End-to-end compilable implementation of quantum elliptic curve logarithm in Qrisp

Diego Polimeni[1,2] and Raphael Seidel[3]

[1]Alice & Bob, 53 Bd du Général Martial Valin, 75015 Paris, France
[2]Institute of Physics, École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
[3]Fraunhofer Institute for Open Communication Systems

Elliptic curve cryptography (ECC) is a widely established technique for a variety of applications. Similar to RSA, ECC is however also threatened by Shor based quantum attackers, which made it a popular field of study in quantum information science. A variety of techniques to realize EC arithmetic in quantum devices has been proposed, however to the best of our our knowledge not a single end-to-end compilable implementation has emerged. Within this work we leverage the Qrisp programming language to realize the first implementation of EC arithmetic and verify its correctness using Qrisp's built-in sparse matrix simulator.

1 Overview

Elliptic curve cryptography is a widely established standard for public key cryptography [18]. The underlying one way function for this type of encryption is arithmetic over finite field elliptic curves, which makes this type of encryption vulnerable to Shor's algorithm based attacks. While the core idea around such an attack is relatively straight-forward and similar to Shor's algorithm for factorization, it is the details of elliptic curve arithmetic, which bring a variety of complications into this algorithm. Recently a series of publications gave efficient circuits for several sub-components in order to conduct resource estimation: Roetteler et al. proposed circuit realizations for controlled elliptic curve addition and estimates are derived from a simulation of a Toffoli gate network in the framework Liqui|⟩ [12]. Häner et al. improved quantum circuits for elliptic curve scalar multiplication and while unit tests and automatic quantum resource estimation are conducted through the framework Q# [7]. Gouzien et al. slightly improved quantum circuits and conducted resource estimation on a cat-qubit based architechture [6]. Until today, however, no end-to-end implementation, combining the aforementioned techniques into an executable quantum circuit, seems to have emerged. We attribute this to engineering difficulties, specifically because most of modern quantum software development still happens by manipulating the assembler-like quantum circuit representation. This problem was recently tackled by a novel quantum programming language called Qrisp [14], which remedies many of these flaws. While Qrisp still supports programming on a low-level, its particular architecture enables seamless usage of low-level program specifications in higher order routines.

```
from qrisp import (QuantumModulus,
QuantumArray, h, QFT)
import src.classical.ec_arithmetic as
↪  clECarithm
import src.quantum.ec_arithmetic as qECarithm

#Elliptic curve parameters
p=5
a=3
b=3
curve = clECarithm.EllCurve(a, b, p)

# Quantum type for mod p arithmetic
mod_p = QuantumModulus(p)

# Allocate variables holding
# the elliptic curve point
ecp = QuantumArray(qtype=mod_p, shape=(2,))

#initialize with the point P_0 = [3,2]
ecp[:] = [3,2]

# (Classical) sub-group gGenerator G = [3,2]
G = [3,2]

# (Classical) Target P = (3,3)
P = [3,3]
```

```
# QPE result variables
n = p.bit_length()
x1 = QuantumFloat(n)
x2 = QuantumFloat(n)

# Superposition
h(x1)
h(x2)

# In-place EC arithmetic step 1:
# ecp = P0 + x1*G
qECarithm.ctrl_ell_mult_add(G, ecp, x1, curve)

# In-place EC arithmetic step 2:
# ecp = P0 + x1*G - x2*P
qECarithm.ctrl_ell_mult_add(P, ecp, x2, curve)

# Inverse Quantum Fourier Transform on
# x1 and x2
with inverse():
    QFT(x1)
    QFT(x2)
```

2 Elliptic curve arithmetic

Elliptic curve definition An elliptic curve over a field $\mathbb{K}$, of characteristic different from 2 and 3, is a projective, non-singular curve of genus 1 with a specified base point. It can be defined as the locus of points $(x, y) \in \mathbb{K} \times \mathbb{K}$ satisfying the Weierstrass equation:

$$E : y^2 = x^3 + ax + b \tag{1}$$

where $a, b \in \mathbb{K}$ are constants. The curve is projective since it contains the point at infinity, O. The curve is non-singular if $4a^3 + 27b^2 \neq 0$ which ensures that it doesn't have cusps or nodes. It is relevant to notice that the curve is symmetric with respect to the x-axys, property easily verified by applying the transformation $y \mapsto -y$ to Eq. (1). An elliptic curve is best visualized over the real field $\mathbb{R}$, as shown of Fig. 1. Nonetheless, for cryptographic applications, the field of interest is the finite field $\mathbb{F}_p$ for $p > 3$ a prime number. The set of points consisting of O and all solutions $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ to Eq. (1) is denoted by:

$$\begin{aligned} E(\mathbb{F}_p) =& \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p \mid y^2 = x^3 + ax + b\} \\ & \cup \{O\} \end{aligned} \tag{2}$$

The set $E(\mathbb{F}_p)$ is an abelian group with respect to a group operation "+", called addition, that is defined via rational functions in the point coordinates with O as the neutral element. An extensive treatment of elliptic curves can be found in [17].

