Reducing the Number of Qubits in Quantum Factoring

Clémence Chevignard, Pierre-Alain Fouque, and André Schrottenloher

Univ Rennes, Inria, CNRS, IRISA
`firstname.lastname@inria.fr`

Abstract. This paper focuses on the optimization of the number of logical qubits in Shor's quantum factoring algorithm. As in previous works, we target the implementation of the modular exponentiation, which is the most costly component of the algorithm, both in qubits and operations.
In this paper, we show that using only $o(n)$ work qubits, one can obtain the first bit of the modular exponentiation output. We combine this result with May and Schlieper's truncation technique (ToSC 2022) and the Ekerå-Håstad variant of Shor's algorithm (PQCrypto 2017) to obtain a quantum factoring algorithm requiring only $n/2 + o(n)$ qubits in the case of an n-bit RSA modulus, while current envisioned implementations require about $2n$ qubits.
Our algorithm uses a Residue Number System and succeeds with a parametrizable probability. Being completely classical, we have implemented and tested it. Among possible trade-offs, we can reach a gate count $\mathcal{O}(n^3)$ for a depth $\mathcal{O}(n^2 \log^3 n)$, which then has to be multiplied by $\mathcal{O}(\log n)$ (the number of measurement results required by Ekerå-Håstad). Preliminary logical resource estimates suggest that this circuit could be engineered to use less than 1700 qubits and 2^{36} Toffoli gates, and require 60 independent runs to factor an RSA-2048 instance.

Keywords: Quantum cryptanalysis, Shor's algorithm, Integer factoring, Residue number system.

1 Introduction

In 1994, Shor [34] introduced a polynomial-time quantum algorithm for factoring integers and computing Discrete Logarithms. This remains to date one of the most powerful applications of quantum cryptanalysis, which caused the birth of *post-quantum cryptography*. In this paper, we will focus on factoring large integers.

Reduction to Modular Exponentiation. Shor's algorithm relies on a quantum order-finding subroutine represented in Figure 1, which finds the order of an element a in any Abelian group $(G, \cdot)$ where operations are efficiently computable.

For factoring a composite number N, the group is $\mathbb{Z}_N^*$, i.e., we simply compute modulo N.

It starts by initializing an *input register* x of m bits and a *workspace register*. It then produces a uniform superposition in the input x and computes a^x in the workspace: $a^x \bmod N$ in the case of factoring. It then performs a Quantum Fourier Transform on the input register again. After measurement, a classical post-processing extracts the order of a, i.e., the smallest positive integer r such that $a^r = 1$. This allows to factor N.

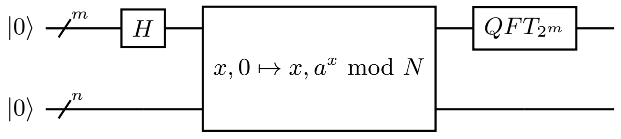

Fig. 1. Shor's quantum factoring subroutine.

Shor shows that using $m = 2\log_2 N$ input qubits is sufficient to recover the order with good probability. Besides, both the Hadamard and Fourier transform can be performed in place and efficiently, meaning that the computational cost is mostly determined by the cost of the modular exponentiation circuit. Like previous works, this is the part that we target. This operation is classical: it does not modify the superposition, and only maps basis states to other basis states. Therefore, it can entirely be implemented using classical reversible logic, i.e., Toffoli, CNOT and NOT gates. This is what we do in this paper, although many previous works have also used "inherently quantum" arithmetic circuits based on the QFT [10].

Optimization of Space. Since Shor's original paper, precisely estimating the cost of the algorithm has remained a crucial question. Many authors have designed circuits optimizing either its qubit or gate count [3,38,36,23,17]. Most of these works report a qubit count of $2n + 1$ or above, where $n = \log_2 N$ (see for example [17] for a comparison). The best count is from [38], where Zalka proposes a circuit with $1.5n$ qubits, and suggests that it could be reduced further. However, to date, this circuit has not been benchmarked in practice.

All of the recent implementations use the semi-classical Fourier transform [22] to save the space used by the input register, by reducing it to a single qubit which is repeatedly measured and reused. Thus, the cost of $1.5n$ qubits comes entirely from our current available implementations of modular exponentiation.

The goal of these optimizations is ultimately to reduce the physical resources required to factor large RSA moduli. For example, in [21] Gidney and Ekerå used a quantum circuit with around $3n$ logical qubits and $0.3n^3$ Toffoli gates. For RSA-2048, they estimated that 20 million physical qubits would be necessary. Reducing the qubit count by a factor 2 or 4 could make the difference in physical implementations, especially since near-term quantum devices remain of limited size.

The Compression Technique. May and Schlieper studied the behavior of Shor's subroutine if the workspace register is reduced to a fraction of its size, down to a single bit [26]. This means that the circuit does not implement $x \mapsto a^x \bmod N$ anymore, but $x \mapsto h(a^x \bmod N)$ where h is a good hash function mapping integers modulo N to $\{0, 1\}$. They showed that if h is picked from a universal hash function family, then the "compressed" version of Shor's algorithm has the same distribution of measurement results as the original one, except that the probability to measure 0 increases to 1/2 (and others are rescaled accordingly). The measure of 0 does not give any useful information and can simply be discarded.

Unfortunately, this can only improve the space complexity of the algorithm if $h(a^x \bmod N)$ can be implemented with less space than naively computing a^x, and then h. Whether such an implementation is possible at all was left as an open question.

As noticed by May and Schlieper, the compression technique, which targets the workspace register rather than the input, could be particularly useful in combination with the Ekerå-Håstad variant of Shor's algorithm [16]. This variant is capable of factoring n-bit RSA moduli using an input register of $\frac{n}{2} + o(n)$ bits only, thanks to their particular structure as the product of two $\frac{n}{2}$-bit primes, at the expense of performing multiple runs and a classical post-processing. Therefore, a "compressed" variant could reduce the number of qubits to $\frac{n}{2} + o(n)$.

