

Received 7 March 2025; accepted 24 June 2025; date of publication 7 July 2025;  
date of current version 28 August 2025.

Digital Object Identifier 10.1109/TQE.2025.3586541

# Quantum Resource Estimates for Computing Binary Elliptic Curve Discrete Logarithms

MICHAEL GARN<sup>1</sup>  AND ANGUS KAN<sup>2,3</sup> 

<sup>1</sup>The Hartree Centre, STFC, Sci-Tech Daresbury, WA4 4AD Warrington, U.K.

<sup>2</sup>PsiQuantum, WA4 4FS Daresbury, U.K.

<sup>3</sup>PsiQuantum, Palo Alto, CA 94304 USA

Corresponding authors: Michael Garn; Angus Kan (e-mail: michael.garn@stfc.ac.uk; akan@psiquantum.com).

**ABSTRACT** We perform logical and physical resource estimation for computing binary elliptic curve discrete logarithms using Shor’s algorithm on fault-tolerant quantum computers. We adopt a windowed approach to design our circuit implementation of the algorithm, which comprises repeated applications of elliptic curve point addition operations and table look-ups. Unlike previous work, the point addition operation is implemented exactly, including all exceptional cases. We provide exact logical gate and qubit counts of our algorithm for cryptographically relevant binary field sizes. Furthermore, we estimate the hardware footprint and runtime of our algorithm executed on surface-code matter-based quantum computers with a baseline architecture, where logical qubits have nearest-neighbor connectivity, and on a surface-code photonic fusion-based quantum computer with an active-volume architecture, which enjoys a logarithmic number of nonlocal connections between logical qubits. At 10% threshold and compared to a baseline device with a 1- $\mu$ s code cycle, our algorithm runs  $\gtrsim$  2–20 times faster, depending on the operating regime of the hardware and over all considered field sizes, on a photonic active-volume device.

**INDEX TERMS** Elliptic curve, quantum circuits, quantum computing, quantum resource estimation, Shor’s algorithm.

## I. INTRODUCTION

Shor’s algorithm [1], [2] computes discrete logarithms for finite Abelian groups in polynomial time and thus can be used to break Rivest–Shamir–Adleman (RSA) cryptography [3], which is based on the hardness of integer factorization. It can also be applied to elliptic curve groups to efficiently break elliptic curve cryptography (ECC) [4]. A large corpus of work has put the theoretical vulnerability of these cryptosystems against quantum attacks in more concrete terms by analyzing and reducing the quantum-computational resources, e.g., gate and qubit counts, which will be sufficient to break them [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]. To safeguard against sufficiently powerful quantum computers, the National Institute of Standards and Technology has recently proposed timelines to transition to postquantum cryptography [27], [28] over the next decade.

Putting practicality aside, computing discrete logarithms is scientifically intriguing, as it belongs to a very selective class of problems, for which a quantum algorithm achieves

a super-polynomial speed-up over any existing classical algorithms, and the computed solution is efficiently verifiable classically. In this work, we focus on solving the discrete logarithm problem for elliptic curves over a binary field. In particular, we are interested in, under reasonable assumptions about the quantum computer architecture, estimating 1) the size of a quantum computer that will be sufficient to solve this problem for cryptographically relevant field sizes, and 2) the runtime of such a computation.

*Architectural assumptions:* The cryptographically relevant problem sizes well exceed the capability of current nonerror-corrected quantum computers; we expect that they can only be solved on fault-tolerant quantum computers (FTQC) enabled by error correction. We will focus on resource estimates for FTQC based on the surface code [9], [29], [30], [31], [32], [33], [34], which encode each logical qubit in a 2-D patch of physical qubits. We consider two types of surface-code architectures: 1) baseline architectures where physical and logical qubits communicate via nearest-neighbor operations on a 2-D grid [9], [32], [33], [34], and

2) the active-volume (AV) architecture [20] that utilizes a logarithmic amount of nonlocal connections between logical qubits; these nonlocal connections facilitate a higher level of parallelization between logical operations, which, in turn, bring about a significant speed-up, compared to a baseline FTQC with a similar physical footprint but only 2-D, local connectivity.

*Previous work:* There have been significant efforts over the past two decades in constructing and optimizing quantum algorithms that solve the binary elliptic curve discrete logarithm problem (ECDLP) [5], [6], [7], [8], [10], [16], [19], [22], [23], [24], [25], [26], [35]. Particularly germane to this article is the work that provides explicit resource estimates in terms of logical gate and qubit counts, and optimize for Toffoli count [16], [24], [25], [26]. This is so because the product of Toffoli count and logical qubit count largely determines the runtime and footprint for baseline architectures [18], [21], [33]. However, to the best of the authors' knowledge, unlike existing work on RSA and ECC over a prime field [13], [17], [18], [20], [21], all the resource estimates from existing binary ECC work remain at an abstract, hardware agnostic level, i.e., logical gate and qubit counts, and stop short of estimating physical, hardware-relevant resources, e.g., physical runtime and number of physical qubits for matter-based FTQC, or number of resource-state generators (RSGs) for photonic FTQC.

*Our contributions:* We provide both abstract and physical resource estimates for both baseline and AV architectures. We focus on superconducting and atomic hardware, including trapped-ion and neutral-atom platforms, for baseline architectures, and photonic fusion-based quantum computing [36] for the AV architecture. We stress that in principle, baseline and AV architectures are both hardware-agnostic. The association between baseline architectures and matter-based hardware, and the association between AV architecture and fusion-based photonic platforms [36], [37], [38], [39], [40], [41] are motivated by practical reasons: Long-range connections between matter-based qubits come with a variety of challenges, e.g., frequency conversion [42], high-quality cavities [43], [44], low-rate Bell measurements [45], [46], and slow shuttling [47], whereas low-loss fiber and high-quality photonic-chip-to-fiber coupling could more directly support long-range connections between photonic qubits hosted on separate chips [48], [49]. Following the work in [20] and [21], we estimate the number of physical qubits and runtime on matter-based platforms, and estimate the number of RSGs and runtime on a photonic fusion-based quantum computer (FBQC), called for by our algorithm.

Our other contributions include pedagogical reviews of recent advances in binary-field arithmetic quantum circuits [24], [25], [26], which implement known classical algorithms [50], [51], [52], [53], [54], and are used in our algorithm, as well as optimizations and necessary corrections therein. Furthermore, our binary elliptic curve point addition routine incorporates all exceptional cases of the point addition operation, including, e.g., the point-doubling case [4],

[10], which has been previously attempted [35] although not accomplished.

The rest of this article is organized as follows. In Section II, we provide necessary background on binary ECC. In Section III, we provide an overview of our algorithm and the subroutines employed therein, supplemented by materials in the Appendixes. In Section IV, we present the methods used to estimate the abstract and physical computational resources, and our estimates for relevant binary-field sizes. Finally, Section V concludes this article.

## II. BINARY ELLIPTIC CURVES

Binary elliptic curves are elliptic curves defined over a binary field  $\mathbb{F}_{2^n}$ . We use a polynomial basis representation:  $\mathbb{F}_{2^n}$  is identified with  $\mathbb{F}_2[x]/p(x)$ , where  $p(x) \in \mathbb{F}_2[x]$  is an irreducible polynomial of degree  $n$ . Then, the elements in  $\mathbb{F}_{2^n}$  are represented as polynomials of degree less than  $n$  with binary coefficients in  $\mathbb{F}_2$ . All computations are done modulo  $p(x)$ . We adopt polynomials  $p(x)$  that are used in the standardized binary elliptic curves listed in [55] and displayed in Table 9.

An ordinary binary elliptic curve is given by

$$y^2 + xy = x^3 + ax^2 + b \quad (1)$$

where  $a \in \mathbb{F}_{2^n}$  and  $b \in \mathbb{F}_{2^n}^*$ . The set of points on a curve consists of tuples  $P = (x, y) \in \mathbb{F}_{2^n}^2$ , which satisfy equation (1), and the so-called point at infinity  $\mathcal{O}$ . This set forms a group under point addition, where given  $P_1 = (x_1, y_1)$  and  $P_2 = (x_2, y_2)$ , and  $P_3 = (x_3, y_3)$  is conventionally given by [10]

$$(x_3, y_3) = \begin{cases} \mathcal{O} & \text{if } P_1 = -P_2 \\ (x_1, y_1) & \text{if } P_2 = \mathcal{O} \\ (x_2, y_2) & \text{if } P_1 = \mathcal{O} \end{cases} \quad (2)$$

else if  $P_1 = P_2$

$$(x_3, y_3) = \left( \lambda^2 + \lambda + a, x_2^2 + (\lambda + 1)x_3 \right) \quad (3)$$

with  $\lambda = x_2 + \frac{y_2}{x_2}$ , else if  $P_1 \neq \pm P_2$

$$(x_3, y_3) = (\lambda^2 + \lambda + x_1 + x_2 + a, (x_2 + x_3)\lambda + x_3 + y_2) \quad (4)$$

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

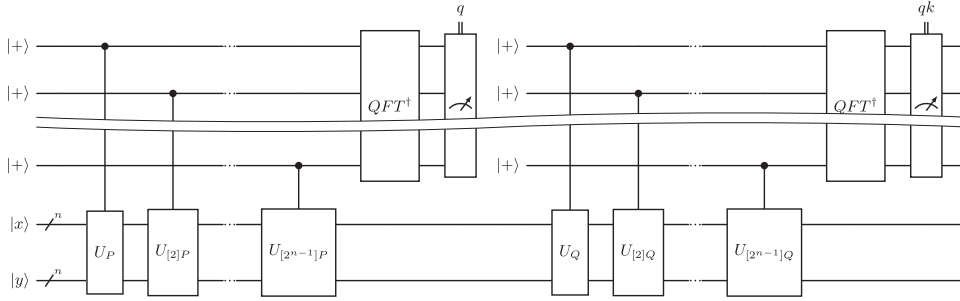
Here,  $-P_2 = (x_2, y_2 + x_2)$ . We choose  $(0,0)$  as the representation of  $\mathcal{O}$  [10]. We work with a recasted form of elliptic curve point addition, which we hereafter abbreviate to EC-POINTADD; we rewrite the cases when  $P_1 = P_2$ , i.e., (3), and  $P_1 \neq \pm P_2$ , i.e., (4), in the following form:

else

$$(x_3, y_3) = (\lambda^2 + \lambda + x_1 + x_2 + a, (x_2 + x_3)\lambda + x_3 + y_2)$$

$$\text{with } \lambda = \begin{cases} \lambda_r = x_2 + \frac{y_2}{x_2} = \frac{y_3 + x_3 + y_2}{x_2 + x_3}, & \text{if } P_1 = P_2 \\ \frac{y_1 + y_2}{x_1 + x_2} = \frac{y_3 + x_3 + y_2}{x_2 + x_3}, & \text{otherwise.} \end{cases} \quad (5)$$

Using the fact that each polynomial is represented as a bit-string, where the  $i$ th bit is the  $i$ th polynomial coefficient, and that adding two polynomials is done



**FIGURE 1.** Shor's algorithm circuit that computes the private key  $k$  from the base point  $P = (x, y)$  and public key  $Q = [k]P$ .

via bitwise XOR, as we explain in Section III-C and Appendix A, one can show that (5) is equivalent to (3) and (4). Consider first the point-doubling case  $P_1 = P_2$ , i.e., (3).  $x_3 \stackrel{(3)}{=} \lambda^2 + \lambda + a = \lambda^2 + \lambda + x_1 + x_2 + a$ , as claimed in (5), because  $x_1 + x_2 = 0$ . Next,  $y_3 \stackrel{(3)}{=} x_2^2 + (\lambda + 1)x_3 = \lambda x_2 + y_2 + (\lambda + 1)x_3 = (x_2 + x_3)\lambda + x_3 + y_2$ , as claimed in (5), because  $x_2^2 = (x_2 + \frac{y_2}{x_2})x_2 + y_2 \stackrel{(3)}{=} \lambda x_2 + y_2$ . Finally,  $y_3 = (x_2 + x_3)\lambda + x_3 + y_2$  implies that  $\lambda = \frac{y_3 + x_3 + y_2}{x_2 + x_3}$ , as claimed in (5). The case where  $P_1 \neq \pm P_2$ , i.e., (4), is now manifestly handled.

The Diffie–Hellman key-exchange mechanism and security of binary ECC rely on the fact that while a sum of  $k$   $P$ 's under point addition, denoted hereafter by  $Q = [k]P$ , can be computed classically in polynomial time via (3), there is no known polynomial-time classical algorithm that computes  $k$  (private key) from  $P$  (base point) and  $Q$  (public key). This problem is known as the binary ECDLP. For more background on ECC, consult, e.g., [56].

### III. ALGORITHM AND SUBROUTINES

In this section, we present our construction of Shor's algorithm for the binary ECDLP. We start by reviewing the high-level structure of the algorithm, and then proceed to break it down into fundamental subroutines, among which, the binary-field arithmetic routines are discussed in detail.

#### A. ALGORITHM STRUCTURE

The high-level circuit of Shor's algorithm for computing binary ECDLP is displayed in Fig. 1. It consists of two rounds of quantum phase estimation (QPE) applied to the unitaries  $U_P$  and  $U_Q$ , which add the base point  $P$  and public key  $Q = [k]P$ , respectively, to any input  $(x, y)$ . The first QPE is performed on the input state  $|P\rangle = |x\rangle \otimes |y\rangle$ ; it prepares an eigenstate of  $U_P$  with nontrivial overlap with  $|P\rangle$

$$|\phi_q\rangle = \sum_{j=0}^{r-1} e^{iqj\frac{2\pi}{r}} |[j]P\rangle \quad (6)$$

whose eigenphase is given by

$$U_P|\phi_q\rangle = e^{-iq\frac{2\pi}{r}} |\phi_q\rangle \quad (7)$$

and outputs a nonnegative integer  $q < r$ , where  $r$  is the order of  $P$ , i.e.,  $[r]P = P$ .  $|\phi_q\rangle$  is also an eigenstate of  $U_Q$  with

$$\begin{aligned} U_Q|\phi_q\rangle &= \sum_{j=0}^{r-1} e^{iqj\frac{2\pi}{r}} |[j]P + Q\rangle = \sum_{j=0}^{r-1} e^{iqj\frac{2\pi}{r}} |[j+k]P\rangle \\ &= e^{-iqk\frac{2\pi}{r}} |\phi_q\rangle. \end{aligned} \quad (8)$$

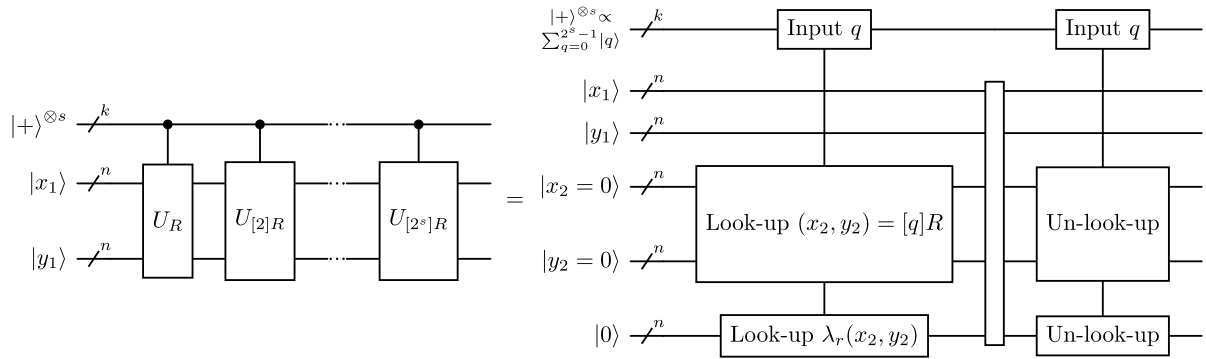
Hence, the second QPE will output  $qk$ , from which  $k$  can be obtained after dividing  $qk$  by  $q$ .

We follow the windowing method from [15], [18], and [21] to implement groups of controlled point addition operations. In particular, we first divide the  $n$  controlled point additions into groups of  $s$  contiguous operations (the last group will contain less than  $s$  operations if  $n$  is not divisible by  $s$ ), and then implement each group using 1) a QROM look-up of  $2^s$  classically computed points  $[q]R$  for  $q \in [0, 2^s - 1]$  and  $\lambda_r$ -values; 2) an uncontrolled ECPOINTADD operation; and 3) an uncomputation of the QROM, as shown in Fig. 2, thereby reducing the number of calls to point additions. The window size  $s$  that minimizes the resource requirements depends on the field size  $n$ ; we discuss this further in Section IV.

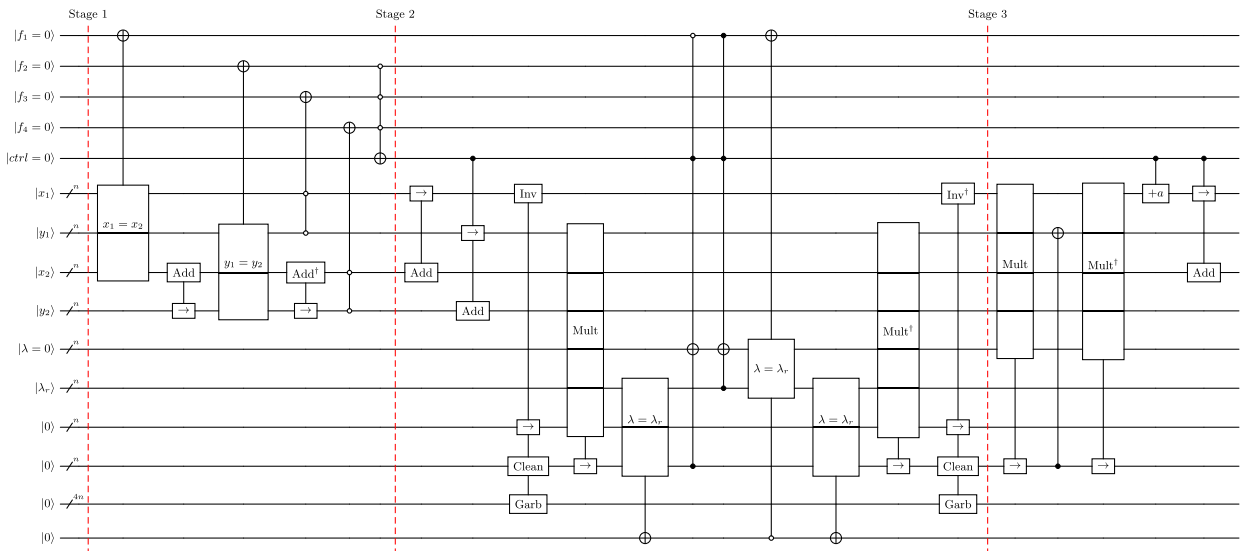
#### B. ELLIPTIC CURVE POINT ADDITION

As a result of our recasting ECPOINTADD's definition, i.e., from (3) and (4) to (5), to a form that is similar to the prime ECPOINTADD definition presented in [21], our ECPOINTADD circuit shares a similar structure to that for prime ECC in [21], along with the corrections suggested in [57] and necessary changes—notably the arithmetic routines—to adapt to the differences between binary and prime ECC. Unlike previous binary ECC work [16], [24], [26], our construction implements the entire point addition including, e.g., the point-doubling case, i.e., (3), and the case where either added points is  $\mathcal{O}$ . Even though the impact of neglecting exceptional cases on the success probability of the algorithm is negligible [11], constructing an exact point addition circuit is still of interest [35].

