addition to the solutions given by the equation over the integers as above, there exist additional solutions given by the condition $a = (2^i - \mu r) - \sum_{j=0}^{i-1} k_j \cdot 2^j$, namely $(k_0, \ldots, k_{i-1}, 1, *, \ldots, *)$. The maximal number of such scalars is $\lfloor (2^i - a)/r \rfloor 2^{n-i}$, but we might have counted some of these already.

For a given $a \in \{1, 2, \ldots, r-1\}$, denote by $S_a$ the set of scalars that contain an exceptional doubling, i.e. the set of all $k = (k_0, k_1, \ldots, k_n) \in \{0, 1, \}^{n+1}$ such that there occurs an exceptional doubling when executing the addition $[a + \sum_{j=0}^{i-1} k_j \cdot 2^j]P + [2^i]P$ for any $i \in \{0, 1, \ldots, n\}$. Let $i_a = \lceil \log(a) \rceil$. Then, an upper bound for the number of invalid scalars is given by

$$\#S_a \leq 2^{n-i_a} + \sum_{i=\lceil \log(r) \rceil}^{n} \lfloor (2^i - a)/r \rfloor 2^{n-i}.$$

Hasse's bound gives $\lceil \log(r) \rceil \geq n - 1$, which means that

$$\#S_a \leq 2^{n-i_a} + 2\lfloor (2^{n-1} - a)/r \rfloor + \lfloor (2^n - a)/r \rfloor \leq 2^{n-i_a} + 8.$$

Hence on average, the number of invalid scalars over a uniform choice of $k \in \{1, \ldots, r-1\}$ can be bounded as

$$\sum_{a=1}^{r-1} \Pr(a) \cdot \#S_a \leq \frac{1}{r-1} \sum_{a=1}^{r-1} 2^{n - \lceil \log(a) \rceil} + 8.$$

Grouping values of $a$ with the same $\lceil \log(a) \rceil$ and possibly adding terms at the end of the sum, the first term can be simplified and further bounded by $\frac{1}{r-1}(2^n + \lceil \log(r-1) \rceil 2^{n-1}) = (2 + \lceil \log(r-1) \rceil) \frac{2^{n-1}}{r-1}$. For large enough bitsizes, we use that $r - 1 \geq 2^{n-1}$ and obtain the upper bound on the expected number of invalid scalars of roughly $\lceil \log(r) \rceil + 10 \approx n + 10$. This corresponds to a negligible fraction of about $n/2^{n+1}$ of all scalars.

**Exceptional additions of a point to its negative.** To determine the number of invalid scalars arising from the second possibility of exceptions, namely the addition of a point to its negative, we carry out the same arguments. An invalid scalar is a scalar that leads to an addition $[-2^i]P+[2^i]P$. The condition on the scalar $a$ is slightly changed with $2^i$ replaced by $r-2^i$, i.e.

$$a + (k_0 + k_1 \cdot 2 + \cdots + k_{i-1} \cdot 2^{i-1}) = r - 2^i \pmod{r}.$$

Whenever this equation holds over the integers, i.e. $r - a = 2^i + (k_0 + k_1 \cdot 2 + \cdots + k_{i-1} \cdot 2^{i-1})$ holds, we argue analogously as above. If $2^i < r$ and $r - a \in \{2^i, \ldots, 2^{i+1} - 1\}$, there are $2^{n-i}$ invalid scalars. Similar arguments as above for the steps such that $2^i > r$ lead to similar counts. Overall, we conclude that in this case the fraction of invalid scalars can also be approximated by $n/2^{n+1}$.

**Exceptional additions of the point at infinity.** Since the quantum register holding the elliptic curve point is initialized with a non-zero point and the multiples of $P$ added during the scalar multiplication are also non-zero, the point at infinity can only occur as the result of an exceptional addition of a point to its negative. Therefore, all scalars for which this occurs have been excluded previously and we do not further consider this case.