Contributions. In this paper, we introduce a dedicated method to compute directly the "compressed" result of the modular exponentiation step in Shor's algorithm. More precisely, we compute only the first bit of $a^x \bmod N$, instead of the complete n-bit output. Our circuit uses only classical reversible logic, contains $\mathcal{O}(n^3)$ gates, depth $\widetilde{\mathcal{O}}(n^2)$ and $m + o(n)$ space, including the input register of size $m = \mathcal{O}(n)$ and a work register of size $o(n)$. Therefore, in combination with the May-Schlieper and Ekerå-Håstad methods, our algorithm can be used to factor n-bit RSA moduli using only $\frac{n}{2} + o(n)$ qubits. For a good trade-off, the total number of runs (i.e., measurements) should be around $\mathcal{O}(\log n)$.

Our method is inspired by results of circuit complexity and methods for optimizing circuit depth using Residue Number Systems (RNS). An RNS represents a large number by a collection of residues modulo a set of small primes. In our case, the number that we represent is a *multi-product*, i.e., the product of multiple precomputed integers modulo N corresponding to powers of a constant. Using the RNS, we find that the output bit that we wish to compute can be ultimately expressed as a large sum of integers over $\mathcal{O}(\log n)$ bits. These integers (cofactors) are precomputed, and the bits which control the sum are computed on the fly by reducing the multi-product modulo the small primes of the RNS.

Like state-of-the art implementations of quantum exponentiation [21], the obtained circuit is not exact. Most of the error stems from the truncation of a sum, which is very similar to the principle of oblivious carry runways [18].

Essentially, the latter allows to compute a sum faster by reasoning independently on several windows of bits, which remain independent with high probability (unless a long carry propagation occurred, which is very unlikely). In our case, we only need to compute one of the output bits, and we succeed for the same reason. Under heuristics, we can reduce the failure probability to any constant $\varepsilon > 0$, which is enough to guarantee that the Ekerå-Håstad and compression methods work.

The major difference with previous works is the circuit layout. Indeed, the size of the input register becomes dominating. Contrary to other works, we cannot use the semi-classical Fourier transform, as input controls are reused *multiple times* throughout the algorithm. But overall, the space complexity is reduced.

One should note that the gate complexity that we obtain, in $\mathcal{O}(n^3)$, is comparable to an implementation of Shor's algorithm using schoolbook multiplication. If a more efficient multiplication algorithm is used, then Shor's algorithm performs better. So does Regev's recent proposal [31]. However, for a typical size of $n = 2048$ bits, current optimizations of Shor's algorithm still use schoolbook multiplication.

Regarding practical gate counts, Gidney and Ekerå [21] report $2^{31.3}$ Toffoli gates to factor RSA-2048 in one run. Our current estimates give 2^{36} Toffoli gates for a single circuit, which is higher but comparable, for 1633 qubits including ancillas, instead of 6144. However, this is the cost for a single run in the Ekerå-Håstad algorithm. In order to succeed, between 2^5 and 2^6 runs are required, which brings the total Toffoli gate count closer to 2^{42}, and the difference is by a factor of a thousand.

Relation with Circuit Complexity. Computing the "compressed" modular exponentiation $(a^x \bmod N) \bmod 2$ in $o(n)$ space can be done by combining several results of circuit complexity in a black-box way, although this does not seem to have been noticed before in the context of Shor's algorithm. However, this may lead to a large polynomial, which will be useless in practice: a dedicated analysis is required to bring this polynomial down to a reasonable $\mathcal{O}(n^3)$.

As noticed above, modular exponentiation is first reduced to a controlled multi-product of n integers. In [2] Beame, Cook and Hoover showed how to perform a product of n integers in logarithmic depth, as well as modular reduction: this was the basis of Cleve and Watrous' log-depth algorithm for factoring, which also introduced a log-depth implementation of the Quantum Fourier Transform [8]. These techniques use the RNS.

Since the circuit for modular multi-product has depth $\mathcal{O}(\log n)$, the first output bit can be written as the root node in a binary tree of size $n^{\mathcal{O}(\log n)} = \mathsf{poly}(n)$, where the leaves are the input bits (in our case, the control bits for the exponentiation in Shor's algorithm) and each node specifies a Boolean operation. All the nodes in the tree (and *a fortiori* the root) can be computed in time $\mathsf{poly}(n)$ and using $\mathcal{O}(\log n)$ space using a depth-first exploration. Using Bennett's time-space tradeoff [4], we can make this computation reversible, it will still run in time $\mathsf{poly}(n)$ and using space $\mathcal{O}\left(\log^2 n\right)$. This applies as well to discrete logarithms in $\mathbb{Z}_p^*$, which essentially rely on the same circuit.

While we are interested in computing multi-products, which are more relevant to quantum factoring, we note that the RNS was also used for the case of modular exponentiation by Bernstein and Sorenson [6]. These results, which concern depth optimizations on parallel machines, rely on an *explicit* version of Chinese remaindering which we reuse here in a different context.

Regarding discrete logarithms on elliptic curves, one can already notice that the best depth available in the literature is $\mathcal{O}\left(\log^2 n\right)$ [32], which leads this generic compression to a superpolynomial complexity $2^{\mathcal{O}\left(\log^2 n\right)} = n^{\mathcal{O}(\log n)}$. The main difference with discrete logarithm in $\mathbb{Z}_p^*$ seems to be the availability of the RNS. For this reason, achieving a polynomial-time workspace compression for the case of elliptic curves remains an open question.

Outline. In Section 2, we provide basic preliminaries of notation, arithmetic and quantum and reversible circuits. Section 3 introduces Shor's algorithm in more detail, recalls the Ekerå-Håstad variant and the May-Schlieper compression technique and explains how our approximate exponentiation circuit will fit in with these results. This algorithm is detailed in Section 4, purely as a classical computation. In Section 5 we report on preliminary cost estimates for this algorithm.

2 Preliminaries

In this section, we give some useful notation and basic preliminaries of arithmetic and quantum algorithms.

2.1 Notation

Throughout this paper we need to go back and forth from numbers and their representation as bits, as this is the main way we will be able to "compress" the computations in less working qubits than previous implementations of Shor's algorithm. We will adopt the following conventions.