Overall, our ECPOINTADD circuit performs an in-place addition of two points  $(x_1, y_1)$  and  $(x_2, y_2)$ . Each coordinate is stored in an  $n$ -qubit register, and the output  $(x_3, y_3)$  will be returned in the registers that originally stored  $x_1$  and  $y_1$ . In addition, the circuit requires  $n$  qubits to store  $\lambda_r$ , five ancilla



**FIGURE 2.** Following the work in [15] and [21], we implement a group of  $s$  controlled-addition of a point  $R$  using a look-up table, uncontrolled point addition operation, and an uncomputation of the look-up table. The optimal window size  $s$  is calculated for every considered elliptic curve field size.



**FIGURE 3.** First half of the ECPOINTADD circuit in Fig. 2. Notations: We use “→” to mark the target registers whose values are modified by modular addition, multiplication, and inversion. We use “Garb” and “Clean” to label zero-in-garbage-out and zero-in-zero-out ancilla qubits. Note that NOTs with only open controls are Toffoli gates, those with one solid control are series of CNOT gates, and those with one or more solid controls on any of the first five qubits are series of CNOT gates controlled by one or more of the first five qubits.

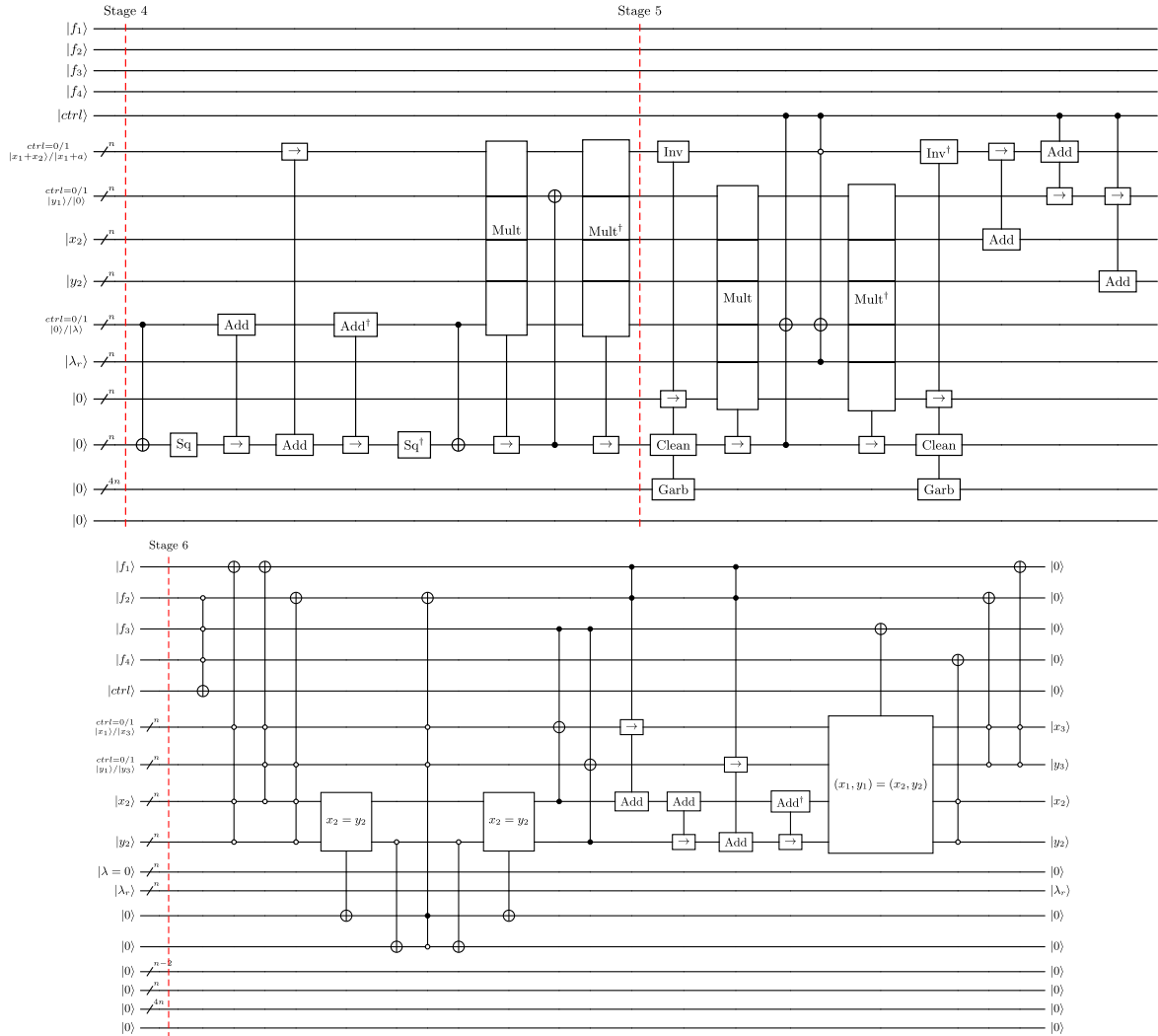
**TABLE 1.** Subroutine Counts of the ECPOINTADD Circuit in Figs. 3 and 4

Subroutine	Count
$n$ -qubit (Controlled) equality test	9
$n$ -qubit Toffoli gate	30
Addition	8
Controlled addition	6
Inversion	4
Multiplication	8
Controlled constant-addition	1
Squaring	2

In this table, we count a  $2n$ -qubit Toffoli gate as two  $n$ -qubit Toffoli gates, since the constant difference in their fault-tolerant costs is negligible. For the same reason, we count a  $(n+c)$ -qubit Toffoli gate with  $c \leq 3$  or  $n$  3-qubit Toffoli gates as a single  $n$ -qubit Toffoli Gate, and a  $2n$ -qubit equality test as two  $n$ -qubit ones.

qubits to flag the exceptional cases, and  $6n$  qubits to implement the conditional logic and arithmetic operations in the point addition. We list and count the subroutines in Table 1. In what follows, we explain the circuit in six stages, shown in Figs. 3 and 4.

In the first stage, we set the flags  $f_1, f_2, f_3,$  and  $f_4$  to 1 when  $x_1 = x_2, y_1 = x_2 + y_2, (x_1, y_1) = \mathcal{O},$  and  $(x_2, y_2) = \mathcal{O},$  respectively, using equality checks and multicontrolled Toffoli gates. The ctrl is set to 1 if  $f_2 = f_3 = f_4 = 0,$  meaning that none of the input or output points are  $\mathcal{O},$  and that we are in the branch (5) of ECPOINTADD. In the second stage, we compute  $\lambda$  into a clean ancilla register, either (i) using modular arithmetic over binary fields if  $f_1 = 0$  and ctrl = 1 or (ii) by copying  $\lambda_r$  if  $f_1 = 1$  and ctrl = 1, corresponding to the two cases in (5). Then, we reset  $f_1$  if the outputs of (i) and (ii) are not equal and  $\lambda = \lambda_r,$  before uncomputing intermediate arithmetic steps. By the end of Stage 2, registers 6 and 7 in Fig. 3 are in  $|x_1 + x_2\rangle$  and  $|y_1\rangle/|y_1 + y_2\rangle$  if ctrl = 0/1, and register 10 is in  $|0\rangle/|\lambda\rangle$  if ctrl = 0/1. Stage 3 maps registers 6 and 7 to  $|x_1 + x_2\rangle/|x_1 + a\rangle$  and  $|y_1\rangle/|0\rangle,$  respectively, if ctrl = 0/1; then, stage 4 maps them to  $|x_1 + x_2\rangle/|x_2 + x_3\rangle$  and  $|y_1\rangle/|y_2 + y_3 + x_3\rangle,$  respectively, if ctrl = 0/1. Note that in Stage 3, we have used the equality  $\lambda = \frac{y_1 + y_2}{x_1 + x_2}$  from the last line in (5), and in Stage 4, we have used



**FIGURE 4.** Second half of the ECPPOINTADD circuit in Fig. 2. Notations: We denote the squaring operation by “Sq.” In stage 4, the NOT gates controlled by a register are bitwise CNOT gates. In stages 5 and 6, the controlled NOT gates are all Toffoli gates.

the equalities  $x_2 + x_3 = \lambda^2 + \lambda + x_1 + a$  and  $\lambda = \frac{y_3 + x_3 + y_2}{x_2 + x_3}$  from (5).

The fifth stage uncomputes  $|\lambda\rangle$  and maps registers 6 and 7 to  $|x_1\rangle/|x_3\rangle$  and  $|y_1\rangle/|y_3\rangle$  if  $ctrl = 0/1$ , which completes the branch (5) of ECPPOINTADD. Specifically, this is done by either computing  $\lambda$  from  $|x_2 + x_3\rangle$  (register 6) and  $|x_3 + y_2 + y_3\rangle$  (register 7), or by adding  $|\lambda_r\rangle$  into  $|\lambda\rangle$  when  $|x_2 + x_3 = 0\rangle$  and  $|ctrl = 1\rangle$ .  $|\lambda_r + \lambda = 0\rangle$  because the conditions  $|x_2 + x_3 = 0\rangle$  and  $|ctrl = 1\rangle$  imply the point-doubling case, i.e.,  $P_1 = P_2$ , which we now prove. Along with the conditions, we suppose  $P_1 \neq P_2$  for the sake of contradiction. Then, (5) and  $x_2 + x_3 = 0$  imply that  $x_1 + x_2 = 0$  because  $\lambda = \frac{x_3 + y_2 + y_3}{x_2 + x_3} = \frac{y_1 + y_2}{x_1 + x_2}$ . Furthermore, by adding  $x_2$  to both sides of  $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$ , taken from (5), and then multiplying both sides by  $(x_1 + x_2)^2$ , we get the equation  $(y_1 + y_2)^2 + (y_1 + y_2)(x_1 + x_2) + (x_1 + a)(x_1 + x_2) = 0$ . Finally, since  $x_1 + x_2 = 0$ , the equation simplifies to  $(y_1 + y_2)^2 = 0$ , which implies  $y_1 = y_2$ . This leads to a contradiction because if  $x_1 = x_2$  (implied by  $x_1 + x_2 = 0$ ) and  $y_1 + 1 = y_2$ , then  $P_1 = P_2$ .

In the final stage, we first reset the  $ctrl$ , and then handle the exceptional cases. The first two Toffoli gates targeting  $f_1$  clear  $f_1$  for the cases where 1)  $(x_2, y_2) = (0, 0)$ ,  $x_1 = 0$ ,  $y_1 \neq 0$ , and 2)  $(x_1, y_1) = (0, 0)$ ,  $x_2 = 0$ ,  $y_2 \neq 0$ . The first two Toffoli gates targeting  $f_2$ , with the latter being conjugated by two Toffoli gates and equality checks, clear  $f_2$  for the cases where 1)  $(x_2, y_2) = (0, 0)$ ,  $x_1 \neq 0$ ,  $y_1 = 0$  and 2)  $(x_1, y_1) = (0, 0)$ ,  $x_2 = y_2 \neq 0$ . If  $f_3 = 1$ , then  $(x_1, y_1) = \mathcal{O}$ , and thus,  $(x_2, y_2)$  is copied into the output registers, before resetting  $f_3$  via an equality check. If  $f_4 = 1$ , then  $(x_2, y_2) = \mathcal{O}$ ; thus, we can simply return  $(x_1, y_1)$  and reset  $f_4$  using a  $2n$ -qubit Toffoli. If  $f_1 = f_2 = 1$ , which implies  $P_1 = -P_2$ , then we set the output to  $\mathcal{O} = (0, 0)$  via arithmetic operations, before resetting the flags.

### C. ARITHMETIC ROUTINES

Now, we proceed to describe the arithmetic operations over binary fields used in our ECPPOINTADD circuit, namely, modular addition, multiplication, and inversion. Note that here

we will provide high-level descriptions of these operations and leave the finer details in the Appendix.

We first consider modular addition of two polynomials  $f(x)$  and  $g(x)$ , defined by

$$f(x) + g(x) = \sum_{i=0}^{n-1} (f_i + g_i)x^i \quad (9)$$

where  $f_i$  is the  $i$ th coefficient of  $f(x)$  and addition is over  $\mathbb{F}_2$ . This is realized by coefficientwise binary addition, i.e., XOR, which in a quantum circuit, is done by applying  $n$  CNOT gates to add  $|f\rangle$  into an  $n$ -qubit input state  $|g\rangle$ , thereby mapping  $|g\rangle$  to  $|f + g\rangle$ . When  $f(x)$  is a classically known constant polynomial, this operation, which we call constant-addition, is realized as an in-place circuit. This circuit consists of at most  $n$  NOT gates applied to  $|g\rangle$ , with one NOT gate for each monomial in  $f(x)$  that has a coefficient of one. A controlled addition can be implemented by simply controlling every CNOT, i.e., turning CNOTs into Toffolis. A controlled constant-addition, i.e.,  $f(x)$  is a constant, is implemented by a series of at most  $n$  CNOTs.

Next, we describe modular multiplication between two degree- $(n - 1)$  polynomials  $f(x)$  and  $g(x)$ , i.e.,

$$f(x) \cdot g(x) = f(x)g(x) \bmod p(x) \quad (10)$$

where  $p(x)$  is a degree- $n$  irreducible polynomial. We implement this using the algorithm from [25], with appropriate modifications. This algorithm is based on well-established classical techniques: it combines Karatsuba-like recurrence formulas from [53] and [54] and the Chinese remainder theorem (CRT) over binary fields [51], [52] (see Theorem 1 in Appendix B). We choose to use this algorithm because compared to the modular multiplication method from [14] and [16], this algorithm has a much lower Toffoli count.

To start, we note that  $f(x)g(x) = f(x)g(x) \bmod m(x)$ , when  $m(x)$  is an arbitrary polynomial with a degree larger than  $2n - 2$ . Furthermore,  $m(x) = \prod_{i=1}^t m_i(x)$ , where the  $m_i(x)$ 's are pairwise coprime polynomials that have degrees  $d_i$ . We display our choices of  $m_i(x)$ 's in Table 10. Then, using the CRT, we can break the product down into products between smaller polynomials  $f^i(x) = f(x) \bmod m_i(x)$  and  $g^i(x) = g(x) \bmod m_i(x)$ . The modular reduction to  $f^i(x)$  can be computed via  $f^i = f_{0,\dots,d_i-1} + M_i f_{d_i,\dots,n-1}$ , where  $f_{i,\dots,j}$  are a subset of contiguous coefficients, running from  $i$  to  $j$ , of  $f(x)$  and  $M_i$  is a binary matrix derived from  $m_i(x)$ . In a quantum circuit, this amounts to XORing certain bits from  $|f_{d_i,\dots,n-1}\rangle$  into  $|f_{0,\dots,d_i-1}\rangle$ , where the locations of the CNOTs that perform the XORs are determined by  $M_i$  [16]. Next, we compute  $c^i(x) = f^i(x)g^i(x) \bmod m_i(x)$  for  $i = 1, \dots, t$ . If  $d_i \leq 8$ , we use existing Karatsuba-like formulas [53], [54] to compute  $c^i(x)$ ; these formulas can be translated to quantum circuits using [25, Algorithm 1], which we explain in Appendix B. If  $d_i > 8$ , we recursively invoke the CRT-based multiplication algorithm. Following the CRT, we then

combine the  $c^i(x)$ 's into a single polynomial using

$$c'(x) = \sum_{i=1}^t (c^i(x)q_i(x) \bmod m(x)) \bmod p(x) \quad (11)$$

where

$$q_i(x) = \left( \frac{m(x)}{m_i(x)} \right) \left( \left( \frac{m(x)}{m_i(x)} \right)^{-1} \bmod m_i(x) \right) \quad (12)$$

are classically precomputed constant polynomials. For each  $i$ , this involves multiplying  $c^i(x)$  by the constant polynomial  $q_i(x)$ , modulo  $m(x)$  and  $p(x)$ ; this is a linear transformation of  $q_i(x)$  over  $\mathbb{F}_2$  and can be expressed as a matrix-vector multiplication, i.e.,  $Q_i c^i$ , where  $Q_i$  is a matrix that depends on  $q_i(x)$ ,  $m(x)$ , and  $p(x)$ . Using permutation-lower-and-upper-triangular (PLU) decompositions,  $Q_i$  can be decomposed into a sequence of permutation and lower and upper triangular matrices [25]; the permutation and triangular matrices prescribe a sequence of swap and CNOT gates, respectively [10], [16] (see Step 3 in Appendix B). If the degree of  $m(x)$  is larger than  $2n - 2$ , then  $c'(x)$  is the desired product and we are done. Otherwise, i.e., the degree of  $m(x) \leq 2n - 2$ , we need to apply a correction and compute

$$c(x) = c'(x) + \sum_{i=2n-1-\omega}^{2n-2} c_i((x^i)) + (x^i \bmod m(x)) \bmod p(x) \quad (13)$$

where  $\omega = 2n - 1 - \deg(m(x))$ , and  $c_i$ 's are referred to as correction coefficients. To implement this in a quantum circuit, the multiplication by the classically precomputed polynomial  $((x^i) + (x^i \bmod m(x))) \bmod p(x)$  is carried out using the previously mentioned PLU decomposition method. We notice that the circuit provided in [25] for computing  $c_i$ 's is incorrect. We correct the circuit by applying appropriate CNOTs and without incurring additional Toffolis, as shown in Fig. 7, and subsequently, roughly halving its CNOT count. See Step 4 in Appendix B for details. Note that this circuit requires both CNOTs and Toffolis, and it is much smaller compared to the previous parts of the modular multiplication algorithm.

The modular inversion operation computes the inverse of a given polynomial  $f(x)$  modulo  $p(x)$ , denoted as

$$f^{-1}(x) \bmod p(x). \quad (14)$$

Using an extension of Fermat's little theorem (FLT) over binary fields, the modular inverse can be equivalently obtained by computing [50]

$$f(x)^{2^n-2} = f^{-1}(x) \bmod p(x). \quad (15)$$

Moreover,  $f(x)^{2^n-2}$  can be computed using a sequence of squaring and modular multiplication operations on appropriate powers of  $f(x)$ . A quantum circuit implementation of this FLT-based inversion method was first described in [58]. This FLT-based approach was compared to a greatest common divisor (GCD)-based approach in [16], which found that

the former has a lower Toffoli count while the latter has a lower qubit count. To address this tradeoff, we make use of the FLT-based algorithm introduced in [26], which improves on previous FLT-based algorithms to reduce the number of ancilla qubits. For a comparison, see Appendix C for a discussion about a more Toffoli-optimal FLT-based algorithm from [24]. This algorithm is based on the observation that in FLT-based algorithms, when computing  $f(x)^{2^n-2}$ , the exponents of  $f(x)$  form an addition chain<sup>1</sup> for  $2^n - 2$ . These addition chains correspond to distinct sequences of squaring and modular multiplication operations. Furthermore, one can find alternative addition chains and thus sequences of squaring and modular multiplication that optimize the number of intermediate terms, i.e., powers of  $f(x)$ , which can be uncomputed to  $|0 \dots 0\rangle$  ancilla qubits that are reused throughout the algorithm, thereby reducing the number of ancilla qubits required. However, this spatial optimization comes at the cost of an increase in the number of squaring and modular multiplication operations. Notably, compared to the GCD-based method in [16] over relevant field sizes, the Toffoli counts remain much lower and the qubit counts are competitive.