Overall, an approximate upper bound for the fraction of invalid scalars among the superposition of all scalars due to exceptional cases in the addition law is $2n/2^{n+1} = n/2^n$.

**Double scalar multiplication.** In Shor's algorithm with the above modification, one needs to compute a double scalar multiplication $[a+k]P+[\ell]Q$ where $P$ and $Q$ are the points given by the ECDLP instance we are trying to solve and $a$ is a fixed uniformly random non-zero integer modulo $r$. We are trying to find the integer $m$ modulo $r$ such that $Q = [m]P$. Since $r$ is a large prime, we can assume that $m \in \{1, \ldots, r-1\}$ and we can write $P = [m^{-1}]Q$. Multiplication by $m^{-1}$ on the elements modulo $r$ is a bijection, simply permuting these scalars. Hence, after having dealt with the scalar multiplication to compute $[a+k]P$ above, we can now apply the same treatment to the second part, the addition of $[\ell]Q$ to this result.

Let a be chosen uniformly at random. For any $k$, we write $[a+k]P = [m^{-1}(a+k)]Q$. Assume that $k$ is a valid scalar for this fixed choice of $a$. Then, the computation of $[a+k]P$ did not involve any exceptional cases and thus $[a+k]P \neq \mathcal{O}$, which means that $a + k \neq 0 \pmod{r}$. If we assume that the unknown discrete logarithm $m$ has been chosen from $\{1, \ldots, r-1\}$ uniformly at random, then the value $b = m^{-1}(a+k) \mod r$ is uniform random in $\{1, \ldots, r-1\}$ as well, and we have the same situation as above when we were looking at the choice of $a$ and the computation of $[a+k]P$.

Using the rough upper bound for the fraction of invalid scalars from above, for a fixed random choice of $a$, the probability that a random scalar $k$ is valid, is at least $1 - n/2^n$. Further, the probability that $(k, \ell)$ is a pair of valid scalars for computing $[a+k]P + [\ell]Q$, conditioned on $k$ being valid for computing $[a+k]P$ is also at least $1 - n/2^n$. Hence, for a fixed uniform random $a$, the probability for $(k, \ell)$ being valid is at least $(1 - n/2^n)^2 = 1 - n/2^{n-1} + n^2/2^{2n} \approx 1 - n/2^{n-1}$. This result confirms the rough estimate by Proos and Zalka [34, Section 4.2] of a fidelity loss of $4n/p \geq 4n/2^{n+1}$.

*Remark 2.* (Complete addition formulas) There exist complete formulas for the group law on an elliptic curve in Weierstrass form [6]. This means that there is a single formula that can evaluate the group law on any pair of $\mathbb{F}_p$-rational points on the curve and thus avoids the occurrence of exceptional cases altogether. For classical computations, this comes at the cost of a relatively small slowdown [35]. Using such formulas would increase the algorithm's fidelity in comparison to the above method. Furthermore, there exist alternative curve models for elliptic curves which allow coordinate systems that offer even more efficient complete formulas. One such example is the twisted Edwards form of an elliptic curve [3]. However, not all elliptic curves allow a curve model in twisted Edwards form, like, for example, the prime order NIST curves. We leave it as an open problem to investigate the use of a complete group law, or more generally the use of different curve models and coordinate systems in Shor's ECDLP algorithm.

## 5 Cost and resource estimates for Shor's algorithm

We implemented the reversible algorithm for elliptic curve point addition on elliptic curves $E$ in short Weierstrass form defined over a prime field $\mathbb{F}_p$, where $p$ has $n$ bits, as shown in Algorithm 1 and Figure 10 in Section 4 in F# within the quantum computing software tool suite LIQ$Ui|\rangle$ [47]. This allows us to test and simulate the circuit and all its components and obtain precise counts of the number of qubits, the number of Toffoli gates and the Toffoli gate depth for a working simulation. We thus do not have to rely on mere estimates obtained by pen-and-paper calculations and thus gain a higher confidence in the results. When implementing the algorithms, our overall emphasis was to minimize first the number of required logical qubits and second the Toffoli gate count.