Let $n \in \mathbb{N}$ be the complexity parameter, typically the bit-size of the RSA modulus we are trying to factor. We will work with integers of bit-size either polylogarithmic in n (typically $\mathcal{O}(\log n)$), which will be denoted by lowercase letters, or bit-size polynomial in n (typically from $\mathcal{O}(n)$ to $\mathcal{O}\left(n^2\right)$ bits), for which we use uppercase. We make this difference to emphasize that integers with $\mathsf{polylog}(n)$ bits can be written in temporary registers of our circuits, because they take negligible additional space with respect to the $\mathcal{O}(n)$-qubit input register for the circuit. However integers with $\mathsf{poly}(n)$ qubits take non-negligible space to write down.

We use $[A]_p$ to denote $A \mod p$, i.e., the remainder in the Euclidean division of A by p. We use $(A)_i$ to denote the i-th bit of A, where bit 0 is the least significant bit, i.e.: $A = \sum_i (A)_i 2^i$.

2.2 Modular Reduction and RNS

At one point in our algorithm, we need to perform modular reduction of a large number by a large modulus. For this we use a specific form of Barrett reduction.

Proposition 1. *Let N be a fixed modulus. let $k \in \mathbb{N}$, then for any integer $A < 2^k$, we have:*

$$0 \leq A - \left\lfloor \frac{A \lfloor 2^k/N \rfloor}{2^k} \right\rfloor N < 2N \ . \tag{1}$$

Furthermore, suppose that A is not a multiple of N, and that $2^k \geq AN$. Then we have:

$$\left\lfloor \frac{A}{N} \right\rfloor = \left\lfloor \frac{A \lfloor 2^k/N \rfloor}{2^k} \right\rfloor \ . \tag{2}$$

Proof. The proof follows trivially from the fact that $x \geq \lfloor x \rfloor > x - 1$ for all x. The first inequality is trivial since $\left\lfloor \frac{A\lfloor 2^k/N \rfloor}{2^k} \right\rfloor \leq \frac{A}{N}$. This also implies that $\left\lfloor \frac{A\lfloor 2^k/N \rfloor}{2^k} \right\rfloor \leq \left\lfloor \frac{A}{N} \right\rfloor$, since it's an integer. Next:

$$\forall A < 2^k, \left\lfloor \frac{A \lfloor 2^k/N \rfloor}{2^k} \right\rfloor > \frac{A \lfloor 2^k/N \rfloor}{2^k} - 1 > \frac{A}{N} - \frac{A}{2^k} - 1 \geq \left\lfloor \frac{A}{N} \right\rfloor - 2 \ .$$

This implies that the approximated quotient can be underestimated by at most 1. However, if we make the additional assumption that A is not a multiple of N, then we have: $\frac{A}{N} \geq \left\lfloor \frac{A}{N} \right\rfloor + \frac{1}{N}$. Therefore:

$$\left\lfloor \frac{A \lfloor 2^k/N \rfloor}{2^k} \right\rfloor > \left\lfloor \frac{A}{N} \right\rfloor + \frac{1}{N} - \frac{A}{2^k} - 1 \ . \tag{3}$$

In particular, setting $2^k \geq AN$ gives us $\left\lfloor \frac{A\lfloor 2^k/N \rfloor}{2^k} \right\rfloor > \left\lfloor \frac{A}{N} \right\rfloor - 1$ which implies the equality. □

RNS. Residue number systems (RNS) have been used to obtain small-depth circuits, e.g. in [2] for logarithmic-depth circuits for multi-products. While our goal is orthogonal (to obtain circuits with larger depth but small memory), there is a relation between them, as we have stated in the introduction: circuits with small depth can be turned into algorithms with small space. Therefore, it is not a surprise that our factoring algorithm relies on the same method, although we need to use it differently to minimize the gate count in our setting.

A RNS uses a basis of ℓ prime moduli, which we denote by $P := \{p_1, \dots p_\ell\}$, and represents a large integer A by its residues modulo the p_i. Indeed, let $M = \prod_{p \in P}^{p}$, then by the Chinese Remainder Theorem we know that there is a bijection:

$$A \mod M \mapsto [A]_{p_1}, \dots, [A]_{p_\ell} \tag{4}$$

Furthermore, this bijection can be effectively computed as follows. For each $p \in P$, let $M_p = M/p$ and $w_p = (M_p)^{-1} \mod p$. Then for any $A < M$:

$$A = \left[\sum_{p \in P} [A]_p w_p M_p \right]_M \quad . \tag{5}$$

The Prime Number Theorem gives us the asymptotic behavior of the prime counting function $\pi(x)$ (the number of primes smaller than x): $\pi(x) \simeq \frac{x}{\log x}$. This means that, asymptotically, $\pi(x) = \frac{x}{\log x} + o\left(\frac{x}{\log x}\right)$ and that the total number of primes between x and $2x$ is of order $\frac{2x}{\log(2x)} - \frac{x}{\log x} = \mathcal{O}\left(\frac{x}{\log(x)}\right)$.

A consequence of this fact is that for any integer n, there exists a set of $\mathcal{O}\left(\frac{n}{\log n}\right)$ of primes of size $\mathcal{O}(\log n)$ bits, which can be used to represent numbers of n bits.

Lemma 1 (Consequence of the Prime Number Theorem). *For any integer $n \geq 8$, there exists a set of prime numbers $p_1, \ldots, p_\ell$ such that $\forall i, p_i < n$ and $\prod_i p_i > 2^n$. Furthermore $\ell = \mathcal{O}(n/\log n)$.*

Explicit CRT. It seems to be folklore knowledge that the quotient in the Euclidean division by M, which appears during the recombination step, can also be computed from the residues with the following formula:

$$q_M := \left\lfloor \sum_{p \in P} [A]_p \frac{w_p M_p}{M} \right\rfloor = \left\lfloor \sum_{p \in P} [A]_p \frac{w_p}{p} \right\rfloor \quad .$$

While early works [27] suggest to simply use floating-point arithmetic with sufficiently low rounding error, Bernstein [5,6] gives an explicit formula to compute the quotient with fixed-point arithmetic. We will follow very similar steps in Subsection 4.2. Although our formula differs slightly from Lemma 3.1 in [5] (and Theorem 2.2 in [6]), it would have been possible to use this Lemma immediately at this step.