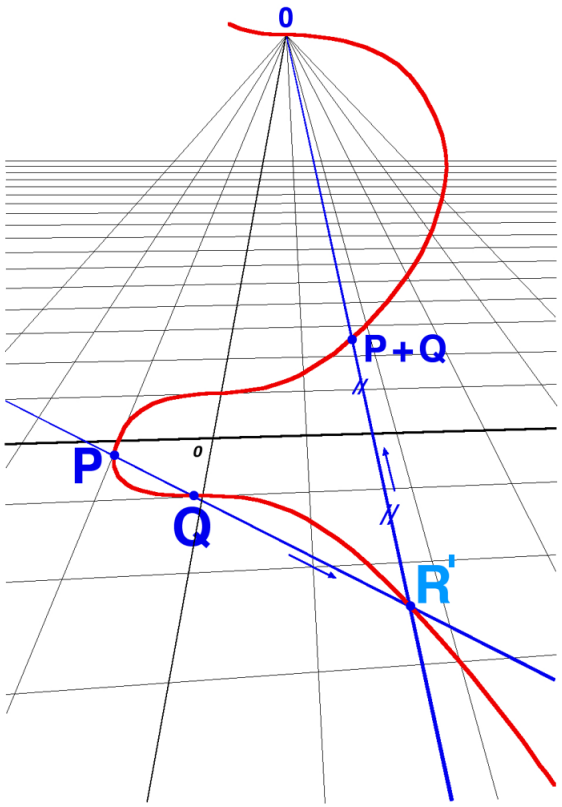

Figure 1: Projective plane representation of the elliptic curve $y^2 = x^3 + 1$. A geometric visualization of the point addition can be appreciated.

Addition The geometric definition of the addition over the elliptic curve is the following: let P and Q be two points on the elliptic curve, the line passing through them intersects the curve in at most another point R'; the result of the addition $R = P + Q$ is the symmetric of R' with respect to the x-axys. Some particular cases arise and need to be defined separately:

- If P and Q are joint by a vertical line, then the result of the addition is the point at infinity: $P + Q = O$.
- If P and Q are joint by a line tangent to the curve in P (respectively Q), the intermediate point R' is equal to P (respectively Q), as the outcome of a double contact between the line and the curve. The result of the addition is

then $P + Q = -P$, (respectively $P + Q = -Q$).

- If $P = Q$ then the line joining them is tangent to the curve in P. The intermediate point R' will be be the intersection of the tangent with the curve and the result R will be the symmetric point with respect to the x-axys.

The geometric definition is consistent with the group structure, since when $Q = O$ the intermediate point will be $-P$ and the result of the addition will be P, highlighting the fact that O is the neutral element for the addition over the elliptic curve.

An equivalent algebraic definition of the addition over the elliptic curve can be given by representing the points on the curve by their coordinates: $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$. When $x_Q \neq x_P$ the slope of the line is given by $\lambda = \frac{y_Q - y_P}{x_Q - x_P}$ division is well-defined as the elliptic curve is defined over a field. The coordinates of the intermediate point R' are obtained by solving the system describing the intersection between the curve and the line, and discarding the solutions corresponding to the points P and Q. These coordinates are:

$$x_{R'} = \lambda^2 - x_P - x_Q \tag{3a}$$
$$y_{R'} = y_P + \lambda(x_{R'} - x_P) \tag{3b}$$

As discussed, the resulting point R will have coordinates:

$$x_R = x_{R'} \tag{4a}$$
$$y_R = -y_{R'} \tag{4b}$$

When $x_Q = x_P$ and $y_Q \neq y_P$, for the horizontal symmetry of the curve, $y_Q = -y_P$, which means that $Q = -P$. The result of the addition $P + (-P) = O$ is the neutral element. When $x_Q = x_P$ and $y_Q = y_P$, which translates to $P = Q$, the slope of the tangent is given by $\lambda = \frac{3x_P^2 + a}{2y_P}$ and the coordinates of the result are given by Eq. (3), except in the case where $y_Q = y_P = 0$, for which the result of the addition is the neutral point.

Scalar multiplication By exploiting the definition of the addition over the elliptic curve, it is possible to define the multiplication of a point by a scalar integer in $\mathbb{Z}$ as such:

$$kP := \underbrace{P + P + ... + P}_{k \text{ times}} \tag{5a}$$
$$0P := O \tag{5b}$$
$$-kP := k(-P) \tag{5c}$$

It is easily verified that the multiplication by a scalar integer is:

- Associative
- Distributive
- Compatible with the identity

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. An essential concept is the order of a point P, noted $ord(P)$, which is the order of the cyclic subgroup generated by P, i.e. the smallest integer r such that $rP = O$.

3 Description of the algorithm

Provided the context described in the previous sections, we define an instance of the ECDLP as such: let $G \in E(\mathbb{F}_p)$ be a fixed and classically known generator of a cyclic subgroup of $E(\mathbb{F}_p)$ of known order $ord(G) = r$, let $P \in \langle G \rangle$ be a fixed and classically known element in the subgroup generated by G; the problem is to retrieve the unique integer $l \in \{1, ..., r\}$, called the discrete logarithm, such that $P = lG$. Shor's algorithm [16], which provides a way to efficiently find the discrete logarithm, proceeds as follow:

- Prepare two registers in a superposition of all possible integers between 0 and a large number.
- Apply a periodic function which takes as input the first two registers and compute the output in an auxiliary register.
- Apply an inverse Quantum Fourier Transform to reveal the period of the function.