For modular multiplication, we use the CRT-based method described earlier. The modular squaring of a polynomial  $f(x)$ , i.e.,  $f(x)^2 \bmod p(x)$ , can be formulated as a multiplication between a matrix  $S$  on the vector coefficients of  $f(x)$ , i.e.,  $|f\rangle \mapsto |Sf\rangle$ . In particular,  $S$  combines two actions: 1) it maps the monomials  $f_i x^i$  to  $f_i x^{2i}$ , and 2) it performs modular reduction with respect to  $p(x)$ ; see Appendix A-5. In the quantum circuit, this combined operation can be realized as a sequence of swap and CNOT gates, using the previously mentioned PLU decomposition method [10].

#### IV. RESOURCE ESTIMATION

In this section, we report our resource estimation methodologies and the resulting estimates of our algorithm applied to binary ECC of cryptographically relevant field sizes:  $n \in \{163, 233, 283, 571\}$  [55]. We begin by describing the quantum circuits at the logical level, accounting for both Clifford and non-Clifford gates; both types of gates are necessary to estimate the AV of the algorithm [20], whereas only the non-Clifford gate count is needed to estimate the circuit volume of the algorithm [20], [33]. From there, we estimate the hardware footprint and runtime on baseline and AV architectures.

##### A. ESTIMATING GATE AND QUBIT COUNTS

First, we estimate the gate and qubit counts of the three main circuit components in our algorithm, namely, the ECPPOINTADDCircuit, QROM look-up, and its uncomputation, disregarding the comparatively negligible cost of quantum Fourier transform as done in [16], [20], [24], and [26].

Starting with the nonarithmetic components in ECPPOINTADD, we implement each  $n$ -qubit equality check as an

<sup>1</sup>An addition chain for a nonnegative integer  $n - 1$  is a sequence  $\alpha_0 = 1, \alpha_1, \alpha_2, \dots, \alpha_l = n - 1$ , with the property that each  $\alpha_i$ , after  $\alpha_0$ , is obtained by adding two earlier terms that are not necessarily distinct.

**TABLE 2. Costs of the CRT-Based Modular Multiplication Algorithm Stated in Terms of CNOTs, Swaps, Toffolis, and AV**

$n$	# CNOTs	# Swaps	# Toffolis	Active Volume
163	110 956	300	999	$4.91 \times 10^5$
233	225 402	448	1448	$9.70 \times 10^5$
283	325 206	618	1776	$1.38 \times 10^6$
571	1 287 610	2208	3860	$5.33 \times 10^6$

Note that  $2n$  input qubits store polynomials  $f(x)$  and  $g(x)$ , and  $n$  output qubits store the result  $h(x) + f(x) \cdot g(x) \bmod p(x)$ .

**TABLE 3. Costs of Computing  $f^{-1}(x) \bmod p(x)$  Given  $f(x)$  via an FLT-Based Inversion algorithm [26] Stated in Terms of the Number of Modular Multiplication Applications, CNOTs, Swaps, and Toffolis, as Well as AV. Qubits Is  $5n$**

$n$	# ModMults	# CNOTs	# Swaps	# Toffolis	Active Volume
163	14	1 651 326	14 765	13 986	$7.26 \times 10^6$
233	16	3 761 228	55 298	23 168	$1.61 \times 10^7$
283	18	6 254 129	47 997	31 968	$2.65 \times 10^7$
571	20	27 646 645	134 422	77 200	$1.14 \times 10^8$

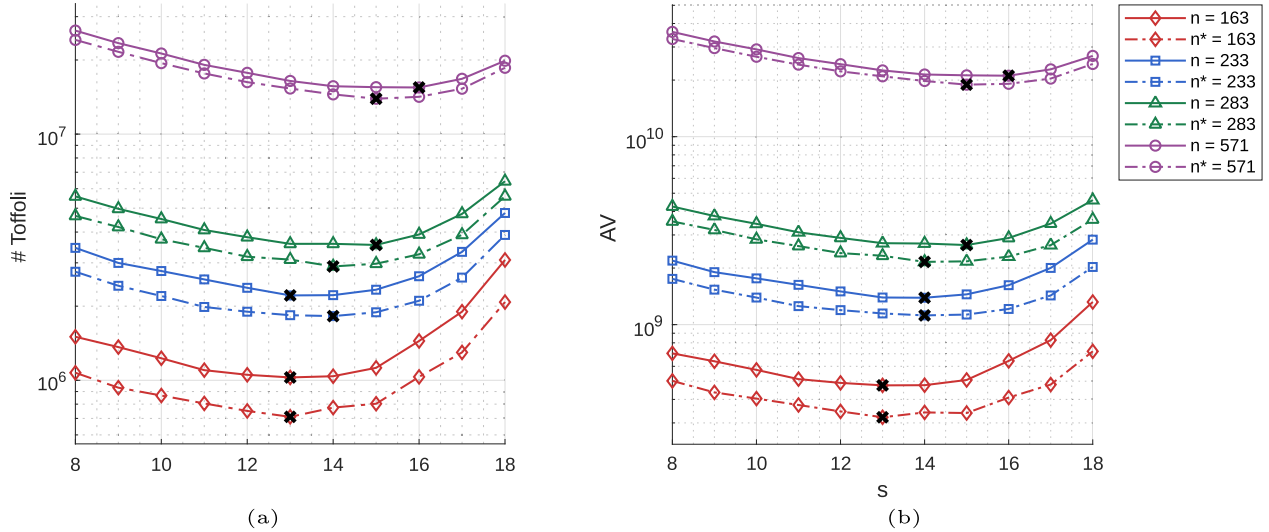
$2n$  qubits are used to store the input and output, and for the field sizes  $n$  stated here, the number of ancilla.

**TABLE 4. Costs of an ECPPOINTADDCircuit**

$n$	# Toffolis	# Qubits	Active Volume
163	$7.13 \times 10^4$	1963	$3.33 \times 10^7$
233	$1.15 \times 10^5$	2803	$7.29 \times 10^7$
283	$1.55 \times 10^5$	3403	$1.18 \times 10^8$
571	$3.65 \times 10^5$	6859	$5.01 \times 10^8$

$n$ -qubit Toffoli conjugated by  $n$  pairs of bitwise CNOT gates, and each  $n$ -qubit Toffoli is decomposed into  $n - 1$  regular Toffoli gates and  $n - 1$  clean ancilla qubits [59]. Next, we summarize the arithmetic circuits, which are explained in Section III-C and Appendix A (controlled) modular addition is implemented by  $n$  bitwise (controlled) CNOTs applied to the addends. A controlled modular constant-addition, which is only used once per ECPPOINTADDCircuit, can be implemented using at most  $n$  CNOTs, which we round up to  $n$  CNOTs in our resource estimates. The CRT-based modular multiplication is implemented using circuits that are constructed from pre-computed polynomials  $m_i(x)$ 's and  $p(x)$ , the Karatsuba-like multiplication circuits for low-degree polynomials from [25], and the correction circuit in Fig. 7. In particular, we generate the CNOT circuits for modular reductions from  $m_i(x)$ 's, and the circuits, consisting of only swap and CNOT gates, for modular multiplication with precomputed constant polynomials, used in (11) and (13), from  $m_i(x)$ 's and  $p(x)$ . Note that the Karatsuba-like multiplication circuit incurs the majority of the Toffoli gates in the modular multiplication algorithm, with a minor contribution from the correction circuit if it is applied. The FLT-based modular inversion circuit comprises repeated calls to modular multiplication and squaring, which is a circuit built from CNOTs and swaps. We provide *exact* gate and qubit counts—computed from the classical inputs of the arithmetic circuits—for an application of modular multiplication in Table 2 and those for an application of modular inversion in Table 3. The resource estimates of a fully compiled ECPPOINTADDCircuit are listed in Table 4.

We choose to use the QROM implementation from [60] and its clean-ancilla-assisted uncomputation from [61] due to



**FIGURE 5.** Cost landscapes of a single round of phase estimation, i.e., half of the circuit in Fig. 1, plotted against window size  $s$ . The optimal window sizes with respect to Toffoli count and AV are marked by crosses in (a) and (b), respectively. (a) and (b) share the same legend, in which the asterisks mark the cases with 48 classically precomputed bits in the private key. (a) Cost landscape with respect to Toffoli count. (b) Cost landscape with respect to AV.

**TABLE 5.** Costs of the Phase Estimation Circuit in Fig. 1 Implemented Using the Windowing Circuit in Fig. 2

$n$	$s$	# Toffolis	Qubits	Active Volume
163	13	$2.05 \times 10^6$	2126	$9.50 \times 10^8$
233	13/14	$4.42 \times 10^6$	3036	$2.78 \times 10^9$
283	15	$7.09 \times 10^6$	3686	$5.30 \times 10^9$
571	16	$3.09 \times 10^7$	7430	$4.22 \times 10^{10}$

We optimize the window size  $s$  for the number of Toffolis and AV separately. Note that the optimal  $s$ -values for Toffolis and AV are not necessarily the same but incidentally, they are the same for all considered field sizes except  $n = 233$ , for which the optimal  $s$ -values for Toffolis and AV are 13 and 14, respectively.

their low Toffoli counts; a  $k$ -item QROM look-up costs  $k - 2$  Toffoli gates and the uncomputation circuit costs approximately  $2\sqrt{k}$  Toffoli gates. Then, the total Toffoli count of the phase estimation circuit in Fig. 1 is two times the following:

$$\left\lfloor \frac{n}{s} \right\rfloor \left( 2^s - 2 + \mathcal{C}(\text{ECPPOINTADD}) + 2^{s/2+1} \right) + \left( 2^{n \bmod s} - 2 + \mathcal{C}(\text{ECPPOINTADD}) + 2^{(n \bmod s)/2+1} \right) \quad (16)$$

where  $\mathcal{C}(\text{ECPPOINTADD})$  is the cost of ECPPOINTADD and  $s$  is the window size of the circuit in Fig. 2. We minimize the Toffoli count over  $s$  and list the optimized Toffoli count and qubit count, including the ancilla qubits used to synthesize the multiply controlled Toffolis, in Table 5. We plot the Toffoli count at various  $s$ -values in Fig. 5(a). Note that our Toffoli counts are lower than those in [26], where the same arithmetic routines are used.

As in [21], we also consider the scenario where 48 bits of the private key are determined on a classical computer using algorithms in [62], which require at most  $2^{25}$  point addition operations. According to [63], a million-instruction-per-second, i.e., MHz, CPU can perform 40 000 point additions per second. Therefore, computing the 48 bits will take about

**TABLE 6.** Costs of the Phase Estimation Circuit in Fig. 1 Implemented Using the Windowing Circuit in Fig. 2 and Assuming 48 Classically Precomputed Bits

$n$	$s$	# Toffolis	Qubits	Active Volume
163	13	$1.42 \times 10^6$	2126	$6.43 \times 10^8$
233	14	$3.65 \times 10^6$	3036	$2.24 \times 10^9$
283	14	$5.80 \times 10^6$	3686	$4.31 \times 10^9$
571	15	$2.78 \times 10^7$	7430	$3.78 \times 10^{10}$

We optimize the window size  $s$  for the number of Toffolis and AV separately.

14 min on a MHz CPU, and a thousandth of that, i.e., less than a second, on a GHz CPU, which is negligible compared to the runtime on the quantum computer, as we discuss in the following. The Toffoli counts to compute the remaining  $n - 48$  bits in this case are listed in Table 6 and plotted in Fig. 5(a).

## B. ESTIMATING HARDWARE FOOTPRINT AND RUNTIME

In what follows, we delineate our methodology for translating the logical circuits discussed above to estimates of the hardware footprint and time required to execute our algorithm. We consider 1) a baseline architecture executed on a superconducting or trapped-ion quantum computer [33], and 2) an AV architecture [20] executed on a photonic FBQC [36]. These architectures implement surface codes [29], [30] and execute logical operations via lattice surgery [33], [64].

In both architectures, the footprint in space and time of a computation is determined by a quantity called *spacetime volume*, which in turn, depends on the number of logical qubits and logical cycles required to execute the computation. A logical cycle comprises  $d$  code cycles, where  $d$  is the code distance and a code cycle is the time needed to perform

a syndrome measurement. Since a lattice surgery operation is implemented in a logical cycle, we measure computational time in units of logical cycles. We now proceed to quantify the spacetime volume for both architectures, following the methods in [20] and [65].

In the baseline architecture, a logical circuit's spacetime volume is roughly twice its circuit volume  $n_Q \cdot n_T$ , where  $n_Q$  and  $n_T$  are the number of qubits and T gates, respectively, in the circuit. We take  $n_T$  to be four times the number of Toffolis because a Toffoli can be decomposed into four T gates [59]. Each T gate is implemented via multiqubit Pauli measurements between a number of qubits and a magic state. This architecture assumes roughly  $2n_Q$  qubits laid out on a 2-D grid:  $n_Q$  qubits, which we call memory qubits, are from the abstract logical circuit,  $n_W = n_Q$  qubits, which we call workspace qubits, are used to mediate Pauli measurements between memory qubits, and a much smaller group of qubits is reserved for distilling T gates to be consumed by the workspace qubits. It is further assumed that one T gate is produced per logical cycle, and that the T gate is consumed by workspace qubits while the next T gate is being produced. As a result, the total runtime is then proportional to  $n_T$ , and the spacetime volume is roughly given by

$$V_B = 2n_Q \cdot n_T. \quad (17)$$

In the AV architecture, the spacetime volume scales with the circuit's AV instead of circuit volume. AV measures the number of lattice surgery [64] operations used to execute a circuit, while leveraging the parallelism made possible by the fact that each surface-code qubit is connected to logarithmically many qubits [20]. There are a total of  $n_Q + n_W$  qubits, where  $n_Q$  and  $n_W$  are the number of memory and workspace qubits, respectively; both logical operations and magic-state distillation are performed using the workspace qubits. The set of logical, lattice surgery operations utilized in the AV architecture are known as logical blocks [20], [66]. The AV architecture further assumes the execution of  $n_W$  logical blocks per logical cycle. We count AV in the number of logical blocks. To estimate the AV of a circuit, one could express the circuit in terms of elementary operations, the AV of which are listed in [20, Table 1], and sum up the AV of the circuit's constituents. We have done that for the arithmetic circuits in our algorithm and listed their AV in Tables 2–5. For the nonarithmetic components, we take the AV estimates from [21, Fig. 3], except for the  $k$ -item QROM uncomputation circuit. Our QROM uncomputation circuit, taken from [61], has an AV of  $\approx 0.75k + 120\sqrt{k}$ , which is lower than that of the uncomputation circuit used in [21], because of the use of a Toffoli-free measurement uncomputation of a binary-to-unary circuit. Note that we minimize the window sizes  $s$  with respect to AV, separately from Toffoli count, and list the optimal  $s$ -values, with and without classical precomputed bits in the private key, in Tables 5 and 6, respectively. We plot the AV at various  $s$ -values in Fig. 5(a). A circuit with a total AV of  $b_{AV}$  has an estimated spacetime

**TABLE 7. Physical Resource Estimates for Baseline Architectures Executed on a Trapped-Ion or Neutral-Atom Device With a 1-ms Code Cycle and a Superconducting Device With a 1- $\mu$ s Code Cycle**

Field size $n$	163	233	283	571
Distance $d$	24	25	26	28
Device size	$2.45 \times 10^6$	$3.79 \times 10^6$	$4.98 \times 10^6$	$1.16 \times 10^7$
Runtime ( $1\mu\text{s/ms}$ cycle)	3.6 min / 2.5 days	8.2 min / 5.7 days	13.7 min / 9.5 days	64.1 min / 44.5 days
Runtime* ( $1\mu\text{s/ms}$ cycle)	2.5 min / 1.8 days	6.8 min / 4.7 days	11.2 min / 7.8 days	57.7 min / 40.0 days

The device size is measured in the number of physical qubits. In the asterisked row, we list the runtimes with 48 classically precomputed bits in the private key.

volume of

$$V_A = \frac{b_{AV}}{n_W} (n_Q + n_W). \quad (18)$$

Hereafter, we set  $n_W = n_Q$  and thus  $V_A = 2b_{AV}$ , as in [20] and [21]. Although, in general, the ratio between logical and workspace qubits can be adjusted to tune the computational performance, as demonstrated in [65].

Now, we proceed to calculate the physical resources, i.e., hardware footprint and runtime, it takes to execute a circuit. We first consider the baseline architecture on a matter-based device. The execution time of a circuit can be estimated as

$$\begin{aligned} T &= \text{number of code cycles} \times \text{code cycle time} \\ &= d \times \text{number of logical cycles} \times \text{code cycle time} \end{aligned} \quad (19)$$

where  $d$  is the code distance. We assume a logical error rate per unit of spacetime volume to be  $p_L = 10^{-d/2}$ , corresponding to 10% of the surface code threshold [21], [66], and a 0.05 failure probability for a single computation, as in [21]. Then,  $d$  can be obtained by solving

$$p_L \cdot V_B \leq 0.05. \quad (20)$$

Note that this failure probability can also be construed as a failure probability  $\leq 0.5$  after ten executions of the algorithm, i.e., one in ten executions is faulty. So, the average runtime of a successful computation will be  $10T/9$ . The number of logical cycles is given by  $n_T$ , and the code cycle time depends on the hardware. We assume a hardware-motivated 1  $\mu$ s and 1 ms code cycle time for a hypothetical superconducting and trapped-ion (or neutral-atom) device, respectively, in line with [18], [20], [21], [67], [68], and [69]. The number of physical qubits can be estimated as  $2n_Q \cdot d^2$ , where  $d^2$  is the number of physical qubits per surface-code logical qubit. We list these physical resource estimates, including the number of physical qubits and average runtime, for our algorithm, with and without classically precomputed bits in the private key, in Table 7.

We move onto the AV architecture on a photonic FBQC, where computation is carried out by performing multiphoton entangling measurements called fusions on photonic entangled resource states [36]. We measure the physical footprint of the computer in the number of physical units called *interleaving modules* (IMs) [20], [34]. Each IM comprises a collection of RSGs and delay lines that store and route the generated resource states to fusion locations [20], [34]. The

**TABLE 8. Physical Resource Estimates for Active Architectures Executed on a Photonic FBQC With 1  $\mu$ s and 10  $\mu$ s Delay**

Field size $n$	163	233	283	571
Distance $d$	22	23	23	25
Device size $(\frac{1}{10\mu s})_{\text{delay}}$	2058/ 206	3213/ 322	3900/ 390	9288/ 929
Runtime $(\frac{1}{10\mu s})_{\text{delay}}$	10.9 s/ 1.8 min	23.4 s/ 3.9 min	36.7 s/ 6.1 min	2.6 min/ 26.3 min
Distance $d^*$	21	22	23	25
Device size* $(\frac{1}{10\mu s})_{\text{delay}}$	1876/ 188	2939/ 294	3900/ 390	9288/ 929
Runtime* $(\frac{1}{10\mu s})_{\text{delay}}$	7.0 s/ 1.2 min	18.0 s/ 3.0 min	29.9 s/ 4.9 min	2.4 min/ 23.5 min

The device size is measured in the number of IMs with a 1 GHz total resource state generation rate. In the asterisked rows, we list the resource estimates with 48 classically precomputed bits in the private key.

execution time  $\mathcal{T}$  of a circuit is given by

$$\mathcal{T} = \frac{V_A d^3}{n_{\text{IM}} r_{\text{IM}}} \quad (21)$$

where the numerator is the spacetime volume given in the units of resource states with  $d^3$  being the number of resource state per logical block,  $n_{\text{IM}}$  is the number of IMs, and  $r_{\text{IM}}$  is the total resource state generation rate per IM, i.e., the sum of the rate of its constituent RSGs. We compute  $d$  by substituting  $V_B$  for  $V_A$  in (20) and then solving it. As done for the baseline architecture, we take  $10\mathcal{T}/9$  to be the average runtime of a successful computation. Moreover, we take  $r_{\text{IM}}$  to be 1 GHz as in [20] and [65]; note that  $r_{\text{IM}}$  is unrelated to the physical clock rate, since an IM with  $x$  RSGs at a rate  $y$  has the same  $r_{\text{IM}} = xy$  as one with  $a \cdot x$  RSGs at a rate  $y/a$ .  $n_{\text{IM}}$  can be computed as follows:

$$n_{\text{IM}} = \frac{nd^2}{n_{\text{RS/IM}}} \quad (22)$$

where  $nd^2$  is the number of resource states needed to construct  $n = n_Q + n_W$  logical qubits, assuming an architecture based on six-ring resource states [36], and  $n_{\text{RS/IM}}$  is the number of resource states stored in an IM at any given time, defined by

$$n_{\text{RS/IM}} = r_{\text{IM}} \cdot \frac{l_{\text{delay}}}{c_{\text{fiber}}} \quad (23)$$

where  $l_{\text{delay}}$  is the length of a fiber optic delay line and  $c_{\text{fiber}} = 2 \times 10^8 \text{ m/s}$  is the speed of light in a fiber optic cable. Combining (22) and (23), we obtain

$$n_{\text{IM}} = \frac{nd^2 c_{\text{fiber}}}{r_{\text{IM}} l_{\text{delay}}} \quad (24)$$

We list these physical resource estimates, including the number of IMs and average runtime, for our algorithm, with and without classically precomputed bits in the private key, in Table 8.