2.3 Quantum Algorithms

We describe quantum algorithms using the quantum circuit model, and refer to [28] for a detailed definition including qubits, quantum states, the ket $|\cdot\rangle$ notation and the Euclidean distance $\|\cdot\|$. We emphasize that we study only *logical* quantum circuits, which are hardware-agnostic. We do not analyze how the obtained circuits should be mapped to a physical architecture, with additional costs of routing and distillation of magic states for certain gates.

We analyze the costs of our circuits as follows. The *width* is the number of qubits required. This always includes *ancilla* qubits, which are qubits whose state is initialized to $|0\rangle$ and returned to $|0\rangle$ afterwards in order to reuse them for several parts of the computation. The *depth* is the amount of time steps required to run the circuit if independent gates can be applied in parallel. The *gate count* is the total number of gates.

While this work is motivated by quantum algorithms, the operation that we analyze is classical. Indeed, the entire quantum procedure of Shor's algorithm performs a QFT, a modular exponentiation, another QFT, and measures. Asymptotically as well as in practice, the bulk of the cost comes from the modular exponentiation circuit. Unless one uses quantum-specific arithmetic circuits (like Draper's QFT adder [10]), which will not be our case, the circuit can be described in classical reversible logic. In this setting, the NOT, CNOT and Toffoli form a universal set of gates, which we will use for our more detailed counts.

The algorithm that we consider requires several measurement results: the same circuit is run multiple times (we use alternatively the terms of "multiple runs" or "multiple measurements" since these are the same for us). While this number of measurements is a factor in the total gate count, it is often considered more practical to separate a quantum computation into more sub-computations of smaller complexities, as it increases the overall robustness to errors: this is the idea behind Regev's trade-off on quantum factoring [31]. Therefore, for practical applications it might be more interesting to focus on the depth of an individual circuit, and to ensure that the algorithm keeps working if individual measurements can fail with non-negligible probability.

3 Preliminaries on Shor's Algorithm

We recall the main results that we use regarding Shor's algorithm [34], the Ekerå-Håstad variant [16], the precise analysis of [11] and its compressed version [26].

3.1 Shor's Algorithm

Shor's algorithm is undoubtedly the most discussed quantum cryptanalysis algorithm to date. Our depiction in this section follows May and Schlieper [26], as it allows to cover quickly the original version, but also the Ekerå-Håstad variant [16] and its compression.

Period-finding Subroutine. We will start with Shor's period-finding algorithm in $\mathbb{Z}$. Consider a function $f\ : \mathbb{Z} \to \{0,1\}^n$ admitting a hidden period r: $\forall z, f(z+r) = f(z)$ of known bit-size. Shor's algorithm finds the period r by calling once or several times the following subroutine Q_f^{shor}:

1. Initialize an *input register* of size m and a *workspace register* of size n: $|0^m\rangle\,|0^n\rangle$
2. Create a uniform superposition in the input register, using Hadamard gates: $\frac{1}{\sqrt{2^m}}\sum_{z=0}^{2^m-1}|z\rangle\,|0^n\rangle$
3. Compute f into the output register: $\frac{1}{\sqrt{2^m}}\sum_{z=0}^{2^m-1}|z\rangle\,|f(z)\rangle$
4. Apply a Fourier transform QFT_{2^m} on the input register
5. Measure and return the value of the input register

The final measurement outputs a value z which is close to some multiple of $\frac{2^m}{r}$. By choosing m large enough, r can be found from a constant number of measurements using a classical post-processing based on continued fractions. Shor proposed $m = 2n$ [34], where n is the bit-size of N.

In the case of factoring a number N, the function is defined as: $f(z) = a^z \bmod N$ where a is a number prime with N. It is periodic of period $s = \mathrm{ord}(a)$, the multiplicative order of a modulo N. If $\mathrm{ord}(a)$ is even, one has: $a^{\mathrm{ord}(a)} = 1 \bmod N \implies (a^{\mathrm{ord}(a)/2} - 1)(a^{\mathrm{ord}(a)/2} + 1) = 1 \bmod N$. Assuming that $a^{\mathrm{ord}(a)/2} - 1 \neq 0 \bmod N$, it suffices to compute a GCD between $a^{\mathrm{ord}(a)/2} - 1$ and N to find one of its prime factors (since we focus on RSA keys, this is enough for us). Shor proved that the probability of achieving such a split is at least $1 - 2^{1-k}$, when N admits k distinct odd prime factors, which is $\geq \frac{1}{2}$ for RSA integers.

In the case of the Discrete Logarithm Problem in an Abelian group $(\mathcal{G}, \cdot)$, one can consider the function defined over $\mathbb{Z}^2$ by: $f(x, y) = g^x \cdot a^{-y}$ where a is the element considered and g is a generator of the group. Since we can first find the order of the elements, we can restrict the function to a finite group $\mathbb{Z}_{k_1} \times \mathbb{Z}_{k_2}$, which simplifies the post-processing. The input register is separated into inputs for x and y and the last QFT is replaced by a product of two independent QFTs on these registers. (Note that we could also run this algorithm without knowing the order of a, but also, that we generally solve the DLP in a group of prime order, in which the order of elements is therefore already known).

Compression. May and Schlieper [26] studied the behavior of quantum period-finding algorithms, among which Shor's and Simon's [35], when the output of the function f is *post-processed* by a hash function which reduces its size. They observed that the algorithm works almost in the same way, even if h compresses the output down to a *single bit*, which will be useful for us. This is due to the following properties.