We have simulated and tested our implementation for cryptographically relevant parameter sizes and were able to simulate the elliptic curve point addition circuit for curves over prime fields of size up to 521 bits. For each case, we computed the number of qubits required to implement the circuit, and its size and depth in terms of Toffoli gates.

**Number of logical qubits.** The number of logical qubits of the modular arithmetic circuits in our simulation that are needed in the elliptic curve point addition are given in Table 1. We list each function with its total required number of qubits and the number of ancilla qubits included in that number. All ancilla qubits are expected to be input in the state $|0\rangle$ and are returned in that state, except for the circuits in the first two rows, which only require one or two such ancilla qubits and $n-1$ or $n-2$ ancillas in an unknown state to which they will be returned. The addition, subtraction and negation circuits all work in place, such that one $n$-qubit input register is replaced with the result. The multiplication, squaring and inversion circuits require an $n$-qubit register with which the result of the computation is XOR-ed.

Although the modular multiplication circuit based on modular doubling and additions uses less qubits than Montgomery multiplication, we have used the Montgomery approach to report the results of our experiments. Since the lower bound on the overall required number of qubits is dictated by the modular inversion circuit, neither multiplication approach adds qubit registers to the elliptic curve addition circuit since they can use ancilla qubits provided by the inversion algorithm. We therefore find that the Montgomery circuit is the better choice then because it reduces the number of Toffoli gates substantially.

Because the maximum amount of qubits is used during an inversion operation, the overall number of logical qubits for the controlled elliptic curve point addition in our simulation is

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

In addition to the $7n + 2\lceil \log_2(n) \rceil + 9$ required by the inversion, an additional qubit is needed for the control qubit $|\text{ctrl}\rangle$ of the overall operation and $2n$ more qubits are needed since two $n$-qubit registers need to hold intermediate results during each inversion.

**Number of Toffoli gates and depth.** Perhaps surprisingly, the precise resource count of the number of Toffoli gates in the constructed circuits is not a trivial matter. There are two main reasons for this: one factor is that since constants are folded, the actual value of the constants matter as different bit-patterns (e.g., of the underlying prime $p$) give rise to different circuits. This effect, however, is not large and does not impact the leading order coefficients which for the functions in the table, due to the incrementer construction based on [18], is either of the form $an \log_2(n)$ or $an^2 \log_2(n)$ with a constant $a$. We determined the leading order term by inspection of the circuit and determining how many constant incrementers occur. Then we computed a regression of the next order term. The results are summarized in the last column of Table 1.

Putting everything together, we now obtain an estimate for the entire group law as computed by Algorithm 1: As there are a total of 4 inverters, 2 squarers, and 4 multipliers, we obtain that the leading order coefficient of a single point addition is $224 = 4 \cdot 32 + 2 \cdot 16 + 4 \cdot 16$. We then again

| Modular arithmetic circuit | # of qubits | | # Toffoli |
|---|---|---|---|
| | total | ancillas | gates |
| `add_const_modp`, `sub_const_modp` | $2n$ | $n$ | $16n\log_2(n) - 26.9n$ |
| `ctrl_add_const_modp`, `ctrl_sub_const_modp` | $2n+1$ | $n$ | $16n\log_2(n) - 26.9n$ |
| `ctrl_sub_modp` | $2n+4$ | 3 | $16n\log_2(n) - 23.8n$ |
| `ctrl_neg_modp` | $n+3$ | 2 | $8n\log_2(n) - 14.5n$ |
| `mul_modp` (dbl/add) | $3n+2$ | 2 | $32n^2\log_2(n) - 59.4n^2$ |
| `mul_modp` (Montgomery) | $5n+4$ | $2n+4$ | $16n^2\log_2(n) - 26.3n^2$ |
| `squ_modp` (dbl/add) | $2n+3$ | 3 | $32n^2\log_2(n) - 59.4n^2$ |
| `squ_modp` (Montgomery) | $4n+5$ | $2n+5$ | $16n^2\log_2(n) - 26.3n^2$ |
| `inv_modp` | $7n + 2\lceil\log_2(n)\rceil + 7$ | $5n + 2\lceil\log_2(n)\rceil + 7$ | $32n^2\log_2(n)$ |