## V. CONCLUSION

We estimated both the logical and physical resource costs of computing binary elliptic curve discrete logarithms on an FTQC. In addition, we constructed the first, to the best of the authors' knowledge, quantum circuit that implements the point addition operation exactly, including all exceptional

cases. We also corrected and optimized the quantum circuit implementation [25] of the CRT-based modular multiplication algorithm [51], [52]. Compared to prior art [26] that uses the same arithmetic algorithms as we do, albeit with the uncorrected modular multiplication circuit, our algorithm incurs a lower Toffoli count over cryptographically relevant binary field sizes, due to the use of a more efficient QROM uncomputation circuit.

We carried out the physical resource estimation, i.e., hardware footprint and runtime, for a baseline architecture executed on hypothetical superconducting, trapped-ion, and neutral-atom quantum computers, and for the AV architecture executed on a hypothetical photonic fusion-based quantum computer. Comparing to the superconducting baseline architecture, the photonic AV architecture executes our algorithm over 20 times faster assuming a 1- $\mu$ s delay and two times faster assuming a 10- $\mu$ s delay, over cryptographically relevant field sizes. The speed-ups can be understood from perspectives discussed in [20] as follows.

- 1) In a baseline architecture, the spacetime cost per Toffoli is proportional to the number of logical qubits, while it is independent of the number of logical qubits in the AV architecture.
- 2) The reduced code distance on an AV device offers another source of speed-up.
- 3) The length of delay acts as a tradeoff parameter between speed and hardware footprint, i.e., number of resource state generators. On a trapped-ion or neutral-atom device, the speed-ups are magnified by a factor of 1000 due to its slower logical clock-speed.

Compared to the algorithms for computing prime elliptic curve discrete logarithms [21] and factoring a 2048-bit RSA integer [18], our algorithm has a much lower Toffoli count—one to two orders of magnitude depending on field size—and a faster runtime on both baseline and AV architectures [20]. The hardware footprint and qubit count of our algorithm are similar to those of the prime-curve algorithm for comparable field sizes and are smaller than those of the 2048-RSA algorithm.

However, the AV-to-Toffoli-count ratio of our algorithm is much higher compared to that of the prime ECC algorithm in [21], which leads to a relatively smaller speed-up from using the AV architecture. This is due to the large number of CNOTs in the modular multiplication circuit. Compared to a close alternative—the Karatsuba multiplication circuit in [14]—our chosen circuit's Toffoli count is about an order of magnitude smaller, but its CNOT count is higher by about 2 – 4 times over the considered field sizes; while our chosen circuit still has a lower AV, there could be more AV-optimal multiplication circuits, which we leave for investigation in future work. An alternative way to improve our AV estimates is via peephole optimization. Instead of simply adding up the AV in a gate-by-gate manner, since AV is subadditive [20], we could optimize the AV of subcircuits, which consist of a collection of gates, before adding them up. For example, the

modular multiplication circuit comprises many purely CNOT subcircuits, over which one could perform gate optimization using methods in, e.g., [70], [71], [72], and [73], which in turn reduces AV, or perform direct AV optimization using tools, such as ZX calculus [20]. Such optimizations are better suited to be carried out via software. We leave such explorations for future work.

In addition, we have left out a couple of potential optimizations. 1) The modular division operation can be parallelized over multiple instances of the algorithm [21]. 2) By tuning the ratio between workspace and memory qubits, the performance of both architectures can be enhanced; see, e.g., [18], [65], and [74]. We further provide some interesting directions to be pursued in the future. 1) Perform a similar study of the recently invented Regev's algorithm [75], [76] applied to binary elliptic curve discrete logarithms, and compare it with our algorithm. 2) Extend Karatsuba-like formulas [53], [54] to ten-way splits. Then, we will not have to recursively call CRT-based modular multiplication for larger field sizes, which in turn, will lead to lower gate counts and AV.

## APPENDIX

### DETAILS ON ARITHMETIC SUBROUTINES

In this section, we present the essential subroutines and circuits required for the elliptic curve point addition circuit in Section III. These subroutines can be constructed from Toffoli, CNOT, and swap gates. The structure of this section is as follows. Appendix A provides Toffoli-free modular arithmetic circuits, Appendix B describes how to perform modular multiplication (the primary contributor to the Toffoli gate count), and Appendix C describes the modular inversion subroutine.

#### A. TOFFOLI-FREE ARITHMETIC

In this section, we summarize arithmetic operations over binary fields that do not require Toffoli gates. We begin by defining basic notation and then describe several subroutines necessary for modular multiplication and inversion algorithms. These subroutines are categorized into two types: out-of-place and in-place algorithms. The subroutine circuits are determined by classical inputs, with binary addition being the exception. Out-of-place subroutines require a classical matrix as input. In-place subroutines involve additional classical preprocessing, specifically a PLU-decomposition, where the resulting matrices determine the circuit construction. This section is based on prior work, with some of the described subroutines detailed in [10], [16], and [25].

We use a polynomial basis representation, where  $\mathbb{F}_{2^n}$  is identified with  $\mathbb{F}_2[x]/p(x)$ , where in this section, we will use  $p(x) \in \mathbb{F}_2[x]$  to denote an irreducible polynomial of degree  $n$ . The elements in  $\mathbb{F}_{2^n}$  are then of the form

$$f(x) = \sum_{i=0}^{n-1} f_i x^i \quad (25)$$

**TABLE 9.** Irreducible Polynomials of Degree  $n$  Used in This Work, Taken From [55]

$n$	Irreducible polynomial
163	$x^{163} + x^7 + x^6 + x^3 + 1$
233	$x^{233} + x^{74} + 1$
283	$x^{283} + x^{12} + x^7 + x^5 + 1$
571	$x^{571} + x^{10} + x^5 + x^2 + 1$

where  $f_i \in \mathbb{F}_2$ . Addition and multiplication are defined modulo an irreducible polynomial  $p(x)$  of degree  $n$ . The irreducible polynomials  $p(x)$  that are used in this work can be found in Table 9. Using one qubit per coefficient of  $f(x)$ , we encode  $f(x)$  as an  $n$ -qubit quantum state  $|f_0\rangle|f_1\rangle \dots |f_{n-1}\rangle$ , which we collectively denote as  $|f\rangle$ ; depending on the context, e.g., when referring to a subset of the  $n$  qubits, we may make the subindices explicit, i.e.,  $|f_{i_1, \dots, i_j}\rangle$ , where  $0 \leq i < j \leq n - 1$ .

#### 1) OUT-OF-PLACE MULTIPLICATION

In this section, we describe how to perform out-of-place multiplication. This operation involves multiplying  $g(x) \in \mathbb{F}_{2^n}$  by a fixed nonzero polynomial  $h(x) \in \mathbb{F}_{2^n}^*$ , with the result reduced modulo an irreducible polynomial  $p(x)$  of degree  $n$ . Since multiplication by a constant nonzero polynomial of  $\mathbb{F}_{2^n}$  is  $\mathbb{F}_2$ -linear, this operation can be represented as a matrix–vector multiplication with a suitable  $n \times n$  matrix  $M$  [10], [25], where, hereafter, the matrix and vector components are binary, and any operations over them are over  $\mathbb{F}_2$ . Explicitly,  $M$  encodes the multiplication by  $h(x) \bmod p(x)$ , where the  $k$ th column of  $M$  corresponds to the coefficients of  $x^k \cdot h(x) \bmod p(x)$ . The matrix  $M$  acts on an  $n$ -dimensional column vector containing the coefficients of  $g(x)$ . This operation can be interpreted as adding coefficients of  $g(x)$ , conditioned on the elements of  $M$ .

To implement this in a quantum circuit, we start with the  $n$ -qubit input state  $|g\rangle$ , storing the coefficients of the polynomial  $g(x)$ , and an  $n$ -qubit state  $|f\rangle$ , initialized to store the coefficients of an arbitrary polynomial  $f(x)$ , where the result  $g(x)h(x) \bmod p(x)$  will be the output. Having expressed modular multiplication by a fixed polynomial  $h(x)$  as a matrix–vector multiplication, we can realize this in a quantum circuit by applying CNOT gates, conditioned on the elements of the matrix  $M$ . Specifically, each 1 in the matrix  $M$  corresponds to applying a CNOT gate, where the column index specifies a control qubit of  $|g\rangle$  and the row index specifies a target qubit of  $|f\rangle$ . The quantum circuit maps the input state  $|g\rangle|f\rangle$  to the state  $|g\rangle|f + g \cdot h\rangle$  and requires on average  $n^2/2$  CNOT gates [25]. However, in this work, we exactly count the number of CNOTs directly from  $M$ .

#### 2) MODULAR REDUCTION

To compute the modular reduction of an  $n - 1$  degree polynomial  $f(x)$  modulo a polynomial  $p'(x)$  of degree  $d \leq n - 1$ , we note that we can express  $f(x) \bmod p'(x)$  as the sum of two

polynomials

$$g(x) + (h(x) \bmod p'(x)) \quad (26)$$

where  $g(x)$  is the polynomial consisting of terms of  $f(x)$  of degree less than  $d$ , and the second polynomial represents the terms of  $f(x)$  with degree greater than or equal to  $d$ , reduced modulo  $p'(x)$ . Equation (26) helps clarify how to compute the result using a matrix–vector multiplication. Specifically,  $h(x) \bmod p'(x)$  can be evaluated by applying a  $d \times (n - d)$  matrix  $M$  to the column vector of coefficients of  $h(x)$ , where the  $k$ th column of  $M$  is given by  $x^{k+d} \bmod p'(x)$ . Consequently, (26) can be interpreted as adding coefficients of  $h(x)$ , conditioned on the elements of  $M$ , to the vector of containing coefficients of  $g(x)$ .

To implement this in a quantum circuit, consider the  $n$ -qubit input state  $|f\rangle = |g\rangle|h\rangle$ , where  $|g\rangle$  and  $|h\rangle$  store the coefficients of the polynomials  $g(x)$  and  $h(x)$  as in (26). Having expressed modular reduction as a matrix–vector multiplication, computing (26) can similarly be implemented as in the previous subroutine. Explicitly, by applying CNOT gates conditioned on the elements of the matrix  $M$ , with the control qubits in the  $|h\rangle$  register and the target qubits in the  $|g\rangle$  register. Similar ideas were first considered in [14], and the case when the degree of  $p'(x)$  can be smaller than  $n - 1$  is considered in [25].

### 3) IN-PLACE ADDITION

In-place binary addition, which adds  $f(x)$  to another polynomial  $g(x)$ , can be straightforwardly implemented as a quantum operation by applying  $n$  CNOT gates to add  $|f\rangle$  to an  $n$ -qubit input state  $|g\rangle$ . In particular, this is a coefficientwise XOR operation. The result of this addition, i.e.,  $|f + g\rangle$ , replaces one of the input states, either  $|f\rangle$  or  $|g\rangle$ , depending on the desired outcome.

### 4) IN-PLACE MULTIPLICATION

In this section, we describe how to perform in-place multiplication, which refers to multiplication by a constant polynomial. This is the same setup as out-of-place multiplication, where we represent multiplying  $g(x) \in \mathbb{F}_{2^n}$  by a fixed nonzero polynomial  $h(x) \in \mathbb{F}_{2^n}^*$ , modulo an irreducible polynomial  $p(x)$  of degree  $n$ , by matrix–vector multiplication with an  $n \times n$  matrix  $M$ . The  $k$ th column of  $M$  contains the coefficients of the polynomial  $x^k \cdot h(x) \bmod p(x)$ , with the coefficient of  $x^0$  appearing in the first row.

Following the work in [10] and [14],  $M$  can be converted into a quantum circuit via a PLU-decomposition. This decomposition expresses  $M$  as a product of three components: a permutation matrix  $P$ , a lower triangular matrix  $L$ , and an upper triangular matrix  $U$ , such that  $M = PLU$ . Such a decomposition allows for in-place multiplication, specifically the following holds.

- 1) The matrices  $U$  and  $L$  can be implemented as sequences of CNOT gates. Each off-diagonal element 1 in  $U$  and  $L$  represents an application of a CNOT gate,

where the column index indicates the control qubit and the row index indicates the target qubit.

- 2) The permutation matrix  $P$  can be implemented as a sequence of swap gates, where each off-diagonal element 1 in  $P$ , i.e.,  $P_{i,j} = 1$  (for  $i \neq j$ ), represents a swap gate between qubits  $i$  and  $j$ .

This in-place multiplication circuit requires at most  $n^2 - n$  CNOT gates and a number of swaps [14]. Note that in our resource estimation, we compute the exact CNOT and swap counts from  $M$ .

### 5) IN-PLACE SQUARING

In-place squaring, which involves squaring, modular reduction, and replacing the input, is a linear operation and can be expressed as an  $n \times n$  matrix [10], [16]. Squaring  $f(x)$  can be written as

$$\left( \sum_{i=0}^{n-1} f_i x^i \right)^2 = \sum_{i=0}^{n-1} f_i \cdot x^{2 \cdot i}. \quad (27)$$

This operation can be expressed as a  $(2n - 1) \times n$ -dimensional matrix acting on the coefficients of  $f(x)$ . To account for modular reduction, we can use a similar approach to that of the previous sections and represent it as an  $n \times (2n - 1)$  matrix. Multiplying these two matrices yields an  $n \times n$  that performs squaring and modular reduction. As in the in-place multiplication section, this  $n \times n$  matrix can be efficiently converted into a quantum circuit using a PLU-decomposition. The decomposition expresses the matrix as a product of a permutation matrix  $P$ , a lower triangular matrix  $L$ , and an upper triangular matrix  $U$ , which can be implemented using swap gates and sequences of CNOT gates. This implementation has the same cost as the in-place multiplication.

When we need to perform  $k$  consecutive squaring operations, we could naively perform each squaring operation separately; the cost scales linearly with  $k$ , requiring  $k$  times the cost of squaring and modular reduction. An alternative approach involves generating the matrix for squaring and modular reduction once, and then multiplying this matrix by itself  $k$  times, followed by a PLU-decomposition of the resulting matrix. The CNOT count of this approach does not grow with  $k$ . In practice, we will numerically evaluate both methods and choose the one with a smaller CNOT count.

## B. MODULAR MULTIPLICATION

For modular multiplication, we use the CRT-based modular multiplication algorithm from [25], with some modifications. This algorithm is based on well-established classical techniques, combining Karatsuba-like recurrence formulas from [53] and [54] and the CRT [51], [52]. Given two polynomials  $f(x)$  and  $g(x)$  in  $\mathbb{F}_{2^n}$ , the algorithm computes their product

$$f(x)g(x) \bmod p(x) \quad (28)$$

where  $p(x)$  is a degree- $n$  irreducible polynomial. It requires  $n$  qubits to store each of  $f(x)$  and  $g(x)$ , and  $n$  qubits to store the result  $h(x) + f(x) \cdot g(x)$ , where  $h(x) \in \mathbb{F}_{2^n}$ . In the following sections, we outline how the CRT can be used to perform modular multiplication. Next, we provide a step-by-step description of the algorithm, detailing how each step is implemented in a quantum circuit, with a necessary correction to the algorithm in the final step. In addition, we describe the optimizations to the algorithm that were applied, the chosen input parameters, and their impact on the resource estimation.

### 1) MODULAR MULTIPLICATION VIA CRT

Let  $f(x) = \sum_{i=0}^{n-1} f_i x^i$  and  $g(x) = \sum_{i=0}^{n-1} g_i x^i$  be two binary polynomials of degree  $n - 1$ . Suppose the aim is to compute their product

$$r(x) = f(x)g(x) = \sum_{l=0}^{2n-2} c_l x^l \quad (29)$$

where  $r_l = \sum_{i+j=l} f_i g_j$ , for  $0 \leq l \leq 2n - 2$ . Directly computing this high-degree multiplication can be costly in terms of space and the number of AND operations<sup>2</sup> required. To mitigate this, it is known that one can use a multiplication method based on the CRT for  $\mathbb{F}_2[x]$  [51], [52].

*Theorem 1 (See [52]):* Let  $m_1(x), m_2(x), \dots, m_t(x)$  be pairwise coprime polynomials and  $m(x) = \prod_{i=1}^t m_i(x)$ . Then, for any polynomials  $r_1(x), r_2(x), \dots, r_t(x)$ , there is a unique polynomial  $r(x) \bmod m(x)$ , with  $\deg(r(x)) < \deg(m(x))$  such that

$$r(x) = \sum_{i=1}^t r_i(x) q_i(x) \bmod m(x) \quad (30)$$

where  $r_i(x) = r(x) \bmod m_i(x)$  and

$$q_i(x) = \left( \frac{m(x)}{m_i(x)} \right) \left( \left( \frac{m(x)}{m_i(x)} \right)^{-1} \bmod m_i(x) \right). \quad (31)$$

Returning to the task of calculating the product in (29): by Theorem 1, to compute the product in (29), we can equivalently compute the sum in (30). Computing each term in the sum involves calculating the product  $r_i(x) = f(x)g(x) \bmod m_i(x)$ , a multiplication by a constant polynomial  $q_i(x)$ , and modular reduction by  $m(x)$ . Furthermore, the product  $r_i(x) = f(x)g(x) \bmod m_i(x)$  can be further reduced to calculating  $f_i(x)g_i(x) \bmod m_i(x)$ , where  $f_i = f(x) \bmod m_i(x)$  and  $g_i = g(x) \bmod m_i(x)$ . Therefore, we can see that we have reduced the problem of calculating the higher degree polynomials to that of calculating the product of lower degree polynomials  $f_i$  and  $g_i$ , both of degree less than  $d_i = \deg(m_i(x))$ . Furthermore, for certain values of  $d_i$ , such as ( $d_i \leq 8$ ), efficient circuits that implement generalizations of the Karatsuba algorithm [53], [54] can be applied to optimize the number

of AND operations, implemented as Toffoli gates, required for modular multiplication.

### 2) QUANTUM CIRCUIT FOR CRT-BASED MODULAR MULTIPLICATION

Next, we explain the quantum circuit implementation of the CRT-based modular multiplication algorithm in four steps. Our implementation is largely based on [25]; we will state explicitly where and why our implementation deviates. A high-level circuit diagram that implements the algorithm is provided in Fig. 6.