Definition 1. *A hash function family $\mathcal{H}_t = \{h : \{0,1\}^n \to \{0,1\}^t\}$ is universal if for all $x, y \in \{0,1\}^n, x \neq y$, one has: $\Pr_{h \in \mathcal{H}_t}[h(x) = h(y)] = 2^{-t}$.*

Theorem 1 (Theorem 7 in [26]). *Let $f : \{0,1\}^q \to \{0,1\}^n$ and $\mathcal{H}_t$ be a universal hash function family. Let Q_f^{period} be a quantum circuit that on input $|0^m\rangle |0^n\rangle$ yields a superposition:*

$$|\Phi\rangle = \sum_{y \in \{0,1\}^m} \sum_{f(x) \in Im(f)} w_{y,f(x)} |y\rangle |f(x)\rangle \tag{6}$$

satisfying

$$\forall y \neq 0, \sum_{f(x) \in Im(f)} w_{y,f(x)} = 0 \ . \tag{7}$$

Let $p(y)$, resp. $p_h(y)$, be the probability to measure $|y\rangle, y \neq 0$ in the m input qubits when applying Q_f^{period}, resp. $Q_{h \circ f}^{\text{period}}$ with h selected u.a.r. from $\mathcal{H}_t$. Then $p_h(y) = (1 - 2^{-t}) p(y)$.

Equation 7 is called the *cancellation criterion*. This criterion is satisfied by Shor's order-finding algorithm, but more generally by quantum period-finding of the same family, like Simon's [35]. The cases that are useful for us are explored in [26]. When this property is satisfied, Theorem 1 shows that we can hash the output register, i.e., eliminate most information from it. Measuring the input register then yields a similar distribution as before, where the probability to measure 0 increases (up to 1/2 when $t = 1$) and the other probabilities are rescaled accordingly. Measuring 0 reveals nothing about the period; we can just discard these results and do more measurements to succeed.

Lemma 2 (Lemma 6 in [26]). *The Q_f^{shor} subroutine has the cancellation property Equation 7.*

A corollary is that the "hashed" version of Shor's algorithm succeeds with two calls to $Q_{h\circ f}^{\text{shor}}$ on average, where the standard version would have needed one call to Q_f^{shor}. However, how to compute $h \circ f$ in a space-efficient way (without essentially computing f, then h) was left as an open question in [26].

In practice, using a single hash function instead of a universal family should give similar results, and we can expect the truncation to the first bit to be a good hash function. We will not attempt to prove this but instead encapsulate this idea as a heuristic.

Heuristic 1. The result of Theorem 1 holds when using the single hash function h defined by truncating to the first bit. That is, the distribution of outputs y is similar to a rescaled version of the one for the uncompressed algorithm.

3.2 Ekerå-Håstad variant

The Ekerå-Håstad algorithm [16] is a variant of Shor's which reduces the number of input qubits for computing short discrete logarithms.

In the Ekera-Håstad circuit the input qubits are separated into $(t+\ell)+\ell$ where t is the bit-size of the discrete logarithm and ℓ determines the post-processing time. To compute the discrete logarithm of $a = g^d$ in a group $\mathcal{G}$, with generator g, the function f is defined as: $f(x, y) = g^x a^{-y} = g^{x-dy}$. Here x is of $(t + \ell)$ bits and y is of ℓ bits, where $\ell = \lceil \frac{m}{s} \rceil$ for some ratio s.

Different from Shor's algorithm, one runs the $Q_f^{\text{Ekerå-Håstad}}$ subroutine a total of $\mu > s$ times, followed by a lattice-based classical post-processing algorithm. Such a procedure is presented and analyzed in detail in [11]. For the case where one wants to do a single run, a refined analysis is given in [14], but we are interested in increasing s as much as possible to reduce the space, and therefore, we consider multiple runs.

In [11], Ekerå shows that the number of measurements necessary tends asymptotically to $s + 1$ when s is fixed and n tends to infinity. More precisely, for the post-processing it should be enough to perform $s + 1$ runs and solve a CVP instance in a lattice of dimension $s + c$, where c is some constant, and $c = 1$ is expected to be enough asymptotically. By using $s = \mathcal{O}(\log n)$ we ensure that this post-processing can be performed in polynomial time in n, using a lattice sieving algorithm of complexity $2^{\mathcal{O}(\log n)}$. This will give a quantum space complexity $\frac{n}{2} + \mathcal{O}\left(\frac{n}{\log n}\right) = \frac{n}{2} + o(n)$. For parameters relevant to cryptography (e.g., RSA-2048), the post-processing routine was entirely simulated in [11]. We will reuse these results for our concrete complexity estimates.

The Ekerå-Håstad method can also be compressed. Since the compression only alters the probability to measure 0, and the zeroes are discarded, the subroutine has simply to be called twice more. The post-processing time is unchanged.

Lemma 3 (Lemma 7 in [26]). *The $Q_f^{\text{Ekerå-Håstad}}$ subroutine has the cancellation property Equation 7.*

Ekerå-Håstad for RSA Factorization. Although we presented the method for the discrete logarithm case, it yields an interesting optimization for factorization of RSA moduli. Consider $N = P \times Q$ where N is of bit-size n, but P and Q are both of bit-size $n/2$. Let $\tilde{P} = \frac{P-1}{2}$ and $\tilde{Q} = \frac{Q-1}{2}$. Select a random element G invertible modulo N and let R be its order: $G^R = 1 \mod N$. Then R divides $2\tilde{P}\tilde{Q}/\gcd(\tilde{P},\tilde{Q})$.

We define $f(N) = (N-1)/2 - 2^{n/2-1} = 2\tilde{P}\tilde{Q} + \tilde{P} + \tilde{Q} - 2^{n/2-1}$. Let $H := G^{f(N)} = G^D$ for some $D < R$. Then: $D = f(N) \mod R = \tilde{P} + \tilde{Q} - 2^{n/2-1} \mod R$. This means that the discrete logarithm D is small.

By computing the discrete logarithm of H, one obtains $\tilde{P} + \tilde{Q} - 2^{n/2-1} \mod R$, which we can assume to be equal to $\tilde{P} + \tilde{Q} - 2^{n/2-1}$, i.e., one obtains $P + Q$. Since $N = PQ$ is also known, we can deduce P and Q.