Table 1: Total number of qubits and number of Toffoli gates needed for the modular arithmetic circuits used in the elliptic curve point addition on $E/\mathbb{F}_p$ with $n$-bit prime $p$. The column labeled "ancilla" denotes the number of required ancilla qubits included in the total count. Except for the first two rows (un-/controlled constant addition/subtraction), they are expected to be input in state $|0\ldots0\rangle$ and are returned in that state. The constant addition/subtraction circuits in the first row only need one clean ancilla qubit and can take $n-1$ dirty ancilla qubits in an unknown state, in which they are returned. The controlled constant addition/subtraction circuits in the second row use two dirty ancillas.

perform a regression to determine the next coefficient. As a result, we estimate that the number of Toffoli gates in the point addition circuit scales as $224n^2\log_2(n) + 2045n^2$. Figure 11 shows the scaling of the estimates for the Toffoli gate count and the Toffoli gate depth of the circuit for a range of relatively small bit sizes $n$. To estimate the overall resource requirements for Shor's algorithm, one simply multiplies by $2n$, since the controlled point addition is iterated $2n$ times. This leads to the overall estimate for the scaling of the number of Toffoli gates in Shor's ECDLP algorithm as

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

With respect to a given circuit, the Toffoli depth is computed as follows: we sweep all gates in the circuits and keep a running counter for each qubit on which time step it was acted upon last by a Toffoli gate. The depth is then the maximum of these quantities over all qubits. As the number of qubits is comparatively small in the circuits considered here, we can perform these updates efficiently, leading to an algorithm to compute the depth in time linear in the number of gates in the circuit. Note that whenever we encounter a CNOT or NOT gate, we do not increase the counter as by our assumption these gates do not contribute to the overall depth as they are Clifford gates. Overall, we find that the circuit Toffoli depth is a little bit smaller than the total number of Toffoli gates which shows that there is some parallelism in the circuit that can be exploited when implementing it on a quantum computer that facilitates parallel application of quantum gates.

We compare our results to the corresponding simulation results for Shor's factoring algorithm presented in [18], where the corresponding function is modular constant multiplication. In this case, the number of Toffoli gates scales as $32n^2(\log_2(n) - 2) + 14.73n^2$, where $n$ is the bitsize of the modulus to be factored. As above, to estimate the overall resource requirements, one again multiplies by $2n$, which gives $(64(\log_2(n) - 2) + 29.46)n^3$.

Table 2 contains the resources required in our simulated circuits for parameters of cryptographic magnitude that are used in practice. The simulation time only refers to our implementation of the