Shor's algorithm for solving the ECDLP is analogous to Shor's algorithm to solve the factoring problem, as it prepares a large superpositions

of inputs, feeds them to a periodic function and exploits the QFT routine to reveal the hidden period, which, in turn, allows to efficiently compute the logarithm. The function mentioned in point two is:

$$f(x_1, x_2) = x_1 G - x_2 P = (x_1 - x_2 l) G \tag{6}$$

where the last equality holds for the definition of the discrete logarithm l. The function f is periodic in both variable, as such:

$$\forall k, f(x_1 + kl, x_2 + k) = f(x_1, x_2) \tag{7}$$

Following the outline proposed above, two registers, $|x_1\rangle$ and $|x_2\rangle$ are initialized in a superposition of all possible numbers between 0 and $r-1$:

$$\frac{1}{r} \sum_{x_1=0}^{r-1} \sum_{x_2=0}^{r-1} |x_1\rangle |x_2\rangle \tag{8}$$

The function f is evaluated on those registers as input and the output is stored in a new register:

$$\frac{1}{r} \sum_{x_1=0}^{r-1} \sum_{x_2=0}^{r-1} |x_1\rangle |x_2\rangle |f(x_1, x_2)\rangle \tag{9}$$

Then, the inverse quantum Fourier transform routine is applied to the registers $|x_1\rangle$ and $|x_2\rangle$. The operation yields the result:

$$\frac{1}{r^2} \sum_{x_1, x_2, y_1, y_2=0}^{r-1} e^{2\pi i (x_1 y_1 + x_2 y_2)/r} |y_1\rangle |y_2\rangle |f(x_1, x_2)\rangle \tag{10}$$

We rewrite the sum, for clearer readability, as such:

$$\sum_{y_1, y_2=0}^{r-1} \sum_{k=0}^{r-1} \left[\frac{1}{r^2} \sum_{\substack{x_1, x_2=0 \\ f(x_1,x_2)=kG}}^{r-1} e^{2\pi i (x_1 y_1 + x_2 y_2)/r} \right] |y_1\rangle |y_2\rangle |kG\rangle \tag{11}$$

In this form, it is apparent that the state $|y_1\rangle |y_2\rangle |kG\rangle$ has probability amplitude given by the inner expression of the sum. The sum can be further rewritten noticing that the condition $f(x_1, x_2)(x_1 - x_2 l) G = kG$ leads to $x_1 - x_2 l = k \bmod r$, that is $x_1 = k + x_2 l \bmod r$. This last equality, plugged in Eq. (11), yields the following probability amplitudes associated to the states $|y_1\rangle |y_2\rangle |kG\rangle$:

$$\frac{1}{r^2} \sum_{x_2=0}^{r-1} e^{2\pi i (k y_1 + (y_2 + l y_1) x_2)/r} \tag{12}$$

If $y_2 + l y_1 = 0 \bmod r$, the exponential has zero argument, hence it will contribute to the sum. In any other case, the sum vanishes, as effect of distructive interferences; this can be seen by developing the sum as a geometric sum and noticing that the numerator is always zero, since the argument of the exponential is always a integer multiple of $2\pi i$. Finally, from the measure of the registers encoding y_1 and y_2, the discrete logarithm is retrieved using $l = -y_2 y_1^{-1} \bmod r$, as long as $y_1 \neq 0$. It should be noticed that both y_2 and k run on r different values, which means that the measurement procedure will yield r^2 different outcomes, with uniform probability. However, r of these will lead to $y_2 = 0$ and $y_1 = 0$, thus they are not useful to find the logarithm. Hence, the probability to obtain the value of the discrete logarithm is $1 - \frac{1}{r}$.

3.1 Montgomery form

The Montgomery Form technique [9] is a alternative representation for numbers that operate under modular arithmetic. The motivation behind is the fact that the divisions required for modular reduction are expensive on most computing hardware and should therefore be avoided.
For an arbitrary modular number $a \in \mathbb{F}_p$, the Montgomery form is defined as

$$\tilde{a} = (Ra) \bmod p \tag{13}$$

where R is an arbitrary number with $gcd(R, p) = 1$. Modular reduction on numbers in Montgomery form can be achieved through an algorithm called Montgomery reduction. Although Montgomery

reduction still requires divisions, they can be performed with respect to the divisor R. This can be chosen to be a power of two, thus replacing expensive division with cheap bit-shifts. It should be noticed that addition and subtraction in Montgomery form is analogous to standard modular addition and subtraction:

$$a2^n + b2^n = (a+b)2^n \tag{14a}$$
$$a2^n - b2^n = (a-b)2^n \tag{14b}$$

Multiplication in Montgomery form is slightly less straight-forward. In fact, multiplying $a2^n$ and $b2^n$ does not yield the product of a and b in Montgomery form because of has an extra factor of 2^n:

$$(a2^n \bmod p)(b2^n \bmod p) \bmod p = (ab2^n)2^n \bmod p \tag{15}$$

To obtain the actual result of the multiplication in Montgomery form, the extra factor of 2^n should be removed. This is simply achieved multiplying by the modular inverse of 2^n, conveniently noted 2^{-n}. The multiplication of two `QuantumModulus` has been implemented using the ideas described in [11] and yields a result with a non trivial Montgomery shift. Arithmetic operations between `QuantumModulus` with differing Montgomery shifts could be problematic. There exists two way around that: the first one is to revert the `QuantumModulus` back to standard representation, using the following functions.

```
def to_montgomery_qm(x: QuantumModulus,
↪  montgomery_shift: int):
    x *= pow(2, montgomery_shift, x.modulus)
    x.m = montgomery_shift

def to_standard_qm(x: QuantumModulus):
    montgomery_shift = x.m
    x *= pow(2, -montgomery_shift, x.modulus)
    x.m = 0
```

The second one is needed to implement the arithmetic between `QuantumModulus` with differing Montgomery shifts, commented in section 3.5.