One should note that, while the post-processing is very different, the Ekerå-Håstad circuit for factorization remains very similar to the Shor circuit. Indeed, the Shor subroutine computes $[A^X]_N$ by a series of n modular multiplications by precomputed powers of A. The Ekerå-Håstad circuit computes $G^X H^{-Y}$, which is also done by modular multiplications. The number of multiplications to perform is $\frac{n}{2} + 2\left\lceil \frac{n}{2s} \right\rceil$ where s is the ratio above, determined by the runtime of the classical post-processing.

For a 2048-bit RSA modulus N, Table 3 in [11] shows that $s = 17$ can be used, $\mu = 20$ runs only are required and the classical post-processing is efficient. In that case the input register has bit-size $m = \frac{2048}{2} + 2\left\lceil \frac{2048}{34} \right\rceil = 1146$.

3.3 Approximate Arithmetic

It has been observed in previous works that Shor's algorithm does not require an exact modular exponentiation circuit. In fact, it is enough to return a good result with large probability on a random instance. Zalka first proposed such a strategy in [37] and coined the term "deterministic errors". Further, the *coset representation of modular integers* [38] is both approximate and inherently quantum, as it represents integers modulo N using a superposition. In [18], Gidney introduced the technique of *oblivious carry runways.* These techniques have been used in state-of-the-art resources estimates of quantum factoring [21].

Analysis. The analysis we give here is specific to our case: we have an erroneous circuit computing a "compressed" version of the output, and we need multiple measurements. From now on, we write h instead of $h \circ f$ for the compressed periodic function with a single-bit output.

We analyze a version of the compressed period-finding subroutine where instead of implementing h, we implement a function h' such that $h'(x) = h(x)$ with probability $1 - p$ only. We do not know a priori on which inputs the function is implemented correctly, but we assume that these inputs are selected uniformly at random (in particular, these errors should not interact with the rest of the algorithm).

We can bound the difference between the "real" and "ideal" runs of compressed-period using the Euclidean distance between quantum states before the last QFT operation and measurement. Indeed, if we define:

$$|\psi_r\rangle = \frac{1}{2^{m/2}} \sum_{z=0}^{2^m-1} |z\rangle \, |h(z)\rangle \text{ and } |\psi_i\rangle = \frac{1}{2^{m/2}} \sum_{z=0}^{2^m-1} |z\rangle \, |h'(z)\rangle$$

then:

$$\| \, |\psi_e\rangle \, \|^2 := \| \, |\psi_r\rangle - |\psi_i\rangle \, \|^2 = (p2^m)\frac{4}{2^m} = 4p \ , \tag{8}$$

and, since the last QFT step is a unitary:

$$\|(I \otimes QFT_{2^m})(|\psi_r\rangle) - (I \otimes QFT_{2^m})(|\psi_i\rangle)\| = \| \, |\psi_r\rangle - |\psi_i\rangle \, \| = 2\sqrt{p} \ . \tag{9}$$

Consider the Ekerå-Håstad algorithm with the post-processing routine of [11], in its "compressed version". We perform 3μ runs of the algorithm before finding μ measurement results which are not zero, and which we can feed to the post-processing routine. We can bound the total distance between the real and ideal runs as follows:

$$\|(|\psi_r\rangle)^{\otimes 3\mu} - (|\psi_i\rangle)^{\otimes 3\mu}\| = \sqrt{3\mu} \cdot 2\sqrt{p} \ . \tag{10}$$

Therefore, the total variation distance between the probability distributions resulting from measuring the ideal and the real runs is bounded by: $4\sqrt{3\mu}2\sqrt{p}$ by Lemma 3.6 in [7].

Let p_i be the probability to succeed in the "ideal" run, i.e., the probability that all measurement results are good. Then the probability to succeed in the "real" run is lower bounded by:

$$p_r \geq p_i - 4\sqrt{3\mu}2\sqrt{p} \ . \tag{11}$$

For our RSA-2048 example, the success probability given in [11] is 0.99 and we need 20 measurements. In the compressed version, using $60 \simeq 2^{5.91}$ measurement results for good measure, we are ensured that at least 20 of them are non-zero with probability ≥ 0.9969. The inequality becomes:

$$p_r \geq 0.99^2 - 2^{6.75}\sqrt{p} \ . \tag{12}$$

That is, by selecting $p \leq 2^{-20}$, we are ensured to succeed with overwhelming probability.

3.4 Overview of Related Works for Factoring

Since Shor's original paper, many authors have attempted to optimize the algorithm in terms of depth, gate count or qubit usage. Since we focus here on space requirements, we list in Table 1 the number of qubits required by different implementations. We emphasize that these results concern the abstract circuit model where only the qubit and gate counts are considered.

It is well known that the complexity of implementing Shor's algorithm depends crucially on the multiplication circuit that is used. Since multiplication of n-bit integers can be done in $\mathcal{O}(n \log n)$ bit operations [24], there exists a quantum circuit for multiplying n-bit integers using $\mathcal{O}(n \log n)$ gates and qubits, and some of the gate counts given in the literature follow this asymptotic rule. However, this circuit is inapplicable for 2048-bit RSA moduli (especially if we want to optimize the space). Other circuits such as fast Karatsuba multiplication [19] improve the gate count significantly (for large numbers) while keeping the space at $\mathcal{O}(n)$, but this also comes at the expense of a larger constant.

So far, when aiming for the space complexity, the only multiplication circuits that have been used are either schoolbook multiplication (combined with more advanced techniques such as windowed arithmetic [20] or coset representation of modular integers [38]) or QFT-based addition [10] and multiplication circuits. These require at least $\mathcal{O}(n^2)$ gates.

Very recently, Kahanamoku-Meyer and Yao proposed a new efficient multiplication circuit that could be suitable for small-size numbers, and requires only few ancilla qubits [25]. While the paper is not available at the time of writing we include this result in Table 1 for completeness.