The algorithm calculates the product

$$c(x) = f(x)g(x) \bmod p(x) \quad (32)$$

where  $p(x)$  is an irreducible polynomial of degree  $n$ . In the quantum circuit implementation, the inputs and outputs of the algorithm are as follows. The input to the quantum circuit consists of three binary polynomials  $f(x), g(x), h(x) \in \mathbb{F}_{2^n}$ , stored in  $n$ -qubit states  $|f\rangle, |g\rangle$  and  $|h\rangle$ , respectively, where  $|h\rangle$  serves as the target register for the result of the multiplication. The quantum circuit maps the input state  $|f\rangle|g\rangle|h\rangle$  to the output state  $|f\rangle|g\rangle|h + fg \bmod p\rangle$ , where  $p$  represents the irreducible polynomial  $p(x)$  of degree  $n$  provided as part of the algorithm's fixed inputs. In addition to the input polynomials, the algorithm requires a set of pairwise coprime polynomials  $m_1(x), m_2(x), \dots, m_t(x)$ , which define the polynomial  $m(x)$  as their product

$$m(x) = m_1(x)m_2(x) \dots m_t(x). \quad (33)$$

Denoting  $d_i$  as the degree of each  $m_i(x)$ , and first computing the polynomials  $q_i(x)$  from each  $m_i(x)$  as described in (31), the steps of the algorithm are then as follows.

- *Step 1 (Residue Computations):* Calculate the residue representations of  $f(x)$  and  $g(x)$  with respect to each  $m_i(x)$

$$f^i(x) = f(x) \bmod m_i(x) \quad (34)$$

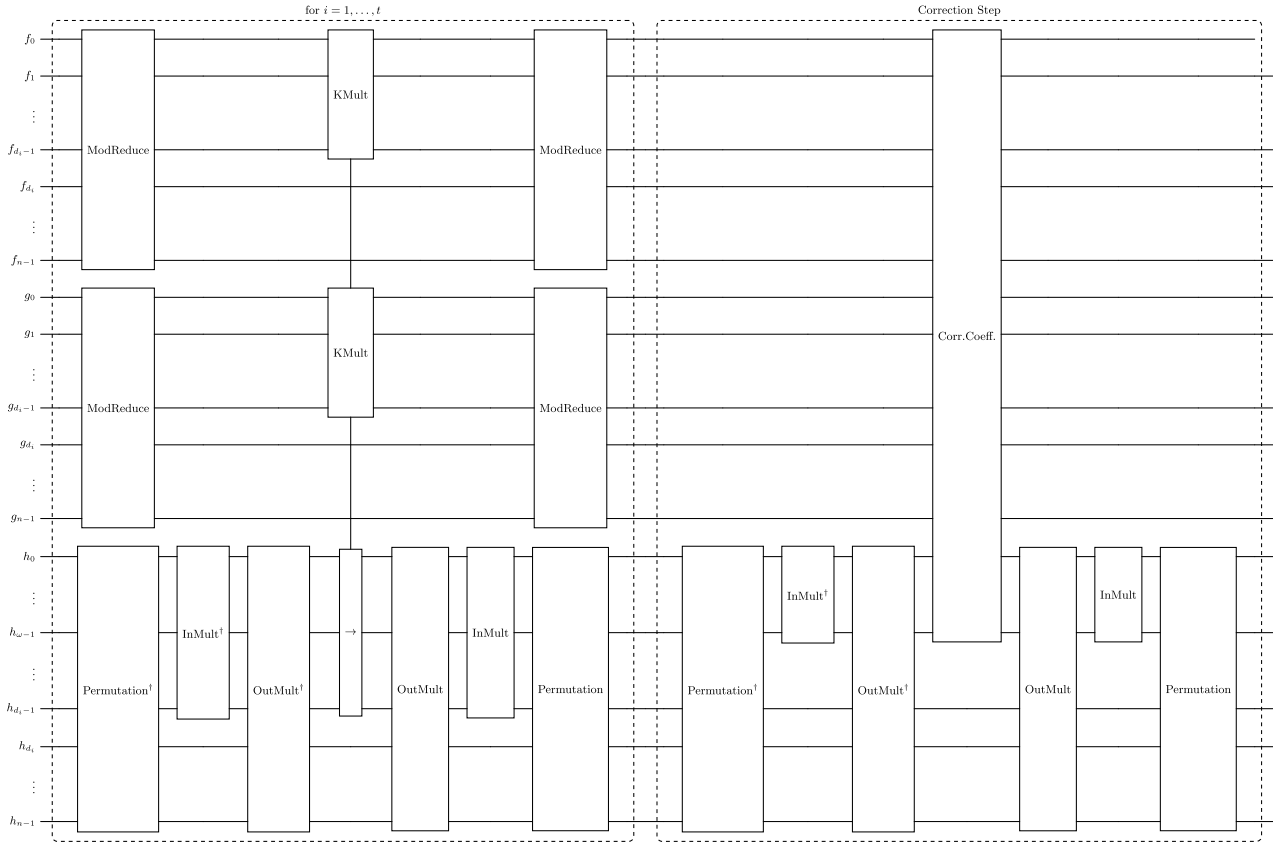
$$g^i(x) = g(x) \bmod m_i(x) \quad (35)$$

for  $i = 1, \dots, t$ .

In the quantum circuit, these residue computations are performed using the modular reduction subroutine, described in Section V-A, with CNOT gates determined by the matrix  $M_i$ . Here,  $M_i$  is a  $d_i \times (n - d_i)$  matrix constructed from the modulus polynomials  $m_i(x)$ , where each column  $k$  contains the coefficients of  $x^{k+d_i} \bmod m_i(x)$ , for  $k = 0, 1, \dots, n - 1 - d_i$ . As shown in circuit implementation in Fig. 6, for each  $i$ , the effects of this modular reduction must be uncomputed after Step 2. This is achieved by reapplying the same modular reduction subroutine with input  $M_i$ , using the fact that addition in  $\mathbb{F}_2$  is its own inverse.

As done in [25], we optimize this step by adding the matrices  $M_i$  (used for uncomputing the modular reduction for loop  $i$ ) and  $M_{i+1}$  (used for computing the modular reduction

<sup>2</sup>In quantum circuits, the AND operation is implemented using a Toffoli gate.



**FIGURE 6.** Quantum circuit overview for the CRT-based modular multiplication algorithm [25]. The circuit takes as fixed inputs the irreducible polynomial  $p(x)$  of degree  $n$  and the polynomials  $m(x) = \prod_{i=1}^t m_i(x)$ . It also takes as input two degree- $(n - 1)$  polynomials  $f(x)$  and  $g(x)$ , stored in separate  $n$ -qubit registers. In addition, a target register initially stores the polynomial  $h(x)$  and outputs  $h(x) + f(x)g(x) \bmod p(x)$ . The first dashed box corresponds to steps 1–3 of the algorithm for  $i = 1, \dots, t$ , while the second dashed box performs step 4 for  $\omega > 0$ . ModReduce denotes the modular reduction subroutine, OutMult represents the out-of-place multiplication subroutine, and InMult corresponds to the in-place multiplication subroutine. These subroutines are described in Section V-A, with the required classical input for each explained in Section V-B2. KMult represents modular multiplication, described in step 2, using generalized Karatsuba multiplication. However, if  $\deg(m_i(x)) > 8$ , then this modular multiplication is replaced by a recursive call to the CRT-based modular multiplication algorithm. Lastly, Corr.Coeff is the algorithm that computes the required correction coefficients, with the circuit shown in Fig. 7.

in the subsequent loop  $i + 1$ ). By appropriately padding these matrices with zeros (ensuring, in particular, that the column indices indicating the control qubits are correctly retained), they can be added into a single matrix  $M_{i,i+1}$ , which can be used as input to the modular reduction subroutine and therefore reduces the CNOT count.

- *Step 2 (Residue Products):* Compute modular polynomial products

$$c^i(x) = f^i(x)g^i(x) \pmod{m_i(x)} \quad (36)$$

for  $i = 1, \dots, t$ . To compute the above equation in a quantum circuit, there are two approaches. If the degree  $d_i$  is greater than 8, we recursively invoke the CRT-based modular multiplication algorithm. However, if the degree  $d_i$  of  $m_i$  is less than or equal to 8, then we use Karatsuba-like formulae [77], [78], i.e., generalized Karatsuba multiplication, to compute (36). This method reduces the multiplication complexity (defined as the

minimum number of multiplications needed to multiply two  $n$ -term polynomials) by recursively splitting a polynomial (in this case of degree  $d_i$ ) into smaller parts and combining their products more efficiently. Standard Karatsuba multiplication uses  $k = 2$ , while generalized Karatsuba multiplication applies  $k$ -way splits, further reducing the number of multiplications required. Moreover, it is possible to express generalized Karatsuba multiplication using a  $v \times n$  matrix  $T$  and an  $l \times v$  matrix  $R$ , where  $v$  is determined by the number of intermediate terms from the polynomial split and  $l$  is the number of terms in the output. The matrices  $T$  and  $R$  are precomputed matrices, depending on the Karatsuba split that is used, and act on the  $n$ -dimensional column vectors containing the coefficients of  $f(x)$  and  $g(x)$  [77], [78]

$$c = R \cdot [(T \cdot f) \circ (T \cdot g)] \quad (37)$$

where  $\cdot$  is the dot product and  $\circ$  is the Hadamard product. In this work, we use  $k$ -way splits, where  $k \in [3, 8]$ ,

and use the corresponding matrices  $T$  and  $R$ , as these values have previously been optimized for minimal gate count in circuits [53], [54]. Note that this is the step that incurs most of the Toffoli cost in the CRT-based modular multiplication algorithm. More specifically:

- 1) If  $d_i \leq 8$ : We invoke the generalized Karatsuba multiplication algorithm. This can be implemented via [25, Algorithm 1], which maps the input state

$$|f^i\rangle|g^j\rangle|h\rangle \mapsto |f^i\rangle|g^j\rangle|h + c^i\rangle \quad (38)$$

where  $c^i$  denotes the coefficients of the polynomial  $c^i(x)$ , of a smaller degree than  $h(x)$ , as defined in (36). The algorithm takes as input the precomputed matrices  $T_{d_i}$  and  $R'_{d_i}$ , where  $R'_{d_i}$  is the modularly reduced matrix<sup>3</sup> of  $R_{d_i}$ . To determine the gate count, the algorithm proceeds as follows for each row of  $T$ . For each input register  $|f^i\rangle$  and  $|g^j\rangle$ , first find the minimum column index  $i$ , where the entry of  $T$  is one. Then, for each column index greater than  $i$ , where the entry of  $T$  is one, a CNOT gate is applied, with the control qubit at index  $i$  and the target qubit at the column index. Next, for each column of  $R$ , identify the minimum row index  $i$ , where the entry of  $R$  is one. For each row index greater than  $i$ , where the entry of  $R$  is one, a CNOT gate is applied in the output register. In addition, for each row in the matrix  $T$ , a Toffoli gate is then applied with controls qubits in  $|f^i\rangle$ ,  $|g^j\rangle$  and target qubits in  $|h\rangle$ . Finally, for every CNOT gate applied for the matrices  $T$  and  $R$ , these must be uncomputed by running the sequence of CNOTs in reverse. This process is repeated for each row of  $T$ . We take the matrices  $T_{d_i}$  and  $R_{d_i}$  from the appendix of [25] and perform the modular reduction on  $R_{d_i}$  by ourselves. Note that  $T_{d_i}$  and  $R_{d_i}$  stem from [77] for  $d_i = 3$ , from [53] for  $d_i \in \{4, 5, 6\}$ , and from [54] for  $d_i \in \{7, 8\}$ . In this work, we use  $k \in [3, 8]$  as these values have previously been optimized for minimal gate count in classical circuits [53], [54].

- 2) If  $d_i > 8$ : To compute (38), we recursively call CRT-based modular multiplication algorithm with the following inputs: the polynomials  $f_i(x)$  and  $g_i(x)$  from Step 1 (each of degree less than  $d_i$ ), as well as the corresponding polynomial part of  $h(x)$ . That is, the terms of  $h(x)$  with degree less than  $d_i$ . All operations are performed modulo  $m_i(x)$ . Lastly, the input includes a polynomial  $m'(x)$ , analogous to  $m(x)$  in Theorem 1.

- *Step 3 (CRT Polynomial):* Compute the CRT-polynomial)

$$c'(x) = \sum_{i=1}^t (c^i(x)q_i(x) \bmod m(x)) \bmod p(x) \quad (39)$$

<sup>3</sup>That is,  $R'_{d_i}$  is obtained from a matrix that performs modular reduction, with respect to  $m_i(x)$ , acting on  $R_i$ .

where each quotient  $q_i(x)$  can be computed through (31).

To evaluate each term,  $(c^i(x)q_i(x) \bmod m(x)) \bmod p(x)$ , in the above sum, this multiplication can be expressed as a matrix–vector multiplication using an  $n \times d_i$  matrix  $Q_i$ , where each column  $k$  contains the coefficients of the polynomial  $(x^k q_i(x) \bmod m(x)) \bmod p(x)$ . Note that, the matrix  $Q_i$  is a nonsquare matrix, so we cannot directly apply the in-place multiplication subroutine.

Nevertheless, this operation can be implemented in a quantum circuit as follows. First, perform a PLU decomposition on the matrix  $Q_i$ . This decomposition expresses  $Q_i$  as the product of three matrices: an  $n \times n$  permutation matrix  $P_i$ , an  $n \times d_i$  matrix  $L_i$ , and a  $d_i \times d_i$  matrix  $U_i$ , such that  $Q_i = P_i L_i U_i$ . By multiplying  $L_i$  and  $U_i$ , we obtain an  $n \times d_i$  matrix  $LU_i$ . When then define two matrices:  $M_i$  consisting of the first  $d_i$  rows and  $d_i$  columns of  $LU_i$ , and  $N_i$ , consisting of the last  $n - d_i$  rows and  $d_i$  columns of  $LU_i$ . Now, to implement the operation

$$Q_i = P_i \begin{bmatrix} M_i \\ N_i \end{bmatrix} \quad (40)$$

in a quantum circuit, we proceed by first implementing the  $n - d_i \times d_i$  matrix  $N_i$  via the out-of-place multiplication subroutine. The inputs are the control qubits  $|h_{0,\dots,d_i-1}\rangle$  and the targets qubits  $|h_{d_i,\dots,n-1}\rangle$ . Each off-diagonal element in  $N_i$  corresponds to an application of a CNOT gate, where the column index indicates the control qubit and the row index indicates the target qubit. Next,  $M_i$  can be realized by the in-place multiplication subroutine. The input is the  $d_i$ -qubit state  $|h_{0,\dots,d_i-1}\rangle$  and the  $d_i \times d_i$  matrix  $M_i$  is implemented in the circuit via another PLU-decomposition. Lastly, the  $n \times n$  permutation matrix  $P_i$  can be implemented as a sequence of swap gates, where each off-diagonal element 1 in  $P$ , i.e.,  $P_{i,j} = 1$  (for  $i \neq j$ ), represents a swap gate between qubits  $i$  and  $j$ .

In addition, in the quantum circuit for CRT-based modular multiplication, an inverse operation must be applied to the target register prior to Step 2. This ensures that the multiplication  $|h + Q_i c^i\rangle$  is performed, rather than inadvertently applying multiplication of  $Q_i$  to any pre-existing values  $h$  in the target register. More specifically, suppose that the vector  $H$ , corresponding to the coefficients of a polynomial, is stored in the  $n$ -qubit target register. To implement Step 3 correctly, we must first apply the inverse of the circuit that implements the operation  $Q_i$ , followed by the circuit in Step 2 that adds  $K_i$ , and finally apply the circuit for  $Q_i$ . It can be verified that this sequence ensures the desired result  $H + Q_i K_i$ , where  $K_i$  is the vector with coefficients of (36).

At this point, we have explained how Steps 1–3 of the CRT-based multiplication algorithm can be vectorized and

implemented in a quantum circuit. We further note that in the circuit implementation, we repeat Steps 1–3 iteratively for  $i = 1, \dots, t$ , resulting in the polynomial  $c'(x)$  being stored in the  $n$ -qubit target register. For the final step of the algorithm, we must consider two cases depending on the degree of the modulus polynomial  $m(x)$ , due to the challenges in selecting pairwise coprime polynomials  $m_i(x)$  [51]. These challenges arise from the need to ensure that the product

$$m(x) = \prod_i m_i(x) \quad (41)$$

has a sufficiently large degree relative to  $n$ , while also minimizing the degrees  $d_i = \deg(m_i(x))$  so that the generalized Karatsuba algorithm is effectively employed. The simplest case to consider is when the modulus polynomial  $m(x)$  is chosen so that  $\deg(m(x)) > 2n - 2$  [51]. In this case, the algorithm has computed the desired product in (32), i.e.,

$$c(x) = (c'(x) \bmod m(x)) \bmod p(x) \quad (42)$$

and the algorithm terminates at this point. However, if  $\deg(m(x)) \leq 2n - 2$ , then the computed product  $c'(x)$  might not match the desired product  $c(x)$ . To see this, consider the case where  $\deg(m(x)) = 2n - 2$ . When multiplying two polynomials  $f(x)$  and  $g(x)$ , each of degree  $n - 1$ , their product can have a maximum degree of  $2n - 2$ . Therefore, after Step 3, the algorithm will produce the polynomial  $c'(x) = (f(x)g(x) \bmod m(x)) \bmod p(x)$  and would cause an undesired modular reduction of the term  $x^{2n-2}$ . Fortunately, we can recover the desired product  $c(x)$  by adding a term to  $c'(x)$  [51]

$$c(x) = c'(x) + (c_{2n-2}m(x)) \bmod p(x) \quad (43)$$

where  $c_{2n-2} = f_{n-1}g_{n-1}$  is referred to as a correction coefficient. This correction process can be generalized for cases where  $\deg(m(x)) < 2n - 2$ , which require multiple correction coefficients [52]. The number of required correction coefficients is given by

$$\omega = 2n - 1 - \deg(m(x)) \quad (44)$$

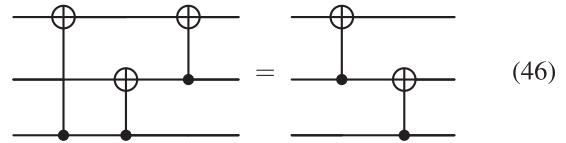
and each correction coefficient can be calculated by the following formula:

$$c_{2n-2-k} = \sum_{i=n-1-k}^{n-1} s_i + \sum_{\substack{i+j=2n-2-k, \\ n>i>j}} s_{i,j} \quad (45)$$

where  $k = 0, \dots, \omega - 1$ ,  $s_i = f_i g_i$ , and  $s_{i,j} = (f_i + f_j)(g_i + g_j)$ . For example, if the  $\deg(m(x)) = 2n - 3$ , then two correction coefficients,  $c_{2n-2}$  and  $c_{2n-3}$ , are required. Note that it is customary in the literature to indicate the use of a correction step by adding the symbolic term  $(x - \infty)$  to  $m(x)$ .