3.2 Kaliski's algorithm

Qrisp offers a set of elementary tools for modular arithmetic, which is however not yet complete. A fundamental ingredient is the modular inversion, used, alongside modular multiplication, to perform modular division and compute the slope λ. The multiplicative modular inverse x^{-1} of a number x is such that $xx^{-1} = x^{-1}x = 1 \pmod p$. First of all, since Montgomery form is used, the modular inversion algorithm must be compatible with it. Kaliski's algorithm is suited for this representation, can handle superposition and does not entangle too many variables. As presented in [8], the algorithm initialize the variables $u = p$, $r = 0$ and $s = 1$, with r such that $p - r = v^{-1}2^{2n} \pmod p$ at the end of the iterations. $v < p$ is the number to invert, which is given in Montgomery form. The variables have the final value of $u = 1$, $r = 0$ and $s = p$, while the modular inverse of v is stocked in the register containing v itself, making it an in-place operation. A more complete treatment of the variable's state can be found in [6]. The psudocode, combining two different version of the Kaliski algorithm, is shown below:

```
def kaliski_swaps(v: int, p: int, n: int):
    u, r, s = p, 0, 1
    for i in range(0, 2*n):
        if v == 0:
            r = r*2 continue
        swap = False
        if (u % 2 == 0 and v % 2 == 1)
or (u % 2 == 1 and v % 2 == 1
and u > v):
            u, v = v, u
            r, s = s, r
            swap = True
        if u % 2 == 1 and v % 2 == 1
            v = v - u
            s = s + r
        v = v//2
        r = 2*r

        if r >= p:
            r = r - p
        if swap:
            u, v = v, u
            r, s = s, r
    r = p - r
    v, r = r, v
    return v
```

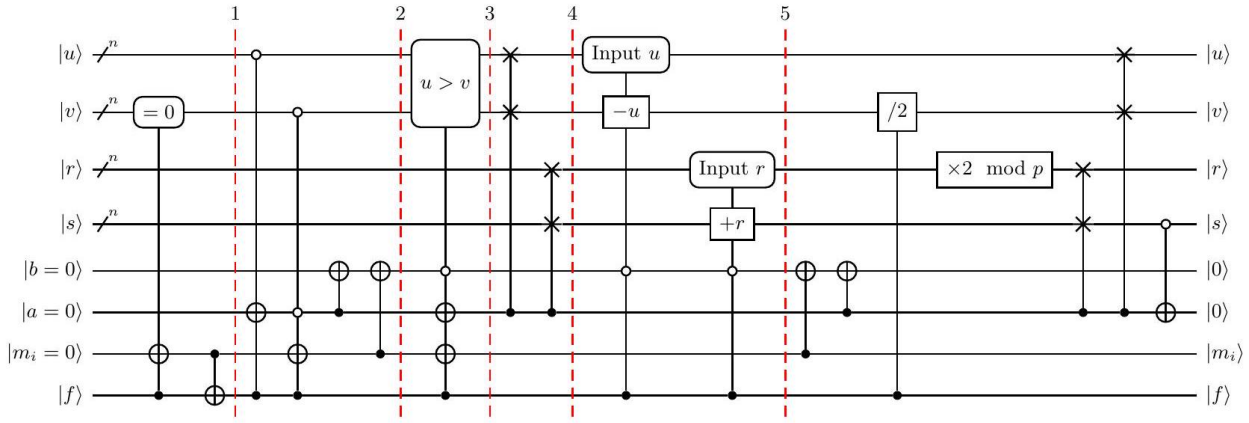

Figure 2: Quantum circuit for the Montgomery-Kaliski round function, which corresponds to one iteration of the loop [6]. Gates controlled by a register are to be understood as to be controlled by the parity of the integer encoded in the register, so the state of the least significant bit. Gates controlled by white dots are triggered when the control qubit is in the '0' state.

Implementing an high-level version of the algorithm, following the psuedocode, revealed to be feasible and classical inputs were correctly treated; however, compilation time and simulation time for superposition of inputs were considerably high, which disqualified this implementation to be used as a subroutine. Instead, a gate level implementation of the algorithm was chosen, restricted to more primitives operations such as simple Toffoli or Fredkin gates. For the comparisons and the modular arithmetic we still used the Qrisp defaults though. Fig. 2, taken from [6], represents the circuit for one iteration of the loop. Notice that the algorithm would stop when the condition $v = 0$ is reached, which happens between n and $2n$ iterations of the loop; however, to ensure reversibility and in a quantum circuit, the unnecessary operations are deactivated the remaining iterations of the loop are run. The last qubit, in state $|f\rangle$, modelled trough `QuantumBool`, signals when the "stop" condition $v = 0$ is met. Initially set to 1, it changes to 0 at the start of the first iteration where $v = 0$ is detected. After this, only modular doubling of r and uncomputations are performed. Qubits $|a\rangle$, $|b\rangle$ and $|m_i\rangle$ are used to decide which branch of the algorithm is run and are modelled trough `QuantumBool` .After the comparison step, the variable a models the condition for applying the swaps in the pseudocode (second `if`), while the logical complement of b models the condition for applying the subtraction and addition in the pseudocode (third `if`). After the modular doubling and before the controlled swap operation, a modular reduction of r is perfomed; this operation is implicitly grouped with the doubling, on the circuit. It is important to notice that while a and b are allocated within the algorithm and can therefore be deallocated within it, leveraging some conserved quantities, a garbage qubit $|m_i\rangle$ is produced for each iteration of the algorithm and remains polluted after the algorithm. For this reason, a `QuantumArray` of pre-allocated `QuantumBool` is passed as a an argument to the function. The uncomputation strategy, is discussed in the next section. Comparisons and arithmetic operations are not detailed here since are built-in function in Qrisp.