Table 1. Quantum space-optimized factoring algorithms. The input is an n-bit number. The gate count is for the entire algorithm, as ours (last lines) requires $\mathcal{O}(\log n)$ independent runs. The difference between RSA and general integers comes from the variant of Shor's algorithm and post-processing that can be applied (see Subsection 4.6 for the general case).

Algorithm	Qubits	Gates (Toffolis)
[3]	$2n + 3$	$\mathcal{O}(n^3 \log n)$
[36]	$2n + 2$	$\mathcal{O}(n^3 \log n)$
[23]	$2n + 2$	$\mathcal{O}(n^3 \log n)$
[17]	$2n + 1$	$\mathcal{O}(n^3 \log n)$
[38]	$1.5n + \mathcal{O}(1)$	$\mathcal{O}(n^3 \log n)$
[21]	$3n + 0.002n \log n$	$0.3n^3 + 0.0005n^3 \log n$
[25]	$2n + \mathcal{O}(\log n)$	$\mathcal{O}(n^{2.4})$
This work (RSA)	$\frac{n}{2} + o(n)$	$\mathcal{O}(n^3 \log n)$
This work (general integers)	$n + o(n)$	$\mathcal{O}(n^3 \log n)$

Regev's Algorithm. Very recently also, Regev [31] introduced a new quantum factoring which can be regarded as a multi-dimensional variant of Shor's, allowing to reduce the circuit size by a factor $\mathcal{O}(\sqrt{n})$ at the expense of making $\mathcal{O}(\sqrt{n})$ measurements. In particular, if schoolbook multiplication is used, this reduces the gate count of each run from $\mathcal{O}(n^3)$ for variants of Shor's algorithm to $\mathcal{O}(n^{5/2})$. Shortly afterwards, Ragavan and Vaikuntanathan [30] showed how to implement this algorithm with $\mathcal{O}(n)$ qubits, which was a difficulty in [31]. The algorithm was also extended to Discrete Logarithms in $\mathbb{Z}_p^*$ [15]. However, regarding space complexity, Regev's algorithm is worse than previous works, and [30] reports $12.32(1+\varepsilon)n$ qubits (where ε is a small heuristic constant) when schoolbook multiplication is used.

4 Compressed Modular Multi-Product Circuit

In this section, we explain our main result: a space-efficient circuit for computing a *compressed modular multi-product*.

Let $N < 2^n$ be the number to factor. In the case of a generic integer, we may consider the standard version of Shor's algorithm, while in the case of an RSA modulus, we consider the Ekerå-Håstad variant. In both cases, the exponentiation circuit reduces to a *modular multi-product*: we have a fixed sequence of m integers $(A_i)_{0 \leq i \leq m-1}$ modulo N, where $m = \mathcal{O}(n)$. Given an input $X := \sum_{i=0}^{m-1} x_i 2^i$ identified with an m-bit string, our goal is to compute the *multi-product*: $A_X := \prod_{i=0}^{m-1} A_i^{x_i}$, to reduce it modulo N, and to truncate to its first bit (compression).

We design a reversible circuit for this, performing the operation:

$$|X\rangle |0\rangle \mapsto |X\rangle \left| \left[[A^X]_N \right]_2 \right\rangle \quad . \tag{13}$$

using only $\mathcal{O}(n^3)$ gates and $\mathcal{O}\left(\frac{n}{\log n}\right) = o(n)$ ancilla space, for a total depth $\mathcal{O}(n^2(\log n)^3)$ (Theorem 2). This space can even be reduced to $\mathcal{O}(\log n)$, but the depth would increase to $\mathcal{O}(n^3)$.

This circuit *fails on some inputs*. However, under heuristics, the probability of failure can be bounded precisely for a uniformly random input m-bit string.

4.1 Main Idea

The core of our technique is to use an RNS to represent the multi-product A_X, i.e., a number of at most nm bits. We only reduce modulo N once.

RNS Parameters and Notation. Let $P = \{p_1, \ldots, p_\ell\}$ be the prime numbers for the RNS. By Lemma 1 we have $\ell = \mathcal{O}(mn/\log n) = \mathcal{O}(n^2/\log n)$ and $\forall p \in P, p \leq mn$. We let $w := \lceil \log_2 \max_{p \in P} p \rceil$. The product $M := \prod_{p \in P} p$ is upper bounded by: $M < 2^{nm} p_\ell \leq 2^{nm+w}$. Indeed, by definition of the RNS, we have $\prod_{i=0}^{\ell-1} p_i < 2^{nm} \implies M < 2^{nm} p_\ell$. Thus M can get bigger than 2^{nm} due to the last prime factor, but only by a small number of bits. Finally, we define: $\alpha := \left\lceil \log_2 \left(\sum_{p \in P} p \right) \right\rceil$. In particular $\alpha \leq w + \lceil \log_2 \ell \rceil$.

For all p, let $M_p = M/p$ (which is an integer) and $w_p = (M_p)^{-1} \bmod p$, and notice that $M_p w_p < M$. Let us define:

$$A'_X := \sum_{p \in P} \underbrace{\Big[A_X \Big]_p}_{w \text{ bits}} \underbrace{M_p w_p}_{mn+w \text{ bits}} \leq 2^{mn+w+\alpha} \quad . \tag{14}$$

From there, we need to perform two layers of modular reduction: modulo M and modulo N, as we have:

$$[A_X]_N = [[A'_X]_M]_N \quad . \tag{15}$$

We define two quantities which are the quotients of both Euclidean divisions:

$$\begin{cases} q_M := \lfloor A'_X / M \rfloor = \left\lfloor \frac{\sum_{p \in P} [A_X]_p M_p w_p}{M} \right\rfloor \\ Q_N := \lfloor [A'_X]_M / N \rfloor = \left\lfloor \frac{\left(\sum_{p \in P} [A_X]_p M_p w_p\right) - q_M M}{N} \right\rfloor \end{cases} \tag{16}$$

We notice here that q_M is a small number, which is why we used a lowercase letter for the notation. Indeed, we have: $A'_X \leq 2^{mn+w+\alpha}$ and $M \geq 2^{nm}$ by definition of the RNS, thus $q_M < 2^{w+\alpha} = \mathsf{poly}(n)$.