We present a corrected quantum algorithm for calculating the required correction coefficients in (45) for a specified  $\omega$ . Our algorithm for computing the correction coefficients is an in-place algorithm and works as follows: first, for all  $k$ , the algorithm computes the first sum in (45), ensuring that the bitstring of the target qubits is preserved. Then, for all  $k$ , it

proceeds to calculate the second sum, with details provided in the quantum circuit shown in Fig. 7. This algorithm is based on the approach in [25]. However, their circuit does not always preserve the existing bitstring in the target register. We have corrected their algorithm by adding the sequence of CNOTs preceding the first Toffoli gate and present an in-place circuit for calculating the correction coefficients. The algorithm requires  $\omega + (\omega^2/4)$  Toffoli gates and  $2(\omega - 1) + \omega^2$  CNOTs if  $\omega$  is even, and  $\omega + (\omega^2 - 1)/4$  Toffoli gates and  $2(\omega - 1) + \omega^2 - 1$  CNOTs if  $\omega$  is odd. In addition, we optimize our algorithm to reduce the CNOT count. Specifically, we minimize the CNOT gates required to calculate the second sum in (45): for each loop  $i$  in the algorithm that computes the second sum, the CNOT gates immediately preceding the Toffoli gates can be propagated forward into a layer of CNOT gates, while the CNOT gates immediately following the Toffoli gates can similarly be propagated backward. Then, by using the CNOT identity



we can remove redundant CNOT gates between the calculations for successive indices  $i$  and  $i + 1$ . The optimized circuit requires  $4(\omega - 1) + \omega^2/2$  CNOTs if  $\omega$  is even, and  $4(\omega - 1) + (\omega^2 - 1)/2$  CNOTs if  $\omega$  is odd.

- **Step 4 Correction:** If  $\deg(m(x)) \leq 2n - 2$ , then compute the remainder coefficients

$$c_{2n-2}, c_{2n-3}, \dots, c_{2n-1-\omega} \quad (47)$$

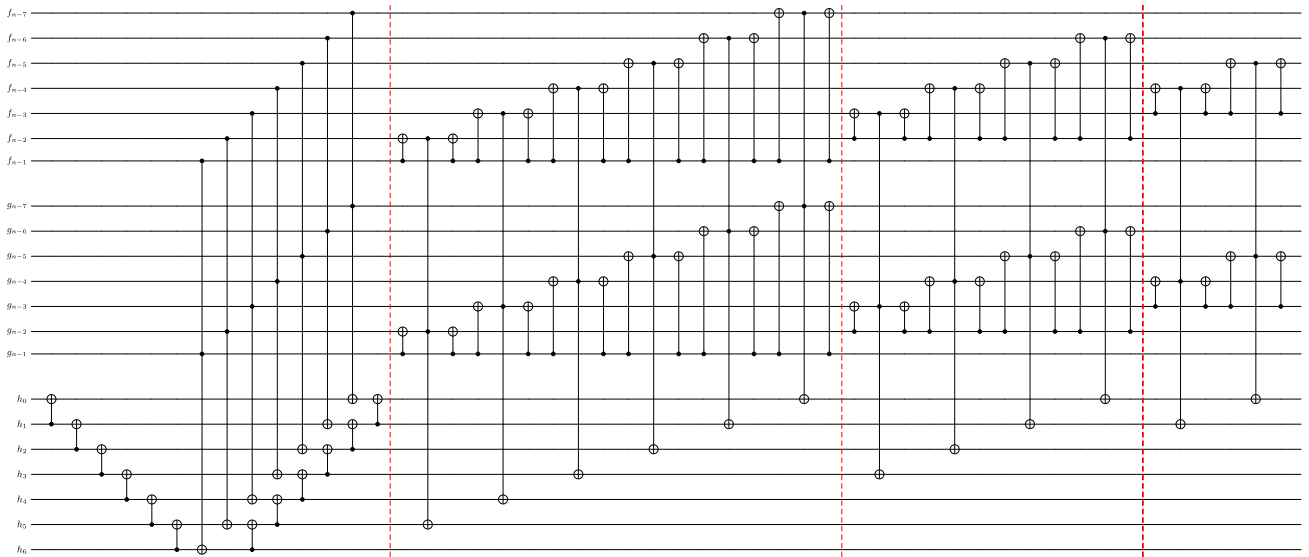
for  $\omega = 2n - 1 - \deg(m(x))$  and compute the final product  $c(x)$

$$c(x) = c'(x)$$

$$+ \sum_{i=2n-1-\omega}^{2n-2} c_i ((x^i) + (x^i \bmod m(x))) \bmod p(x). \quad (48)$$

The implementation of the correction step in the quantum circuit is analogous to Steps 2 and 3 of the algorithm. Specifically, the multiplication by the term  $((x^i) + (x^i \bmod m(x))) \bmod p(x)$ , in (48), can be represented as an  $n \times \omega$  matrix  $H_\infty$ . As in Steps 2 and 3, the quantum circuit that implements (48) is as follows. Run the circuit for  $H_\infty$  in reverse to undo its action on the target register. Compute the correction coefficients in (47) using the circuit in Fig. 7. Run the circuit that implements  $H_\infty$ .

The circuit that implements  $H_\infty$  is similarly implemented via PLU-decomposition. As in Step 3,  $H_\infty$  is decomposed via a PLU decomposition and then equivalently rewritten in terms of an  $n \times n$  permutation matrix  $P_\infty$ , a  $w \times w$  matrix  $M_\infty$ , and an  $(n - \omega) \times \omega$  matrix  $N_\infty$ . The matrix  $H_\infty$  can then be realized in the circuit by



**FIGURE 7.** Quantum circuit for calculating correction coefficients for  $\omega = 7$ . The inputs to the algorithm are two polynomials  $f$  and  $g$  of maximum degree  $n - 1$ , and a target register of size  $\omega$  storing arbitrary coefficients  $h_0, \dots, h_{\omega-1}$ . The output consists of the addition of correction coefficients  $c_{2n-1-\omega}, \dots, c_{2n-3}, c_{2n-2}$  stored in the target register. The circuit works as follows: first, for all  $k = 0, \dots, \omega - 1$ , it computes the first sum in (45). After the first vertical dashed line in the circuit, the algorithm proceeds to compute the second sum in (45). Specifically, for each  $k = 0, \dots, \omega - 1$ , the algorithm starts by computing terms in the sum corresponding to  $i = n - 1$  satisfying  $i + j = 2n - 2 - k$  (where  $j = n - k - 1$ ). After the next vertical line, where similarly for each  $k = 0, \dots, \omega - 1$  and  $i + j = 2n - 2 - k$ , the circuit then computes terms for  $i = n - 2$  (where  $j = n - k$ ), followed by  $i = n - 3$  (where  $j = n - k + 1$ ). This process is further repeated for subsequent indices  $i$  and  $j$  satisfying  $i + j = 2n - k - 2$  and  $n > i > j$ . The final output of the circuit consists of the coefficients  $h_0 + c_{2n-\omega-1}, \dots, h_{\omega-1} + c_{2n-2}$ .

an out-of-place multiplication (determined by  $N_\infty$ ), an in-place multiplication subroutine (determined by  $M_\infty$ ), and a sequence of swap gates determined by  $P_\infty$ .

We will now discuss the input choices for the quantum circuit and how they affect the resource estimates. The input to the CRT-based modular multiplication algorithm requires a modulus polynomial  $m(x)$ , which is the product of pairwise coprime polynomials  $m_i(x)$ . How this  $m(x)$  is chosen can affect the gate count and in particular the Toffoli count. Suppose the degree of  $m(x)$  is chosen such that  $\deg(m(x)) > 2n - 2$ . If  $n$  is large, e.g.,  $n = 283\,571$ , then attaining  $\deg(m(x)) > 2n - 2$  may require introducing many higher degree polynomials  $m_i(x)$  with degrees exceeding 8. This could lead to multiple recursive calls to the algorithm, which would incur significant gate costs, particularly in terms of Toffoli gates. On the other hand, if  $n$  is small, e.g.,  $n = 163\,233$ , then attaining  $\deg(m(x)) > 2n - 2$  could be achieved by adding polynomials  $m_i(x)$  with relatively small degrees. This approach would also incur gate costs resulting in calls to the generalized extended Karatsuba algorithm. For the  $k(\in [3, 8])$ -way Karatsuba split used in the algorithm, the exact gate cost can be found in [25]. Alternatively, suppose that the degree of  $m(x)$  is chosen such that  $\deg(m(x)) \leq 2n - 2$ . In this case, the extra cost comes from calls to the circuit for calculating correction coefficients, which becomes more expensive as the number of correction steps increases, leading to a higher count of Toffoli gates. Therefore, the choice of modulus polynomials impacts the overall Toffoli

gate count; there is a tradeoff between adding  $m_i(x)$  and the cost of computing correction coefficients.

Repeated calls to the CRT-based modular multiplication algorithm, for large  $n$ , are costly compared to computing a few correction coefficients. For smaller  $n$ , one could increase the number of  $m_i(x)$  polynomials to attain  $\deg(m(x)) > 2n - 2$ , resulting in additional calls to the generalized Karatsuba algorithm. While correction steps can reduce costs, requiring more correction terms also adds to the gate cost. Thus, there is a tradeoff in the Toffoli gate cost. A similar tradeoff in AV should apply as well. In practice, when choosing the polynomial  $m(x)$ , we have adopted a greedy and likely nonoptimal approach, increasing the number of polynomials  $m_i(x)$  until  $\deg(m(x)) > 2n - 2$ , while avoiding recursive calls to the algorithm in favor of correction steps where possible. This approach can be potentially be improved by choosing  $m(x)$  more optimally in both Toffoli count and AV, but we do not pursue it here as we anticipate the improvement to be minimal. Note that in the work of Kim et al. [25], for  $n = 283\,571$ , the degree of  $m(x)$  for the modulus polynomials listed in Table 10 does not satisfy the stated number of required correction coefficients. The polynomials  $m(x)$  we have used in this work are listed in Table 10.

### C. MODULAR INVERSION

The computation of modular inverses is the most resource-intensive arithmetic operation in elliptic curve point addition. In this work, we will compute inverses using algorithms based on FLT as FLT-based inversion algorithms have significantly lower Toffoli gate costs compared to algorithms

**TABLE 10. Modulus Polynomials Used for CRT-Based Modular Multiplication**

$n$	Modulus Polynomials
163	$x^8, (x + 1)^8, I_2^{1 \times 4}, I_3^{2 \times 2}, I_4^{3 \times 2}, I_5^{6 \times 1}, I_6^{9 \times 1}, I_7^{18 \times 1}, I_8^{7 \times 1}$
233	$x^6, (x + 1)^6, I_2^{1 \times 4}, I_3^{2 \times 2}, I_4^{3 \times 2}, I_5^{6 \times 1}, I_6^{9 \times 1}, I_7^{18 \times 1}, I_8^{25 \times 1}$
283	$x^7, (x + 1)^6, I_2^{1 \times 4}, I_3^{2 \times 2}, I_4^{3 \times 2}, I_5^{6 \times 1}, I_6^{9 \times 1}, I_7^{18 \times 1}, I_8^{30 \times 1}, I_9^{6 \times 1}$
571	$x^9, (x + 1)^8, I_2^{1 \times 4}, I_3^{2 \times 2}, I_4^{3 \times 2}, I_5^{6 \times 2}, I_6^{9 \times 1}, I_7^{18 \times 1}, I_8^{30 \times 1}, I_9^{56 \times 1}, I_{10}^{9 \times 1}$

Let  $f_{a,i}(x)$  denote the  $i$ -th irreducible polynomial of degree  $D$ , where  $i$  indexes the polynomials in arbitrary order. For integers  $A$  and  $B$ , we define the set  $I_{a^{A \times B}} = \{f_{a,i}(x)^b \mid i = 1, \dots, a\}$ . The number of required correction coefficients is given by  $\omega = 2n - 1 - \deg(m(x))$ . For  $n = 283$  and  $n = 571$ , we require  $\omega = 4$  and  $\omega = 6$  correction coefficients, respectively. For  $n = 163233$ , no correction step is required in the algorithm. Lists of irreducible polynomials are readily available online, e.g., see <https://mathworld.wolfram.com/IrreduciblePolynomial.html>.

using the extended GCD [16], [24], [26]. This method requires repeated use of the modular multiplication subroutine, which is costly in terms of Toffoli gates. However, this comes at the cost of needing more ancilla qubits, whereas GCD-based algorithms, while requiring fewer qubits overall, incur a much higher Toffoli gate count. In this work, we will use the inversion algorithm from [26] because it has the lowest qubit count—competitive to the GCD-based inversion algorithm—among FLT-based inversion algorithms. The input to the algorithm is the state  $|f\rangle|0\rangle^{R-n}$ , which is mapped to  $|f\rangle|f^{-1}\rangle|\text{garbage}\rangle|0\rangle^n$ , where  $|\text{garbage}\rangle$  is of size  $(R - 2)n$  qubits and  $R$  is a parameter dependent on the algorithm’s classical input. The  $n$ -qubit states  $|f\rangle$  and  $|f^{-1}\rangle$  encode the polynomials  $f(x)$  and  $f(x)^{-1}$ , respectively. In the following sections, we outline how the FLT theorem can be used to compute modular inverses. Then, we describe the quantum implementation. Lastly, we discuss the optimizations applied to the algorithm, the chosen input parameters, and their impact on resource estimation.

### 1) MODULAR INVERSION VIA FLT

Given a polynomial  $f(x) \in \mathbb{F}_{2^n}^*$ , computing its inverse  $f^{-1}(x) \in \mathbb{F}_{2^n}$  can be computed via FLT [50] as

$$f^{2^n-2} = f^{-1} \pmod{p(x)} \quad (49)$$

where  $n$  is the degree of the irreducible  $p(x)$ , and we use the notation  $f$  instead of  $f(x)$  for convenience. In addition, from here on, we will adopt the shorthand  $f^x := \langle x \rangle$ .

Equation (49) can be computed efficiently by the classical algorithm by Itoh and Tsujii [50]. This algorithm takes as input  $f^1 = \langle 2^0 - 1 \rangle$  and computes successive powers of  $f$  through repeated squaring and multiplications. Specifically, it calculates  $\langle 2^1 - 1 \rangle, \langle 2^2 - 1 \rangle, \dots, \langle 2^{k_1} - 1 \rangle$ , where  $k_1 = \lfloor \log(n - 1) \rfloor$ . Next, using the polynomials computed in this step, these intermediate results are combined to obtain  $\langle 2^{n-1} - 1 \rangle$ , which is then squared to produce the desired polynomial  $\langle 2^n - 2 \rangle = \langle -1 \rangle$ . A quantum circuit that implements this method for computing inverses was first described in [58] and requires  $k_1 + t - 1$  Toffoli gates and  $n \cdot \max(k_1 + t - 1, k_1 + 1)$  ancilla qubits., where  $t \leq k_1 + 1$  [16].

This method for computing inverses can be generalized to reduce the resource count—in some cases, reducing the Toffoli gate count [24]—or to reduce the ancilla qubits required [26]. The key observation is that the exponents of 2, calculated in the Itoh and Tsujii algorithm, can be observed to be a specific addition chain<sup>4</sup> for  $n - 1$ . To see this, consider an example for  $n = 163$ . Using the Itoh and Tsujii algorithm, we would compute  $\langle 2^i - 1 \rangle$  for  $i = 1, \dots, 7$ , and then combine the previous polynomials to obtain  $\langle 2^{2^7+2^5} - 1 \rangle$  and  $\langle 2^{2^7+2^5+2^1} - 1 \rangle = \langle 2^{n-1} - 1 \rangle$ . By expressing the exponents of 2 as an addition chain for  $n - 1 = 162$ , we have

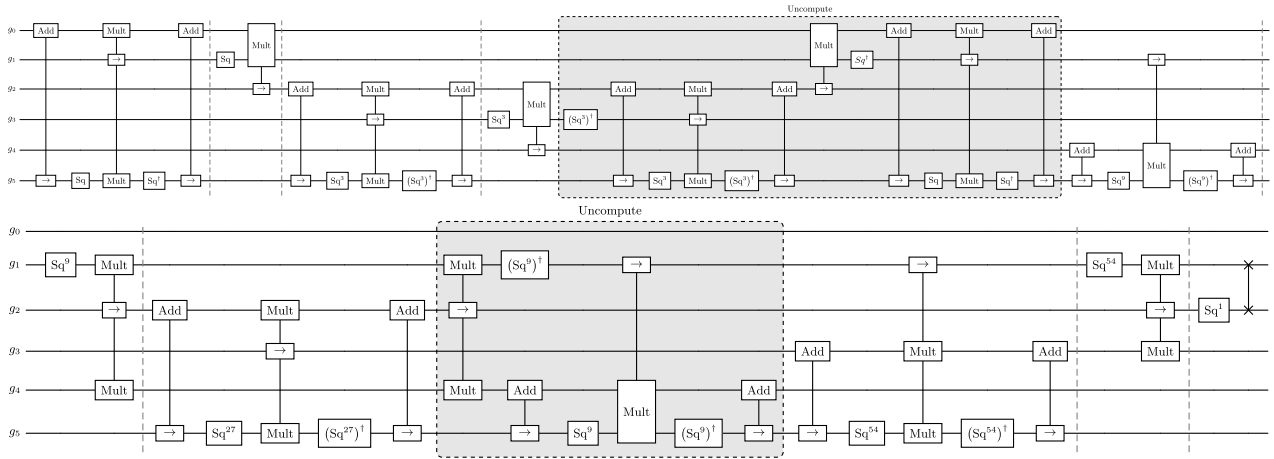
$$\begin{aligned} &\{2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^7 + 2^5, 2^7 + 2^5 + 2^1\} \\ &= \{1, 2, 4, 8, 16, 32, 64, 128, 160, 162\}. \end{aligned} \quad (50)$$

Note that the length  $l$  of the addition chain (not including the first term) is 9. There are seven terms  $\{2, 4, 8, 16, 32, 64, 128\}$ , which we refer to as doubled terms, and two terms,  $\{160, 162\}$ , which we refer to as added terms. Every doubled term is computed by doubling its previous term, and every added term is computed by adding any two of its previous terms. The algorithms in [24] and [26] allow the use of arbitrary addition chains to calculate  $n - 1$ . For example, consider an alternative addition chain for  $n - 1 = 162$

$$\{1, 2, 3, 6, 9, 18, 27, 54, 108, 162\}. \quad (51)$$

This chain also has a length of 9; however, note that it contains five doubled terms  $\{2, 6, 18, 54, 108\}$  and four added terms  $\{3, 9, 27, 162\}$ . In the next section, we will explain how addition chains are used to compute inverses in quantum circuits, which will clarify how properties of the addition chain affect the gate count. For now, it is important to note that the length of the chain  $l$  corresponds to the number of modular multiplications required (i.e., Toffoli gates), and that computing doubled terms is more resource-intensive in terms of CNOT gates compared to computing added terms, which we discuss below. Thus, in this case, computing inverses

<sup>4</sup>An addition chain for a nonnegative integer  $n - 1$  is a sequence  $\alpha_0 = 1, \alpha_1, \alpha_2, \dots, \alpha_l = n - 1$ , with the property that each  $\alpha_i$ , after  $\alpha_0$ , is obtained by adding two earlier terms (not necessarily distinct). The number  $l$  is called the length of the addition chain.



**FIGURE 8.** Quantum circuit for computing the modular inverse for  $n = 163$  using the FLT-based inversion algorithm from [26]. The circuit is split into two parts: the top circuit represents the first half, while the bottom circuit corresponds to the second half. The circuit takes as input an irreducible polynomial  $p(x)$  of degree  $n$  and a polynomial  $f(x) \in \mathbb{F}_{2^n}^*$ . The input polynomial  $f(x)$  is stored in register  $g_0$ , and its computed inverse  $f(x)^{-1}$  is stored in the  $g_1$  register. The algorithm requires  $R$   $n$ -qubit ancilla registers, where  $R = 2l - \tilde{l} + 1$ . At the end of the algorithm, the  $g_5$  register is cleared to the  $|0\rangle^n$  state. The addition chain used in this example can be found in Table 11. As in Fig. 3, the “ $\rightarrow$ ” marks the target registers whose values are modified by modular multiplication and addition. The squaring operation is denoted by “Sq,” and  $k$  consecutive squaring operations are denoted “Sq<sup>k</sup>.” Each vertical dashed line indicates the computation of a doubled or added term in the addition chain. The highlighted sections of the circuit correspond to the clearing steps of the algorithm, specifically the terms {6, 3, 2} and {27, 18} in the given addition chain example.

with the addition chain in (51) requires fewer CNOT gates compared to the addition chain (50).