The whole Kaliski's algorithm is implemented by the following function:

```
from qrisp import (QuantumBool, QuantumFloat,
QuantumModulus, QuantumArray, control, invert,
x, cx, mcx, swap, cyclic_shift)

def kaliski_quantum(v: QuantumModulus,
                    m: QuantumArray):
    """
    Utilizes the Kaliski-Algorithm to perform
    an in-place mod inversion on the
    QuantumModulus v.

    m is a QuantumArray cotaining pre-allocated
    QuantumBools, which are required for the
    algorithm as temporary values but will be
    uncomputed using Bennet's trick.
    """

    p = v.modulus
    n = p.bit_length()
    # Convert to Montgomery Form
    to_montgomery(v, p)
    u = QuantumFloat(n)
```

```
    u[:] = p
    r = QuantumModulus(2 * p)
    r[:] = 0
    s = QuantumModulus(2 * p)
    s[:] = 1

    a = QuantumBool()
    b = QuantumBool()
    add = QuantumBool()
    f = QuantumBool()
    f[:] = True
    for i in range(2 * n):
        is_zero = v == 0
        mcx([f, is_zero], m[i])
        is_zero.uncompute()
        cx(m[i], f)
        # STEP 1
        mcx([f, u[0]], a, ctrl_state="10")
        mcx([f, a, v[0]], m[i],
        ↪  ctrl_state="100")
        cx(a, b)
        cx(m[i], b)

        # STEP 2
        l = u > v
        mcx([f, l, b], a, ctrl_state="110")
        mcx([f, l, b], m[i], ctrl_state="110")
        l.uncompute()

        # STEP 3
        with control(a):
            swap(u, v)
            swap(r, s)

        # STEP 4
        mcx([f, b], add, ctrl_state="10")
        with control(add):
            v -= u
            s += r
        # STEP 5
        mcx([f, b], add, ctrl_state="10")
        # uncompute b
        cx(m[i], b)
        cx(a, b)

        # Division by 2
        with control(f):
            with invert():
                cyclic_shift(v)

        cyclic_shift(r)
        larger = r > p
        with larger:
            r -= p
        cx(r[0], larger)
        larger.delete()

        with control(a):
            swap(u, v)
            swap(r, s)
        # uncompute a
        mcx([s[0]], a, ctrl_state="0")
        a.delete()
        add.delete()
        b.delete()

        inpl_rsub(r, p)

        for i in range(v.size):
            swap(v[i], r[i])

        # Uncompute u,s,f
        f.delete()
        x(u[0])
        u.delete()
        r.delete()
        s -= p
        s.delete()
        # Convert back to standard representation
        to_standard(v, p)
        return v
```

After $2n$ iteration of the Kaliski's loop, the transformation $|r\rangle \rightarrow |p-r\rangle$ is applied using the `inpl_rsub` function, allowing an in-place operation that does not pollute additional qubits. Using the properties of the two-complement representation, a negation of the bits induces the transformation $|r\rangle \rightarrow |-(r+1)\rangle$. Subsequently we add the modulus + 1 to bring the variable into a positive state, which now represents $|(-r) \mathrm{mod} N\rangle$. The procedure is finalized by performing a semi-classical modular in-place addition with p. This is implemented by the following function:

```
def inpl_rsub(r: QuantumModulus,
              p: int):
    """
    Performs a modular in-place
    right-subtraction.
    Implying after executing this function, the
    new r has the value (p-r)
    """

    # Use the two complement for negation:
    # NOT(r) = -(r + 1)
    x(r)

    # Make positive again
    # The inpl_adder attribute calls a
    # user-specified, 2^n modular adder
    r.inpl_adder(r.modulus + 1, r)

    # Modular addition of p
    r += p
```