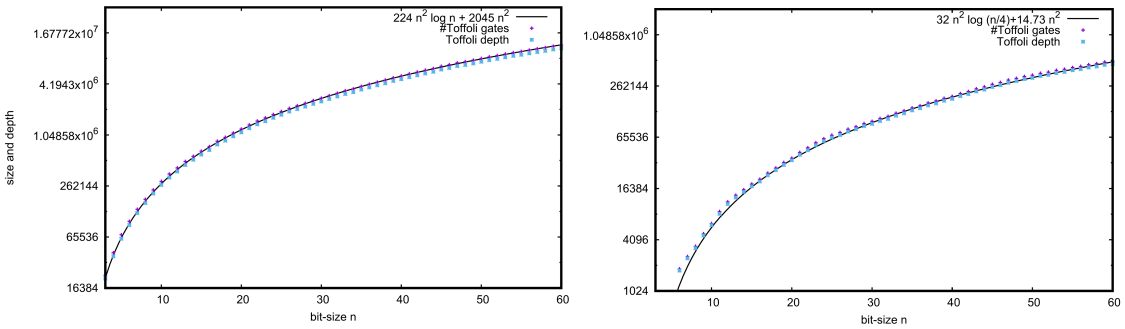


Fig. 11: Shown on the left are resource estimates for the number of Toffoli gates and the Toffoli gate depth for the implementation of elliptic curve point addition $|P\rangle \mapsto |P+Q\rangle$, where $Q$ is a constant point. Shown on the right are resource estimates for the same metrics for modular multiplication $|x\rangle \mapsto |ax \bmod N\rangle$, where $a$ and $N$ are constants. Fitting the data for the elliptic curve case we obtain a scaling as $224n^2 \log_2(n) + 2045n^2$ up to lower order terms. The cost for the entire Shor algorithm over the elliptic curve scales as $2n$ the cost for a single point addition, i.e. $448n^3 \log_2(n)$ up to lower order terms. As shown in [18], the cost for modular multiplication scales as $32n^2(\log_2(n) - 2) + 14.73n^2$ and the cost of the entire Shor factoring algorithm scales as $64n^2 \log_2(n)$.

elliptic curve group law. The simulation timings were measured when running our LIQ$Ui|\rangle$ implementation on an HP ProLiant DL580 Gen8 machine consisting of 4 Intel Xeon processors @ 2.20 Ghz and 3TB of memory.

| ECDLP in $E(\mathbb{F}_p)$ simulation results | | | | | Factoring of RSA modulus $N$ interpolation from [18] | | |
|---|---|---|---|---|---|---|---|
| $\lceil \log_2(p) \rceil$ bits | #Qubits | #Toffoli gates | Toffoli depth | Sim time sec | $\lceil \log_2(N) \rceil$ bits | #Qubits | #Toffoli gates |
| 110 | 1014 | $9.44 \cdot 10^9$ | $8.66 \cdot 10^9$ | 273 | 512 | 1026 | $6.41 \cdot 10^{10}$ |
| 160 | 1466 | $2.97 \cdot 10^{10}$ | $2.73 \cdot 10^9$ | 711 | 1024 | 2050 | $5.81 \cdot 10^{11}$ |
| 192 | 1754 | $5.30 \cdot 10^{10}$ | $4.86 \cdot 10^{10}$ | 1 149 | – | – | – |
| 224 | 2042 | $8.43 \cdot 10^{10}$ | $7.73 \cdot 10^{10}$ | 1 881 | 2048 | 4098 | $5.20 \cdot 10^{12}$ |
| 256 | 2330 | $1.26 \cdot 10^{11}$ | $1.16 \cdot 10^{11}$ | 3 848 | 3072 | 6146 | $1.86 \cdot 10^{13}$ |
| 384 | 3484 | $4.52 \cdot 10^{11}$ | $4.15 \cdot 10^{11}$ | 17 003 | 7680 | 15362 | $3.30 \cdot 10^{14}$ |
| 521 | 4719 | $1.14 \cdot 10^{12}$ | $1.05 \cdot 10^{12}$ | 42 888 | 15360 | 30722 | $2.87 \cdot 10^{15}$ |

Table 2: Resource estimates of Shor's algorithm for computing elliptic curve discrete logarithms in $E(\mathbb{F}_p)$ versus Shor's algorithm for factoring an RSA modulus $N$, stating the required number of qubits and number of Toffoli gates. The same row contains parameters that provide a similar classical security level according to NIST recommendations from 2016. A resource estimate for the number of Toffoli gates in the entire Shor algorithm for solving the ECDLP is obtained by regression as $448n^3 \log_2(n) + 4090n^3$. Resource estimates for factoring are according to the interpolation $(64(\log_2(n) - 2) + 29.46)n^3$ [18]. Simulation time only refers to simulation of a single elliptic curve point addition.