However, a further observation can be made. Suppose each term  $\alpha$  in the addition chain, corresponding to a polynomial  $\langle 2^\alpha - 1 \rangle$ , must be stored in an  $n$ -qubit register within the circuit. Notice that some terms in the addition chain are not reused later in the computation. Consequently, these terms can potentially be cleared—i.e., the corresponding registers can be returned to the  $|0\rangle^n$  state—and the cleared registers can then be reused to compute subsequent terms in the chain. For example, consider the addition chain for  $n - 1 = 162$

$$\{1, 2, 3, 6, 9, 6, 3, 2, 18, 27, 54, 27, 18, 108, 162\}. \quad (52)$$

In this addition chain, there are now decreasing terms that indicate which polynomials can be cleared during the algorithm. According to (52), the algorithm takes as input the register that stores the polynomial  $\langle 2^1 - 1 \rangle$  and then proceeds to compute the polynomials  $\langle 2^2 - 1 \rangle$ ,  $\langle 2^3 - 1 \rangle$ ,  $\langle 2^6 - 1 \rangle$ , and  $\langle 2^9 - 1 \rangle$ , corresponding to  $\{2, 3, 6, 9\}$ , which are stored in distinct  $n$ -qubit registers. Subsequently, the decreasing terms  $\{6, 3, 2\}$  indicate that the registers storing  $\langle 2^6 - 1 \rangle$ ,  $\langle 2^3 - 1 \rangle$ , and  $\langle 2^2 - 1 \rangle$  can be cleared. This is because the terms are not reused in the addition chain and, importantly, can be cleared using previously stored polynomials. Explicitly, the following holds.

- 1)  $\langle 2^6 - 1 \rangle$ , corresponding to  $\{6\}$ , can be cleared using  $\langle 2^3 - 1 \rangle$ , corresponding to  $\{3\}$ .
- 2)  $\langle 2^3 - 1 \rangle$  can be cleared using the previously computed polynomials  $\langle 2^2 - 1 \rangle$  and  $\langle 2^1 - 1 \rangle$ .
- 3)  $\langle 2^2 - 1 \rangle$  can be cleared using the polynomial  $\langle 2^1 - 1 \rangle$ .

Similarly, the terms  $\{27, 18\}$  indicate that registers storing  $\langle 2^{27} - 1 \rangle$  and  $\langle 2^{18} - 1 \rangle$  can also be cleared. However, this

clearing process—i.e., returning a register to the zero state for reuse—comes at a cost. Each term that is cleared requires a modular multiplication and may also involve additional squaring operations. However, the upshot of this clearing process is significant: for a slight increase in the number of multiplication operations, we can approximately halve the number of ancilla qubits required for inversion for our values of  $n = 163$  233 283 571. The total number of required ancilla qubits is  $Rn$ , where  $R = (2l - \tilde{l} + 1)$ ,  $\tilde{l}$  is the length of the addition chain (excluding the initial term 1), and  $l$  is the number of strictly increasing terms in the addition chain (also excluding the initial term 1). Note that the clearing process can be omitted to further reduce the Toffoli count [24], but this significantly increases the number of ancilla qubits required. The resource estimates for this inversion algorithm without the clearing process are given in Table 12.

## 2) QUANTUM CIRCUIT FOR FLT-BASED MODULAR INVERSION WITH ADDITION CHAINS

In this section, we outline how to implement the quantum circuit for FLT-based modular inversion with addition chains [26]. We provide an explicit inversion circuit for  $n = 163$  in Fig. 8.

To compute  $f^{-1}$ , the inversion algorithm [26] takes as input an addition chain of  $\tilde{l}$ , excluding the first term, and the state  $|f\rangle|0\rangle^{Rn}$ . The output state is  $|f\rangle|f^{-1}\rangle|\text{garbage}\rangle|0\rangle$ , where  $|\text{garbage}\rangle$  is of size  $(R - 2)n$ . The number of ancilla qubits required is given by  $Rn$ , where  $R = (2l - \tilde{l} + 1)$ . The number of modular multiplications is determined by  $\tilde{l}$ , and the number of CNOT gates depends on the added and doubled terms in the addition chain. At the end of the inversion algorithm, the first ancilla register contains the result  $f^{-1}$ , and an

ancilla register is cleared to the  $|0\rangle^n$  state. This cleared register will be exploited and used to store intermediate arithmetic results (see Section III).

To explain the algorithm, and without loss of generality, consider the following addition chain:

$$\{\alpha, \beta, \gamma, \delta, \gamma, \beta\} \quad (53)$$

where  $\alpha, \beta, \gamma$ , and  $\delta$  are nonnegative integers such that  $\alpha < \beta < \gamma < \delta$ ,  $\beta$  is a doubled term ( $\beta = 2\alpha$ ),  $\gamma$  is an added term such that  $\gamma = \alpha + \beta$ , and  $\delta$  is a doubled term ( $\delta = 2\gamma$ ). We now demonstrate how each term  $\nu$  in the addition chain—whether a doubled or added term—corresponding to  $\langle 2^\nu - 1 \rangle$  is computed and then cleared within a quantum circuit [24], [26].

- *Computing a doubled term:* Suppose we have a term  $\alpha$  in the addition chain that we want to double, i.e.,  $\beta = 2\alpha$ . This corresponds to a register  $g_i$  (for some index  $i$ ) storing the polynomial  $\langle 2^\alpha - 1 \rangle$ . To compute  $\beta = 2\alpha$ , proceed as follows. Using the binary addition subroutine, add the polynomial stored in register  $g_i$  to  $h$ , where  $h$  is initially in the  $|0\rangle^n$  state, resulting in  $h = \langle 2^\alpha - 1 \rangle$ . Next, apply the squaring operation to  $h$ ,  $\alpha$  number of times, resulting in  $h = \langle 2^\alpha - 1 \rangle^{2^\alpha}$ . Then, apply modular multiplication to the polynomials stored in  $g_i$  and  $h$ , outputting the result in a register  $g_{i+1}$ , which is in the state  $|0\rangle^n$ . This results in

$$g_{i+1} = \langle 2^\alpha - 1 \rangle^{2^\alpha} \langle 2^\alpha - 1 \rangle = \langle 2^{2\alpha} - 1 \rangle \quad (54)$$

where  $\beta = 2\alpha$  is the desired doubled term in the addition chain. Lastly, to clear  $h$ , apply the circuit for squaring in reverse,  $\alpha$  times, and add  $g_i$  to  $h$ . This last step ensures that  $h$  is returned to its initial state  $|0\rangle^n$  and can be reused.

- *Computing an added term:* Suppose we have terms  $\alpha$  and  $\beta$  in the addition chain that we want to add, where this corresponds to a register  $g_i$  storing  $\langle 2^\alpha - 1 \rangle$  and  $g_{i+1}$  storing  $\langle 2^\beta - 1 \rangle$ , respectively. To compute  $\gamma = \alpha + \beta$ , proceed as follows. Apply the squaring operation to  $g_{i+1}$ ,  $\alpha$  number of times, resulting in  $g_{i+1} = \langle 2^\beta - 1 \rangle^{2^\alpha}$ . Then, apply modular multiplication to the polynomials stored in  $g_i$  and  $g_{i+1}$ , outputting the result in a register  $g_j$ , (which initially is in the state  $|0\rangle^n$ ). This results in

$$g_j = \langle 2^\alpha - 1 \rangle \langle 2^\beta - 1 \rangle^{2^\alpha} = \langle 2^{\alpha+\beta} - 1 \rangle \quad (55)$$

where  $\gamma = \alpha + \beta$  is the desired added term in the addition chain. Notice that the register  $g_{i+1}$  now stores  $\langle 2^\beta - 1 \rangle^{2^\alpha}$ ; therefore, if  $\beta$  is reused, then we would need to undo the squaring operations.

The addition chain in (53) indicates that we can clear, in order, the registers containing  $\langle 2^\gamma - 1 \rangle$  and  $\langle 2^\beta - 1 \rangle$ , corresponding to the terms  $\gamma$  and  $\beta$ , respectively. The clearing process, returning a register to the  $|0\rangle^n$  state, entails reversing the operation used to compute the term. Specifically, this involves, potentially squaring (or an inverse squaring)

operation, reapplying the corresponding operation (whether doubling or addition), and then performing binary addition. Since adding a polynomial to itself twice results in zero, the register is effectively reset to the  $|0\rangle^n$  state. This process can be performed as long as the terms in the addition chain are not reused later, and the corresponding polynomial remains present and stored in the quantum circuit at the time of clearing.

- *Clearing an added term:* Suppose we want to clear the register  $g_j$  containing the polynomial corresponding to the added term  $\gamma = \alpha + \beta$ . We have that  $\langle 2^\beta - 1 \rangle^{2^\alpha}$  is stored in register  $g_{i+1}$ , and  $\langle 2^\alpha - 1 \rangle$  is stored in register  $g_i$ , from the previous steps of the algorithm.<sup>5</sup> To clear the register  $g_j$ , apply modular multiplication to the polynomials stored in  $g_i$  and  $g_{i+1}$ , outputting the result in the register  $g_j$ , which contains the polynomial  $\langle 2^\gamma - 1 \rangle$ . This results in

$$g_j = \langle 2^\gamma - 1 \rangle + \langle 2^\gamma - 1 \rangle = 0 \quad (56)$$

hence, the register  $g_j$  is cleared to the  $|0\rangle^n$  state.

- *Clearing a doubled term:* Suppose we want to clear the register containing the polynomial corresponding to the doubled term  $\beta = 2\alpha$ . From the previous steps of the algorithm,  $\langle 2^\beta - 1 \rangle^{2^\alpha}$  is stored in register  $g_{i+1}$ , and  $\langle 2^\alpha - 1 \rangle$  is stored in register  $g_i$ . To clear the register,  $g_{i+1}$  we first apply the circuit for squaring in reverse,  $\alpha$  times, resulting in  $g_{i+1} = \langle 2^\beta - 1 \rangle$ . Then, the process is similar to that of computing a doubled term. Add the polynomial stored in register  $g_i$  to  $h$ , where  $h$  is initially in the  $|0\rangle^n$  state. Next, apply the squaring operation to  $h$ ,  $\alpha$  number of times. Then, apply modular multiplication to the polynomials stored in  $g_i$  and  $h$ , outputting the result in the register  $g_{i+1}$

$$g_{i+1} = \langle 2^\beta - 1 \rangle + \langle 2^\beta - 1 \rangle = 0. \quad (57)$$

This has now cleared the  $g_{i+1}$  register to the  $|0\rangle^n$  state.

The above description shows how to compute/uncompute each term (either a doubled or added term) in the addition chain in a quantum circuit. The last term in the addition chain corresponds to calculating  $\langle 2^{n-1} - 1 \rangle$ . Lastly, to compute the inverse, we simply square the term  $\langle 2^{n-1} - 1 \rangle$  to obtain  $\langle 2^n - 2 \rangle \equiv \langle -1 \rangle$ .

We briefly comment on optimizations performed in the resource estimation. In the algorithm, there are many repeated consecutive calls to the modular squaring operation—for example, to double the term  $\gamma$ , one would need to perform  $\gamma$  squaring operations (as well as running the inverse of this circuit to clear the register). As discussed in Section V-A, there are two approaches one could use: perform each squaring operation separately in the circuit or combine the

<sup>5</sup>If this is not the case, e.g., for a different addition chain where the register  $g_{i+1}$  contains  $\langle 2^\beta - 1 \rangle^{2^k}$  for some integer  $k$ , and then apply the necessary squaring (or inverse) operation to ensure the exponent is  $2^\alpha$ , i.e.,  $\langle 2^\beta - 1 \rangle^{2^\alpha}$ .

**TABLE 11. Addition Chains for Values of  $n$  Used in This Work for the Inversion Algorithm from [26]**

$n$	Addition Chains
163	{1, 2, 3, 6, 9, 6, 3, 2, 18, 27, 54, 27, 18, 108, 162}
233	{1, 2, 3, 4, 7, 4, 3, 2, 14, 28, 29, 28, 14, 58, 116, 58, 232}
283	{1, 2, 3, 6, 9, 15, 9, 6, 3, 30, 45, 47, 45, 30, 2, 94, 141, 94, 282}
571	{1, 2, 3, 4, 7, 4, 3, 2, 14, 28, 29, 57, 29, 28, 14, 114, 171, 285, 171, 114, 570}

The chains were taken directly from [26] and include terms that indicate which register can be cleared in the quantum circuit.

**TABLE 12. Costs of Computing  $f^{-1}(x) \bmod p(x)$  Given  $f(x)$  Via the FLT-Based Inversion algorithm [24], Using the Addition Chains in Table 11 Without the Clearing Process**

$n$	# ModMults	# Toffolis	# CNOTs	# Swaps	Active Volume
163	9	8991	1 096 546	13265	$4.81 \times 10^6$
233	10	14 480	2 408 816	52610	$1.03 \times 10^7$
283	11	19 536	3 977 687	43671	$1.68 \times 10^7$
571	12	46 320	16 058 155	114550	$6.64 \times 10^7$

The costs are stated in terms of the number of modular multiplication applications, Toffolis, CNOTs, and swaps, as well as AV. In this approach, the ancilla qubit requirement is  $l \cdot n$ , where  $l$  is the number of strictly increasing terms in the addition chain (excluding the initial term 1) and is equal to the number of modular multiplications.

consecutive squaring operations into one operation, and then implement this in the circuit. In our resource estimation, we numerically calculated both methods and chose the method with a smaller CNOT count. A similar optimization was also applied in [26], although it is unclear to us whether our method coincides with theirs. The resource estimates for the FLT-based inversion algorithm from [26], using the addition chains in Table 11 and the clearing process, are summarized in Table 3. Note that the gate counts reported in [26] are derived using the circuits from [25] without our corrections and optimizations in the modular multiplication routine.

Lastly, we note that the Toffoli count in the inversion algorithm can be reduced if the clearing process is omitted [24]. In this case, one can use the algorithm from [24], which takes as input  $|f\rangle|0\rangle^{l \cdot n}$  and outputs  $|f\rangle|\text{garbage}\rangle|f^{-1}\rangle$ , with  $|\text{garbage}\rangle$  being of size  $(l - 1)n$ . This approach reduces the number of modular multiplications to  $l$  but increases the ancilla qubit requirement to  $l \cdot n$ . The resource estimates for this inversion algorithm, using the addition chains in Table 11 without the clearing process, are provided in Table 12. Note that the gate counts reported in [24] are derived using the circuits from [25] without our corrections and optimizations in the modular multiplication routine.

## ACKNOWLEDGMENT

Angus Kan thanks Sukin Sim for insightful discussions on physical resource estimation methodologies, and Daniel Litinski and Sam Pallister for valuable comments.

## REFERENCES

- [1] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134, doi: [10.1109/SFCS.1994.365700](https://doi.org/10.1109/SFCS.1994.365700).
- [2] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Rev.*, vol. 41, no. 2, pp. 303–332, 1999, doi: [10.1137/S0036144598347011](https://doi.org/10.1137/S0036144598347011).
- [3] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 26, no. 1, pp. 96–99, Jan. 1983, doi: [10.1145/357980.358017](https://doi.org/10.1145/357980.358017).

- [4] S. D. Galbraith and P. Gaudry, "Recent progress on the elliptic curve discrete logarithm problem," *Des., Codes Cryptography*, vol. 78, pp. 51–72, 2016, doi: [10.1007/s10623-015-0146-7](https://doi.org/10.1007/s10623-015-0146-7).
- [5] J. Proos and C. Zalka, "Shor's discrete logarithm quantum algorithm for elliptic curves," *arXiv:quant-ph/0301141*, 2004, doi: [10.48550/arXiv.quant-ph/0301141](https://doi.org/10.48550/arXiv.quant-ph/0301141).
- [6] P. Kaye and C. Zalka, "Optimized quantum implementation of elliptic curve arithmetic over binary fields," *arXiv:quant-ph/0407095*, 2004, doi: [10.48550/arXiv.quant-ph/0407095](https://doi.org/10.48550/arXiv.quant-ph/0407095).
- [7] D. Cheung, D. Maslov, J. Mathew, and D. K. Pradhan, "On the design and optimization of a quantum polynomial-time attack on elliptic curve cryptography," in *Proc. Theory Quantum Comput., Commun., Cryptography: 3rd Workshop, TQC 2008*, 2008, pp. 96–104, doi: [10.1007/978-3-540-89304-2\\_9](https://doi.org/10.1007/978-3-540-89304-2_9).
- [8] D. Maslov, J. Mathew, D. Cheung, and D. K. Pradhan, "An  $O(m^2)$ -depth quantum algorithm for the elliptic curve discrete logarithm problem over  $GF(2^m)^a$ ," *Quantum Inf. Comput.*, vol. 9, no. 7, pp. 610–621, Jul. 2009, doi: [10.26421/QIC9.7-8.4](https://doi.org/10.26421/QIC9.7-8.4).
- [9] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Phys. Rev. A*, vol. 86, Sep. 2012, Art. no. 032324, doi: [10.1103/PhysRevA.86.032324](https://doi.org/10.1103/PhysRevA.86.032324).
- [10] B. Amento, M. Rötteler, and R. Steinwandt, "Efficient quantum circuits for binary elliptic curve arithmetic: Reducing T-gate complexity," *Quantum Inf. Comput.*, vol. 13, no. 7–8, pp. 631–644, Jul. 2013, doi: [10.48550/arXiv.1209.6348](https://doi.org/10.48550/arXiv.1209.6348).
- [11] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter, "Quantum resource estimates for computing elliptic curve discrete logarithms," in *Advances in Cryptology—ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds., Cham, Switzerland: Springer, 2017, pp. 241–270, doi: [10.1007/978-3-319-70697-9\\_9](https://doi.org/10.1007/978-3-319-70697-9_9).
- [12] T. Häner, M. Roetteler, and K. M. Svore, "Factoring using  $2n+2$  qubits with Toffoli based modular multiplication," *Quantum Inf. Comput.*, vol. 17, no. 7–8, pp. 673–684, Jun. 2017, doi: [10.48550/arXiv.1611.07995](https://doi.org/10.48550/arXiv.1611.07995).
- [13] V. Gheorghiu and M. Mosca, "Benchmarking the quantum cryptanalysis of symmetric, public-key and hash-based cryptographic schemes," *arXiv:1902.0233*, 2019, doi: [10.48550/arXiv.1902.02332](https://doi.org/10.48550/arXiv.1902.02332).
- [14] I. van Hoof, "Space-efficient quantum multiplication polynomials for binary finite fields with sub-quadratic Toffoli gate count," *Quantum Inf. Comput.*, vol. 20, no. 9–10, pp. 721–735, 2020, doi: [10.26421/QIC20.9-10-1](https://doi.org/10.26421/QIC20.9-10-1).
- [15] T. Häner, S. Jaques, M. Naehrig, M. Roetteler, and M. Soeken, "Improved quantum circuits for elliptic curve discrete logarithms," in *Post-Quantum Cryptography*, J. Ding and J.-P. Tillich, Eds., Cham, Switzerland: Springer, 2020, pp. 425–444, doi: [10.1007/978-3-030-44223-1\\_23](https://doi.org/10.1007/978-3-030-44223-1_23).
- [16] G. Banegas, D. J. Bernstein, I. van Hoof, and T. Lange, "Concrete quantum cryptanalysis of binary elliptic curves," *Cryptol. ePrint Arch.*, Paper 2020/1296, 2020, [Online]. Available: <https://eprint.iacr.org/2020/1296>
- [17] E. Gouzien, D. Ruiz, F.-M. Le Régent, J. Guillaud, and N. Sangouard, "Performance analysis of a repetition cat code architecture: Computing 256-bit elliptic curve logarithm in 9 hours with 126 133 cat qubits," *Phys. Rev. Lett.*, vol. 131, Jul. 2023, Art. no. 040602, doi: [10.1103/PhysRevLett.131.040602](https://doi.org/10.1103/PhysRevLett.131.040602).
- [18] C. Gidney and M. Ekerå, "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits," *Quantum*, vol. 5, Apr. 2021, Art. no. 433, doi: [10.22331/q-2021-04-15-433](https://doi.org/10.22331/q-2021-04-15-433).
- [19] D. S. C. Putranto, R. W. Wardhani, H. T. Larasati, and H. Kim, "Another concrete quantum cryptanalysis of binary elliptic curves," *Cryptol. ePrint Arch.*, Paper 2022/501, 2022, [Online]. Available: <https://eprint.iacr.org/2022/501>
- [20] D. Litinski and N. Nickerson, "Active volume: An architecture for efficient fault-tolerant quantum computers with limited non-local connections," *arXiv:2211.15465*, 2022, doi: [10.48550/arXiv.2211.15465](https://doi.org/10.48550/arXiv.2211.15465).
- [21] D. Litinski, "How to compute a 256-bit elliptic curve private key with only 50 million Toffoli gates," *arXiv:2306.08585*, 2023, doi: [10.48550/arXiv.2306.08585](https://doi.org/10.48550/arXiv.2306.08585).
- [22] D. S. C. Putranto, R. W. Wardhani, H. T. Larasati, J. Ji, and H. Kim, "Depth-optimization of quantum cryptanalysis on binary elliptic curves," *IEEE Access*, vol. 11, pp. 45083–45097, 2023, doi: [10.1109/ACCESS.2023.3273601](https://doi.org/10.1109/ACCESS.2023.3273601).