3.3 Elliptic curve addition

Addition over elliptic curves, as described in Section 2.1, is implemented in three steps. The first one is the classical doubling of the generator P, the second one is the quantum addition of a classically known point to a quantum point, the third one is the combination of the first one and the controlled version of the second one. To compute the scalar multiplication kP of a known base point P efficiently, we precompute all n 2-power multiples of P classically. Then, we compute the scalar multiple using a sequence of n controlled additions of these precomputed points and store the result in a quantum register, following the binary representation of the scalar. Let$k = \sum_{i=0}^{n-1} k_i 2^i$, with $k_i \in \{0, 1\}$, be the binary representation of the n-bit scalar k. Then:

$$kP = \sum_{i=0}^{n-1} k_i 2^i P = \sum_{i=0}^{n-1} k_i (2^i P) \tag{16}$$

This approach offers the benefit that all doubling operations can be performed on a classical computer, leaving the quantum circuit responsible only for the generic point addition. This simplification makes the overall implementation more straightforward and less demanding in terms of resources. The doubling of the classically known generator is implemented as follows:

```
def ell_double(P: list, curve: EllCurve):

    #Return 2P
    p = curve.p
    s = ((3 * (P[0] * P[0] % p) + curve.a) % p)
    ↪  * pow((2 * P[1]) % p, -1, p)
    xr = (s * s - 2 * P[0]) % p
    yr = P[1] - s * ((P[0] - xr) % p) % p

    return [xr, (p - yr) % p]
```

Following [12] and [6], the implementation of elliptic curve addition focuses on the generic case where the two points are distinct, not inverses of each other, and neither is the neutral element. These exceptional cases are rare, except when the accumulation register is initialized to the neutral element. To avoid this issue, the accumulation register should be initialized with a point other than the neutral element, which minimally affects the measurement statistics after the Fourier transform. As described above, a classically known point is added to a point whose coordinates are encoded in a quantum register.

The implementation of the addition in Qrisp follows the circuit given on Fig. 3, adapted from [12], to which we refer the reader for more details. The addition is implemented as in-place operation, meaning that the coordinates of the result are stored in the registers containing the coordinates of the inputs.

```
from qrisp import (custom_control,
↪  QuantumArray,
QuantumBool, QuantumModulus, conjugate,
control, cx, Qubit)

@custom_control
def ell_add_inpl(anc: QuantumArray,
                         G: list,
                         p: int,
                         ctrl: Qubit = None):
    """
    Performs in-place elliptic curve point
    addition of a point whose coordinates are
    stored in a quantum register and a
    classically known point; the result of the
    operation is stored in the registers
    initially containing the coordinates of the
    input.
    """
    # STEP 1
    if ctrl is None:
        anc[1] -= G[1]
    else:
        with control(ctrl):
            anc[1] -= G[1]
    anc[0] -= G[0]

    # STEP 2
    l = QuantumModulus(p)

    # m is an array of temporary QuantumBools,
    # which will be uncomputed Bennett style
    m = QuantumArray(qtype=QuantumBool(),
                     shape=2*p.bit_length())

    with conjugate(kaliski_quantum)(anc[0], m)
    ↪  as inv:
        temp = anc[1] * inv
        to_standard_qm(temp)
        l[:] = temp
        temp.uncompute()
    for a in m:
        a.delete()

    temp = l * anc[0]
    to_standard_qm(temp)
    anc[1] -= temp
    temp.uncompute()

    # STEP 3
    if ctrl is None:
        anc[0] += 3 * G[0]
    else:
        with control(ctrl):
```

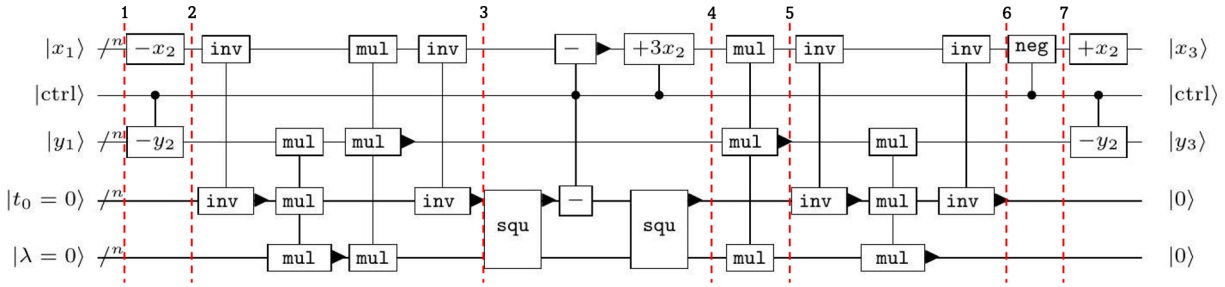

Figure 3: Quantum circuit for controlled elliptic curve point addition [12]. Five register are involved, two of which are uncomputed at the end of the operation: two quantum registers, $|x_1\rangle$ and $|y_1\rangle$, to encode the coordinates of the point, a control qubit $|ctrl\rangle$ and two auxiliary registers $|t_0\rangle$ and $|\lambda\rangle$. The rectangles specify the type of operation performed and the registers involved, while the black triangle designates the output register.

```
        anc[0] += 3 * G[0]

    temp = l * l
    to_standard_qm(temp)
    if ctrl is None:
        anc[0] -= temp
    else:
        with control(ctrl):
            anc[0] -= temp

    temp.uncompute()

    # STEP 4
    temp = l * anc[0]
    to_standard_qm(temp)
    anc[1] += temp
    temp.uncompute()

    # STEP 5
    m = QuantumArray(qtype=QuantumBool(),
    ↪  shape=2 * p.bit_length())

    with conjugate(kaliski_quantum)(anc[0], m)
    ↪  as inv:
        temp = anc[1] * inv
        to_standard_qm(temp)
        cx(temp, l)
        temp.uncompute()

    for a in m:
        a.delete()

    # STEP 6
    l.delete()
    if ctrl is None:
        anc[1] -= G[1]
    else:
        with control(ctrl):
            anc[1] -= G[1]

    # STEP 7
    anc[0] -= G[0]

    if ctrl is None:
        inpl_rsub(anc[0], p)
    else:
        with control(ctrl):
            inpl_rsub(anc[0], p)

    return anc
```