## 6 Discussion

Comparing to the theoretical estimates by Proos and Zalka in [34], our results confirm the overall picture that for cryptographically relevant sizes, elliptic curve discrete logarithms can be computed more easily than a corresponding RSA modulus can be factored at a similar classical security level.

However, neither the Toffoli gate counts for factoring from [18], nor for elliptic curves here are as low as the theoretically predicted "time" estimates in [34]. Also, the number of qubits in our simulation-based estimates is higher than the ones conjectured in [34].

The reasons for the larger number of qubits lie in the implementation of the modular inversion algorithm. Proos and Zalka describe a version of the standard Euclidean algorithm which requires divisions with remainder. We chose to implement the binary GCD algorithm, which only requires additions, subtractions and binary bit shifts. One optimization that applies to both algorithms is register sharing as proposed in [34, Section 5.3.5]. The standard Euclidean algorithm as well as the binary GCD work on four intermediate variables, requiring $4n$ bits in total. In our description in Section 3.4, these are the variables $u, v, r, s$. However, Proos and Zalka use a heuristic argument to show that they actually only need about $2n + 8\sqrt{n}$ bits at any time during the algorithm. A major complication for implementing this optimization is that the boundaries between variables change during the course of the algorithm. We leave it for future work to implement and simulate a reversible modular inversion algorithm that makes use of register sharing to reduce the number of qubits.

Since the basis for register sharing in [34] is an experimental analysis, Proos and Zalka provide a space analysis that does not take into account the register sharing optimization. With this space analysis, we still need about $2n$ qubits more than their Euclidean algorithm. These qubits come from the fact that our extended binary GCD algorithm generates one bit of garbage in each of the $2n$ rounds. In contrast, [34] only needs $n$ carry qubits. Furthermore, we need an additional $n$-qubit register to copy out the result and run the algorithm in reverse to clean-up all garbage and ancilla qubits. From the description in [34], we could not see how to avoid this, and leave it as a future challenge to match or even lower the number of qubits for reversible modular inversion from [34].

To summarize, we presented quantum circuits to implement Shor's algorithm to solve the ECDLP. We analyzed the resources required to implement these circuits and simulated large parts of them on a classical machine. Indeed, the overwhelming majority of gates in our circuits are Toffoli, CNOT, and NOT gates, which implement the controlled addition of an elliptic curve point that is known at circuit generation time. We were able to classically simulate the point addition circuit and hence test it for implementation bugs. Our findings imply that attacking elliptic curve cryptography is indeed easier than attacking RSA, even for relatively small key sizes.

## References


1. Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(6):818–830, June 2013.
2. Stephane Beauregard. Circuit for Shor's algorithm using 2n+3 qubits. *Quantum Info. Comput.*, 3(2):175–185, March 2003.
3. Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards curves. In Serge Vaudenay, editor, *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer, 2008.
4. S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic curve cryptography (ECC) cipher suites for Transport Layer Security (TLS). RFC 4492, 2006.
5. Joppe W. Bos, Craig Costello, and Andrea Miele. Elliptic and hyperelliptic curves: A practical security analysis. In Hugo Krawczyk, editor, *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, volume 8383 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2014.
6. Wieb Bosma and Hendrik W. Lenstra. Complete system of two addition laws for elliptic curves. *Journal of Number Theory*, 53(2):229–240, 1995.
7. Certicom Research. Standards for efficient cryptography 2: Recommended elliptic curve domain parameters. Standard SEC2, Certicom, 2000.
8. Richard Crandall and Carl Pomerance, editors. *Prime Numbers - A Computational Perspective*. Springer, 2005.

9. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol version 1.2. RFC 5246, 2008.
10. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976.
11. Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In Matt Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 303–320. USENIX, 2004.
12. ECC Brainpool. ECC Brainpool Standard Curves and Curve Generation. `http://www.ecc-brainpool.org/download/Domain-parameters.pdf`, 2005.
13. Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Phys. Rev. A*, 86:032324, 2012. arXiv:1208.0928.
14. Steven D. Galbraith and Pierrick Gaudry. Recent progress on the elliptic curve discrete logarithm problem. *Des. Codes Cryptography*, 78(1):51–72, 2016.
15. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1984.
16. Daniel M. Gordon. Discrete logarithms in *GF(P)* using the number field sieve. *SIAM J. Discrete Math.*, 6(1):124–138, 1993.
17. Mike Hamburg. Ed448-Goldilocks, a new elliptic curve. *IACR Cryptology ePrint Archive*, 2015:625, 2015.
18. Thomas Häner, Martin Roetteler, and Krysta M. Svore. Factoring using $2n+2$ qubits with Toffoli based modular multiplication. 2016. arXiv preprint arXiv:1611.07995.
19. Arno Hollosi, Gregor Karlinger, Thomas Rössler, Martin Centner, et al. Die österreichische Bürgerkarte. `http://www.buergerkarte.at/konzept/securitylayer/spezifikation/20080220/`, 2008.
20. Don Johnson, Alfred Menezes, and Scott A. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *Int. J. Inf. Sec.*, 1(1):36–63, 2001.
21. Antoine Joux and Reynald Lercier. Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the gaussian integer method. *Math. Comput.*, 72(242):953–967, 2003.
22. Burton S. Kaliski Jr. The Montgomery inverse and its applications. *IEEE Trans. Computers*, 44(8):1064–1065, 1995.
23. Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Practical approximation of single-qubit unitaries by single-qubit quantum Clifford and $T$ circuits. *IEEE Transactions on Computers*, 65(1):161–172, 2016.
24. Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.
25. A. Langley, M. Hamburg, and S. Turner. Elliptic curves for security. RFC 7748, 2016.
26. Arjen K. Lenstra and Hendrik W. Lenstra, editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer, 1993.
27. Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1985.
28. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
29. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2009.
30. Michael. A. Nielsen and Ike. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, UK, 2000.
31. Kenneth G. Paterson. Formal request from TLS WG to CFRG for new elliptic curves. CFRG mailing list, 2014-07-14, `http://www.ietf.org/mail-archive/web/cfrg/current/msg04655.html`.
32. John M. Pollard. Monte Carlo methods for index computation mod $p$. *Math. Comput.*, 32(143):918–924, 1978.
33. John M. Pollard. Kangaroos, Monopoly and discrete logarithms. *J. Cryptology*, 13(4):437–447, 2000.
34. John Proos and Christof Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Information & Computation*, 3(4):317–344, 2003.
35. Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT*

*2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 403–428. Springer, 2016.
36. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
37. Peter Selinger. Quantum circuits of $T$-depth one. *Phys. Rev. A*, 87:042302, 2013.
38. Peter Selinger. Efficient Clifford+$T$ approximation of single-qubit operators. *Quantum Information & Computation*, 15(1-2):159–180, 2015.
39. Peter W. Shor. 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*, pages 124–134. IEEE Computer Society, 1994.
40. Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.
41. J. H. Silverman. *The Arithmetic of Elliptic Curves (2nd Edition)*. Number 106 in Graduate texts in mathematics. Springer-Verlag, 2009.
42. D. Stebila and J. Green. Elliptic curve algorithm integration in the Secure Shell Transport Layer. RFC 5656, 2009.
43. J. Stein. *Journal of Computational Physics*, 1(3):397–405, 1967.
44. Yasuhiro Takahashi, Seiichiro Tani, and Noboru Kunihiro. Quantum addition circuits and unbounded fan-out. *Quantum Information & Computation*, 10(9&10):872–890, 2010.
45. U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-4, 2013. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`.
46. Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.
47. Dave Wecker and Krysta M. Svore. LIQ$Ui|\rangle$: A Software Design Architecture and Domain-Specific Language for Quantum Computing, 2014. `https://arxiv.org/abs/1402.4467`.
48. WhatsApp Inc. Whatsapp encryption overview. Technical white paper, 2016.
49. Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors. *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*. Springer, 2006.