- [23] H. Kim and S. Hong, "New space-efficient quantum algorithm for binary elliptic curves using the optimized division algorithm," *Quantum Inf. Process.*, vol. 22, no. 6, 2023, Art. no. 237, doi: [10.1007/s11128-023-03991-6](https://doi.org/10.1007/s11128-023-03991-6).
- [24] R. Taguchi and A. Takayasu, "Concrete quantum cryptanalysis of binary elliptic curves via addition chain," in *Topics in Cryptology—CT-RSA*, M. Rosulek, Ed., Cham, Switzerland: Springer, 2023, pp. 57–83, doi: [10.1007/978-3-031-58868-6](https://doi.org/10.1007/978-3-031-58868-6).
- [25] S. Kim, I. Kim, S. Kim, and S. Hong, "Toffoli gate count optimized space-efficient quantum circuit for binary field multiplication," *Quantum Inf. Process.*, vol. 23, no. 10, 2024, Art. no. 330, doi: [10.1007/s11128-024-04536-1](https://doi.org/10.1007/s11128-024-04536-1).
- [26] R. Taguchi and A. Takayasu, "On the untapped potential of the quantum FLT-based inversion," in *Applied Cryptography and Network Security*, C. Pöpper and L. Batina, Eds., Cham, Switzerland: Springer Nature, 2024, pp. 79–100, doi: [10.1007/978-3-031-33488-7](https://doi.org/10.1007/978-3-031-33488-7).
- [27] L. Chen, D. Moody, A. Regenscheid, and K. Randall, "Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters," Nat. Inst. Standards Technol., Gaithersburg, MD, USA: NIST Special Publication, Tech. Rep., 800-186, 2023, doi: [10.6028/NIST.SP.800-186](https://doi.org/10.6028/NIST.SP.800-186).
- [28] D. Moody, R. Perlner, A. Regenscheid, A. Robinson, and D. Cooper, "Transition to post-quantum cryptography standards," *Nat. Inst. Standards Technol.*, Gaithersburg, MD, USA, NIST Internal, Tech. Rep., 8547, doi: [10.6028/NIST.IR.8547.ipd](https://doi.org/10.6028/NIST.IR.8547.ipd).
- [29] S. B. Bravyi and A. Y. Kitaev, "Quantum codes on a lattice with boundary," *arXiv:quant-ph/9811052*, 1998, doi: [10.48550/arXiv.quant-ph/9811052](https://doi.org/10.48550/arXiv.quant-ph/9811052).
- [30] A. Kitaev, "Fault-tolerant quantum computation by anyons," *Ann. Phys.*, vol. 303, no. 1, pp. 2–30, 2003, doi: [10.1016/S0003-4916\(02\)00018-0](https://doi.org/10.1016/S0003-4916(02)00018-0).
- [31] S. Bravyi and A. Kitaev, "Universal quantum computation with ideal Clifford gates and noisy ancillas," *Phys. Rev. A*, vol. 71, Feb. 2005, Art. no. 022316, doi: [10.1103/PhysRevA.71.022316](https://doi.org/10.1103/PhysRevA.71.022316).
- [32] A. G. Fowler and C. Gidney, "Low overhead quantum computation using lattice surgery," *arXiv:1808.06709*, 2019, doi: [10.48550/arXiv.1808.06709](https://doi.org/10.48550/arXiv.1808.06709).
- [33] D. Litinski, "A game of surface codes: Large-scale quantum computing with lattice surgery," *Quantum*, vol. 3, Mar. 2019, Art. no. 128, doi: [10.22331/q-2019-03-05-128](https://doi.org/10.22331/q-2019-03-05-128).
- [34] H. Bombin et al., "Interleaving: Modular architectures for fault-tolerant photonic quantum computing," *arXiv:2103.08612*, 2021, doi: [10.48550/arXiv.2103.08612](https://doi.org/10.48550/arXiv.2103.08612).
- [35] H. T. Larasati and H. Kim, "Quantum circuit designs of point doubling operation for binary elliptic curves," in *Information Security Applications*, H. Kim and J. Youn, Eds., Singapore: Springer Nature, 2024, pp. 297–309, doi: [10.1007/978-981-99-8024-6](https://doi.org/10.1007/978-981-99-8024-6).
- [36] S. Bartolucci et al., "Fusion-based quantum computation," *Nature Commun.*, vol. 14, no. 1, 2023, Art. no. 912, doi: [10.1038/s41467-023-36493-1](https://doi.org/10.1038/s41467-023-36493-1).
- [37] W. Song, N. Kang, Y.-S. Kim, and S.-W. Lee, "Encoded-fusion-based quantum computation for high thresholds with linear optics," *Phys. Rev. Lett.*, vol. 133, Aug. 2024, Art. no. 050605, doi: [10.1103/PhysRevLett.133.050605](https://doi.org/10.1103/PhysRevLett.133.050605).
- [38] B. Pankovich et al., "High-photon-loss threshold quantum computing using GHZ-state measurements," *Phys. Rev. Lett.*, vol. 133, Aug. 2024, Art. no. 050604, doi: [10.1103/PhysRevLett.133.050604](https://doi.org/10.1103/PhysRevLett.133.050604).
- [39] M. L. Chan et al., "Tailoring fusion-based photonic quantum computing schemes to quantum emitters," *PRX Quantum*, vol. 6, Apr., 2025, Art. no. 020304, doi: [10.1103/PRXQuantum.6.020304](https://doi.org/10.1103/PRXQuantum.6.020304).
- [40] S. Bartolucci et al., "Creation of entangled photonic states using linear optics," *arXiv:2106.13825*, 2021, doi: [10.48550/arXiv.2106.13825](https://doi.org/10.48550/arXiv.2106.13825).
- [41] B. Pankovich, A. Neville, A. Kan, S. Omkar, K. H. Wan, and K. Brádler, "Flexible entangled-state generation in linear optics," *Phys. Rev. A*, vol. 110, Sep. 2024, Art. no. 032402, doi: [10.1103/PhysRevA.110.032402](https://doi.org/10.1103/PhysRevA.110.032402).
- [42] S. Meesala et al., "Quantum entanglement between optical and microwave photonic qubits," *Phys. Rev. X*, vol. 14, Sep. 2024, Art. no. 031055, doi: [10.1103/PhysRevX.14.031055](https://doi.org/10.1103/PhysRevX.14.031055).
- [43] J. Sinclair, J. Ramette, B. Grinkemeyer, D. Bluvstein, M. Lukin, and V. Vuletić, "Fault-tolerant optical interconnects for neutral-atom arrays," *arXiv:2408.08955*, 2024, doi: [10.48550/arXiv.2408.08955](https://doi.org/10.48550/arXiv.2408.08955).
- [44] C. Monroe et al., "Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects," *Phys. Rev. A*, vol. 89, Feb. 2014, Art. no. 022317, doi: [10.1103/PhysRevA.89.022317](https://doi.org/10.1103/PhysRevA.89.022317).
- [45] V. Krutyanskiy et al., "Entanglement of trapped-ion qubits separated by 230 meters," *Phys. Rev. Lett.*, vol. 130, Feb. 2023, Art. no. 050803, doi: [10.1103/PhysRevLett.130.050803](https://doi.org/10.1103/PhysRevLett.130.050803).
- [46] S. Storz et al., "Loop-hole-free bell inequality violation with superconducting circuits," *Nature*, vol. 617, no. 7960, pp. 265–270, 2023, doi: [10.1038/s41586-023-05885-0](https://doi.org/10.1038/s41586-023-05885-0).
- [47] R. D. Delaney et al., "Scalable multispecies ion transport in a grid-based surface-electrode trap," *Phys. Rev. X*, vol. 14, Nov. 2024, Art. no. 041028, doi: [10.1103/PhysRevX.14.041028](https://doi.org/10.1103/PhysRevX.14.041028).
- [48] PsiQuantum team, "A manufacturable platform for photonic quantum computing," *Nature*, vol. 641, Feb. pp. 876–883, 2025, doi: [10.1038/s41586-025-08820-7](https://doi.org/10.1038/s41586-025-08820-7).
- [49] H. Aghaee Rad et al., "Scaling and networking a modular photonic quantum computer," *Nature*, vol. 638, pp. 912–919, Jan. 2025, doi: [10.1038/s41586-024-08406-9](https://doi.org/10.1038/s41586-024-08406-9).
- [50] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in  $GF(2^m)$  using normal bases," *Inf. Comput.*, vol. 78, no. 3, pp. 171–177, 1988, doi: [10.1016/0890-5401\(88\)90024-7](https://doi.org/10.1016/0890-5401(88)90024-7).
- [51] B. Sunar, "A generalized method for constructing subquadratic complexity  $GF(2^k)$  multipliers," *IEEE Trans. Comput.*, vol. 53, no. 9, pp. 1097–1105, Sep. 2004, doi: [10.1109/TC.2004.52](https://doi.org/10.1109/TC.2004.52).
- [52] H. Fan and M. A. Hasan, "Comments on 'five, six, and seven-term Karatsuba-like formulae,'" *IEEE Trans. Comput.*, vol. 56, no. 5, pp. 716–717, May 2007, doi: [10.1109/TC.2007.1024](https://doi.org/10.1109/TC.2007.1024).
- [53] M. G. Find and R. Peralta, "Better circuits for binary polynomial multiplication," *IEEE Trans. Comput.*, vol. 68, no. 4, pp. 624–630, Apr. 2019, doi: [10.1109/TC.2018.2874662](https://doi.org/10.1109/TC.2018.2874662).
- [54] Ç. Çalik, M. Dworkin, N. Dykas, and R. Peralta, "Searching for best Karatsuba recurrences," in *Analysis of Experimental Algorithms*, I. Kotsireas, P. Pardalos, K. E. Parsopoulos, D. Souravlias, and A. Tsokas, Eds., Cham, Switzerland: Springer, 2019, pp. 332–342, doi: [10.1007/978-3-030-34029-2](https://doi.org/10.1007/978-3-030-34029-2).
- [55] C. F. Kerry and P. D. Gallagher, "Digital signature standard," Nat. Inst. Standards Technol., Gaithersburg, MD, Tech. Rep., 2013, doi: [10.6028/NIST.FIPS.186-4](https://doi.org/10.6028/NIST.FIPS.186-4).
- [56] H. Cohen et al., *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, 1st ed. Boston, MA, USA: Chapman & Hall/CRC, 2005, doi: [10.1201/9780367801625](https://doi.org/10.1201/9780367801625).
- [57] F. P. Papa, "Validation of quantum elliptic curve point addition circuits," *arXiv:2506.03318*, 2025, doi: [10.48550/arXiv.2506.03318](https://doi.org/10.48550/arXiv.2506.03318).
- [58] B. Amento, M. Rötteler, and R. Steinwandt, "Quantum binary field inversion: Improved circuit depth via choice of basis representation," *Quantum Inf. Comput.*, vol. 13, no. 1-2, pp. 116–134, Jan. 2013, doi: [10.48550/arXiv.1209.5491](https://doi.org/10.48550/arXiv.1209.5491).
- [59] C. Jones, "Low-overhead constructions for the fault-tolerant Toffoli gate," *Phys. Rev. A*, vol. 87, Feb. 2013, Art. no. 022328, doi: [10.1103/PhysRevA.87.022328](https://doi.org/10.1103/PhysRevA.87.022328).
- [60] R. Babbush et al., "Encoding electronic spectra in quantum circuits with linear T complexity," *Phys. Rev. X*, vol. 8, Oct. 2018, Art. no. 041015, doi: [10.1103/PhysRevX.8.041015](https://doi.org/10.1103/PhysRevX.8.041015).
- [61] D. W. Berry, C. Gidney, M. Motta, J. R. McClean, and R. Babbush, "Qubitization of arbitrary basis quantum chemistry leveraging sparsity and low rank factorization," *Quantum*, vol. 3, Dec. 2019, Art. no. 208, doi: [10.22331/q-2019-12-02-208](https://doi.org/10.22331/q-2019-12-02-208).
- [62] S. D. Galbraith, P. Wang, and F. Zhang, "Computing elliptic curve discrete logarithms with improved baby-step giant-step algorithm," *Cryptol. ePrint Arch.*, Paper 2015/605, 2015. [Online]. Available: <https://eprint.iacr.org/2015/605>
- [63] N. Kobitz, A. Menezes, and S. Vanstone, "The state of elliptic curve cryptography," *Des., Codes Cryptography*, vol. 19, pp. 173–193, 2000, doi: [10.1023/A:1008354106356](https://doi.org/10.1023/A:1008354106356).
- [64] D. Horsman, A. G. Fowler, S. Devitt, and R. Van Meter, "Surface code quantum computing by lattice surgery," *New J. Phys.*, vol. 14, no. 12, 2012, Art. no. 123011, doi: [10.1088/1367-2630/14/12/123011](https://doi.org/10.1088/1367-2630/14/12/123011).
- [65] A. Caesura et al., "Faster quantum chemistry simulations on a quantum computer with improved tensor factorization and active volume compilation," *arXiv:2501.06165*, 2025, doi: [10.48550/arXiv.2501.06165](https://doi.org/10.48550/arXiv.2501.06165).

- [66] H. Bombín, C. Dawson, R. V. Mishmash, N. Nickerson, F. Pastawski, and S. Roberts, “Logical blocks for fault-tolerant topological quantum computation,” *PRX Quantum*, vol. 4, Apr. 2023, Art. no. 020303, doi: [10.1103/PRXQuantum.4.020303](https://doi.org/10.1103/PRXQuantum.4.020303).
- [67] J. Vizslai, S. F. Lin, S. Dangwal, J. M. Baker, and F. T. Chong, “An architecture for improved surface code connectivity in neutral atoms,” *arXiv:2309.13507*, 2023, doi: [10.48550/arXiv.2309.13507](https://doi.org/10.48550/arXiv.2309.13507).
- [68] Q. Xu et al., “Constant-overhead fault-tolerant quantum computation with reconfigurable atom arrays,” *Nature Phys.*, vol. 20, pp. 1084–1090, 2024, doi: [10.1038/s41567-024-02479-z](https://doi.org/10.1038/s41567-024-02479-z).
- [69] M. E. Beverland et al., “Assessing requirements to scale to practical quantum advantage,” *arXiv:2211.07629*, 2022, doi: [10.48550/arXiv.2211.07629](https://doi.org/10.48550/arXiv.2211.07629).
- [70] J. Jiang, X. Sun, S.-H. Teng, B. Wu, K. Wu, and J. Zhang, “Optimal space-depth trade-off of CNOT circuits in quantum logic synthesis,” in *Proc. 2020 ACM-SIAM Symp. Discrete Algorithms*, 2020, pp. 213–229, doi: [10.1137/1.9781611975994.13](https://doi.org/10.1137/1.9781611975994.13).
- [71] D. Maslov and B. Zindorf, “Depth optimization of CZ, CNOT, and Clifford circuits,” *IEEE Trans. Quantum Eng.*, vol. 3, 2022, Art. no. 2500408, doi: [10.1109/TQE.2022.3180900](https://doi.org/10.1109/TQE.2022.3180900).
- [72] D. Maslov and M. Roetteler, “Shorter stabilizer circuits via Bruhat decomposition and quantum circuit transformations,” *IEEE Trans. Inf. Theory*, vol. 64, no. 7, pp. 4729–4738, Jul. 2018, doi: [10.1109/TIT.2018.2825602](https://doi.org/10.1109/TIT.2018.2825602).
- [73] K. N. Patel, I. L. Markov, and J. P. Hayes, “Optimal synthesis of linear reversible circuits,” *Quantum Inf. Comput.*, vol. 8, no. 3, pp. 282–294, Mar. 2008, doi: [10.5555/2011763.2011767](https://doi.org/10.5555/2011763.2011767).
- [74] T. Leblond, C. Dean, G. Watkins, and R. Bennink, “Realistic cost to execute practical quantum circuits using direct Clifford T lattice surgery compilation,” *ACM Trans. Quantum Comput.*, vol. 5, no. 4, pp. 1–28, Oct. 2024, doi: [10.1145/3689826](https://doi.org/10.1145/3689826).
- [75] O. Regev, “An efficient quantum factoring algorithm,” *J. ACM*, vol. 72, no. 1, pp. 1–13, Dec. 2024, doi: [10.1145/3708471](https://doi.org/10.1145/3708471).
- [76] M. Ekerå and J. Gärtner, “Extending Regev’s factoring algorithm to compute discrete logarithms,” in *Post-Quantum Cryptography*, M.-J. Saarinen and D. Smith-Tone, Eds., Cham, Switzerland: Springer Nature, 2024, pp. 211–242, doi: [10.1007/978-3-031-62746-0\\_10](https://doi.org/10.1007/978-3-031-62746-0_10).
- [77] P. Montgomery, “Five, six, and seven-term Karatsuba-like formulae,” *IEEE Trans. Comput.*, vol. 54, no. 3, pp. 362–369, Mar. 2005, doi: [10.1109/TC.2005.49](https://doi.org/10.1109/TC.2005.49).
- [78] A. Weimerskirch and C. Paar, “Generalizations of the Karatsuba algorithm for efficient implementations,” *Cryptol. ePrint Arch.*, Paper 2006/224, 2006. [Online]. Available: <https://eprint.iacr.org/2006/224>