Finally, the classical and quantum routines are combined toghether, using the decomposition provided by Eq. 16 to control the addition. The controlled version of the addition is much more expensive that the plain version, which leads us to implement a series of optimisations as the `custom_control` decorator, whose functioning is detailed in [14].

```
from qrisp import control, invert, cyclic_shift

def ctrl_ell_mult_add(power: list,
                      res: QuantumArray,
                      k: QuantumFloat,
                      curve: EllCurve):
    # Elliptic curve multiplication Q + kP
    n = k.size
    p = curve.p
    for i in range(n):
        with control(k[i]):
            res = ell_add_inpl(res, power, p)
        power = ell_double(power, curve)
    return res
```

3.4 Modular adder from an arbitrary non-modular adder

As elaborated, the EC addition requires more primitive arithmetic operation such as a modular adder. Such an adder has been described in [1]; however, it uses QFT based addition, which is expected to be particularly costly on fault tolerant devices due to the cost of synthesizing ar-

bitrary angle rotations [13]. We generalize the overall strategy to enable the use of an arbitrary (non-modular) adder to perform modular addition using only Clifford + T gates (or whichever gates the adder backend requires).

```
from qrisp import (QuantumBool, QuantumFloat,
Qubit, invert, control, custom_control)

@custom_control
def mod_adder(a : QuantumFloat,
              b : QuantumFloat,
              inpl_adder : callable,
              modulus : int,
              ctrl : Qubit = None):
    """
    Performs the modular in-place addition
    b += a
    The adder backend can be specified in form
    of a function "inpl_adder" which performs
    non-modular addition.
    """

    reduction_not_necessary = QuantumBool()
    sign = QuantumBool()

    b = list(b) + [sign[0]]

    if ctrl is None:
        inpl_adder(a, b)
    else:
        with control(ctrl):
            inpl_adder(a, b)

    with invert():
        inpl_adder(modulus, b)

    cx(sign, reduction_not_necessary)

    with control(reduction_not_necessary):
        inpl_adder(modulus, b)

    with invert():
        if ctrl is None:
            inpl_adder(a, b)
        else:
            with control(ctrl):
                inpl_adder(a, b)

    cx(sign, reduction_not_necessary)
    reduction_not_necessary.flip()

    if ctrl is None:
        inpl_adder(a, b)
    else:
        with control(ctrl):
            inpl_adder(a, b)

    sign.delete()
    reduction_not_necessary.delete()
```

Note the `custom_control` decorator which enables an efficient controlled version of this function. To learn more about this feature please refer to [14].
Qrisp offers a variety of built-in adder backend implementations such as the Draper adder [4], the Gidney adder [5], the Cuccaro adder [3] or the QCLA introduced by Wang et al. [19].

3.5 Modular adder on arbitrary Montgomery shifts

As mentioned, instead of reverting back the `QuantumModulus` to the standard representation after each quantum-quantum multiplication, it might be more efficient to have the arithmetic interoperable between arbitrary Montgomery shifts. To achieve this, consider two numbers $a, b \in \mathbb{F}_p$ in Montgomery form with respective Montgomery shift $k, l \in \mathbb{N}$. The sum of both in Montgomery form with shift l is

$$(a+b)2^l \bmod p \tag{17}$$
$$=(2^{l-k}a2^k + b2^l) \bmod p \tag{18}$$
$$=(\sum_{i=0}^{n} \tilde{a}_i 2^{i-k} + b)2^l \bmod p \tag{19}$$

Where $\tilde{a}_i$ is the bitstring representing $a2^k \bmod p$. This bitstring is directly available since a is encoded in Montgomery form with shift k. We can now execute the sum as a series of controlled modular additions.

```
def montgomery_addition(a : QuantumModulus,
                        b : QuantumModulus):
    """
    Performs the modular inplace addition
    b += a
    where a and b don't need to have the same
    montgomery shift
    """"
    for i in range(len(a)):
        with control(a[i]):
            # .m is the attribute, which
            # stores the Montgomery shift
            b += 2**(i-a.m)%(a.modulus)
```

However, a benchmark of the quantum operations of the two proposed strategies showed that they are comparable and there is not a significant advantage of using the latter instead of the former. We conclude that further research is required to find a definitive answer to this question.

3.6 Uncomputation

As with many quantum algorithms, the exponential speed-up of Shor's is fundamentally related to interference phenomena. Interference between several branches can however only occur if they are disentangled completely, implying the EC arithmetic has to act in-place and every temporary value has to be uncomputed. Achieving uncomputation is however in many cases far from trivial, which posed several challenges when developing the code. In this work several techniques for uncomputation are leveraged.

- The **Unqomp algorithm** [10], which enables an automatic procedure for synthesizing uncomputation. This algorithm has been implemented within Qrisp [15] and can be called via the `.uncompute` method of the `QuantumVariable` class. While this way of uncomputation is especially convenient because of its automation, it comes with the disadvantage that the values which were required for the computation need to be accessible[1] or otherwise the Unqomp DAG will contain a cycle. This condition is satisfied for some of the uncomputations here but not for all.

- **Problem specific approaches** to uncomputation. In situations where the Unqomp algorithm fails because the required values for uncomputation are no longer accessible, it is sometimes possible to leverage knowledge about the program state to reverse the computation by using an alternative way for uncomputation. An example for this can be found in Figure 2, where the value a is uncomputed in a way that is fundamentally different from it's computation. Another interesting example of this is the uncomputation of the "Montgomery garbage" in [11].

- **Bennett's trick** [2] is a technique for uncomputing the intermediate values of an arbitrary reversible classical function f. The procedure can be roughly summarized like this

  1. Compute the result $y = f(x)$.
  2. Copy y into a result register.
  3. Reverse f by using y and the intermediate values from step 1.

  While this procedure indeed also applies for classical functions implemented on a quantum computers, it comes with two important drawbacks.

  – f has to be evaluated twice (forwards and backwards).

  – The memory requirements of f grow with the amount of intermediate values required since the occupied space can't be recycled during the execution of f.

  Because of the heavy toll on resource requirements, Bennett's trick should in practice therefore be seen as a "last resort" to uncomputation.

Within our implementation we effectively leverage a combination of all these techniques. When executing Kaliski's algorithm, the b values (see Fig. 2) can be essentially uncomputed the same way they have been computed, however prevent memory churn[2], we manually uncompute (instead of call `.uncompute`) to prevent deallocation of b, such that the variable can be reused in the next iteration. The a values require the problem specific approach - for the details we refer to [6]. The m values seem to allow no direct uncomputation, which is why we had to resort to Bennett's trick [2]. We implement this using the Qrisp inbuilt `ConjugationEnvironment` [14], which provides a structured interface to achieve this task. Note that the a and b values are uncomputed after each iteration, so the Bennett related memory overhead is restricted to the m values.

4 Conclusion

In this work, we provided and assembled all the necessary components to solve the ECDLP using Shor's algorithm. We detailed the implementation of in-place modular inversion, specifically Kaliski's algorithm, within the Qrisp quantum computing framework. Additionally, we described the implementation of point addition over

[1] For further details what accessible means in this context we refer to the Unqomp paper [10].

[2] Memory churn is the compiler overhead that comes with the cost of (de)allocating a lot of variables.

elliptic curves, its controlled version, and scalar multiplication. Although some routines were developed in previous works, an end-to-end compilable implementation seemed to be missing in the world of open-source software. Indeed, many challenges arose when trying to integrate these routines with each other. Many questions about performance and compilation remain and will be addressed in future versions of this work.

5 Code availability

Qrisp is an open-source python framework for high-level programming of quantum computers. The source code is available in https://github.com/eclipse-qrisp/Qrisp. The source code for solving the ECDLP using Shor's algorithm is currently under an NDA between Alice&Bob and Fraunhofer FOKUS, and will be published when this paper will be published.

References

[1] Stephane Beauregard. Circuit for shor's algorithm using 2n+3 qubits, 2003.

[2] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[3] Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, and David Petrie Moulton. A new quantum ripple-carry addition circuit, 2004.

[4] Thomas G. Draper. Addition on a quantum computer, 2000.

[5] Craig Gidney. Halving the cost of quantum addition. *Quantum*, 2:74, June 2018.

[6] Élie Gouzien, Diego Ruiz, Francois-Marie Le Régent, Jérémie Guillaud, and Nicolas Sangouard. Performance analysis of a repetition cat code architecture: Computing 256-bit elliptic curve logarithm in 9 hours with 126133 cat qubits. *Physical Review Letters*, 131(4), July 2023.

[7] Thomas Häner, Samuel Jaques, Michael Naehrig, Martin Roetteler, and Mathias Soeken. Improved quantum circuits for elliptic curve discrete logarithms, 2020.

[8] B.S. Kaliski. The montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.

[9] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[10] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. Unqomp: synthesizing uncomputation in quantum circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 222–236, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Rich Rines and Isaac Chuang. High performance quantum modular multipliers, 2018.

[12] Martin Roetteler, Michael Naehrig, Krysta M. Svore, and Kristin Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms, 2017.

[13] Neil J. Ross and Peter Selinger. Optimal ancilla-free clifford+t approximation of z-rotations, 2016.

[14] Raphael Seidel, Sebastian Bock, René Zander, Matic Petrič, Niklas Steinmann, Nikolay Tcholtchev, and Manfred Hauswirth. Qrisp: A framework for compilable high-level programming of gate-based quantum computers, 2024.

[15] Raphael Seidel, Nikolay Tcholtchev, Sebastian Bock, and Manfred Hauswirth. *Uncomputation in the Qrisp High-Level Quantum Programming Framework*, page 150–165. Springer Nature Switzerland, 2023.

[16] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997.

[17] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer-Verlag, 2nd edition, 2009.

[18] Shamsher Ullah, Jiangbin Zheng, Nizamud Din, Muhammad Tanveer Hussain, Farhan Ullah, and Mahwish Yousaf. Elliptic curve

cryptography; applications, challenges, recent advances, and future trends: A comprehensive survey. *Computer Science Review*, 47:100530, 2023.

[19] Siyi Wang, Anubhab Baksi, and Anupam Chattopadhyay. A higher radix architecture for quantum carry-lookahead adder, 2023.