Finally, we can express our end result as:

$$\begin{aligned} [[[A'_X]_M]_N]_2 &= [(A'_X - q_M M) - Q_N N]_2 \\ &= [A'_X]_2 \oplus [q_M]_2 \, [M]_2 \oplus [Q_N]_2 \quad . \end{aligned}$$

We can already observe that $[A'_X]_2$ can be computed on the fly, by summing all least significant bits of $[A_X]_p M_p w_p$. Next, we will see how we compute q_M, then $[q_M]_2$ and $[Q_N]_2$. For this, we will make two successive approximations. The first one requires X to have bounded Hamming weight, which is ensured with high probability if X is selected uniformly at random. The second one is probabilistic, and its probability of success is a tunable parameter in our algorithm.

Remark 1. We could write instead:

$$[[[A'_X]_M]_N]_2 = [[A_X]_N]_2 = [A_X]_2 \oplus [Q_N]_2 \quad , \tag{17}$$

and notice that $[A_X]_2 = 0$ iff one of the numbers in the product is even. Computing this exactly requires slightly more work, since we need to maintain a counter of size $\log n$ for the number of even numbers in the product. However, we may also assume that the product will be even with high probability, and simply fail if this is not the case. In all cases, the cost of this computation is insignificant, and we still need to compute q_M as it appears in the expression of Q_N.

The Hamming Weight Constraint on X. The success of our algorithm depends on the Hamming weight of X. We have the following.

Lemma 4. *When X is chosen u.a.r., with probability $2\exp\left(-\frac{4}{6}m^{1/3}\right) = negl(n)$, the following holds:*

$$\begin{cases} hw(X) \leq \frac{m}{2} + m^{2/3} \\ A_X \leq N^{\frac{m}{2}+m^{2/3}} \end{cases} \tag{18}$$

Proof. Since we just need to count the 1-coordinates of X, we use a multiplicative Chernoff bound:

$$\Pr\left(\left|hw(X) - \frac{m}{2}\right| \geq 2m^{-1/3} \times \frac{m}{2}\right) \leq 2\exp\left(-(2m^{-1/3})^2\frac{m}{6}\right)$$
$$\implies \Pr\left(\left|hw(X) - \frac{m}{2}\right| \geq m^{2/3}\right) \leq 2\exp\left(-\frac{4}{6}m^{1/3}\right) \ .$$

The second inequality is a direct implication. □

An immediate consequence of this fact is that we do not need our RNS to represent nm-bit numbers, but only $n(\frac{m}{2} + m^{2/3})$-bit numbers. To be more precise, we modify our definition as follows: we let $P = p_1, \ldots, p_\ell$ be $\mathcal{O}(n^2/\log n)$ prime numbers such that $M = \prod_{p\in P} p \geq 2^{n(\frac{m}{2}+2m^{2/3})}$. The additional factor $2^{nm^{2/3}}$ has a reason which will be explained below.

4.2 Reduction Modulo M and First Approximation

We will now see how we compute q_M. Recall its definition:

$$q_M := \left\lfloor \frac{\sum_{p\in P}[A_X]_p w_p M_p}{M} \right\rfloor = \left\lfloor \sum_{p\in P} [A_X]_p \frac{w_p}{p} \right\rfloor \ . \tag{19}$$

Here p, w_p, $[A_X]_p$ and eventually q_M are all small numbers (bit-size logarithmic in n), so intuitively we may compute this sum directly by approximating $\frac{1}{p}$ by $\frac{1}{2^u}\lfloor 2^u/p \rfloor$, where $u = \mathcal{O}(\log n)$ is chosen appropriately.

We note that the following Lemma is essentially a variant of the Explicit CRT, where we also allow a small probability of failure, which is asymptotically negligible in n. This result in itself is not new, see for example [5,6]. It would be possible here to use the previous Explicit CRT from [5,6], and it would just induce a small change in the formula for q_M.

Lemma 5. *Let $u = \left\lceil \log_2 \sum_{p\in P} p^2 \right\rceil$. Asymptotically in n, with probability $1 - negl(n)$, the following holds:*

$$q_M = \left\lfloor \frac{1}{2^u}\left(\sum_{p\in P}[A_X]_p w_p \lfloor 2^u/p \rfloor\right)\right\rfloor + 1 \ . \tag{20}$$

Proof. Recall that we chose our RNS product M such that $M \geq 2^{n(\frac{m}{2}+2m^{2/3})}$, but with high probability we have $A_X \leq 2^{n(\frac{m}{2}+m^{2/3})}$. As a consequence: $0 < \frac{A_X}{M} \leq 2^{-nm^{2/3}}$. As a consequence:

$$\left\{ \sum_{p \in P} [A_X]_p \frac{w_p}{p} \right\} = \frac{A_X}{M} \leq 2^{-nm^{2/3}} \quad .$$

That is, the fractional part of this sum is very small.

Next, we choose u such that: $2^u > \sum_{p \in P} p^2$. Then we have the following inequalities:

$$\frac{2^u}{p} \geq \left\lfloor \frac{2^u}{p} \right\rfloor + \frac{1}{p} \text{ and } \left\lfloor \frac{2^u}{p} \right\rfloor \geq \frac{2^u}{p} - 1 \quad . \tag{21}$$

On the one hand, we have:

$$\left(\sum_{p \in P} [A_X]_p \frac{w_p}{p} \right) - \left(\sum_{p \in P} [A_X]_p \frac{w_p}{2^u} \right) \leq \frac{1}{2^u} \left(\sum_{p \in P} [A_X]_p w_p \left\lfloor 2^u/p \right\rfloor \right) \tag{22}$$

Therefore, our choice of u ensures that:

$$q_M - 1 \geq \left\lfloor \frac{1}{2^u} \left(\sum_{p \in P} [A_X]_p w_p \left\lfloor 2^u/p \right\rfloor \right) \right\rfloor \quad . \tag{23}$$

On the other hand:

$$\frac{1}{2^u} \left(\sum_{p \in P} [A_X]_p w_p \left\lfloor 2^u/p \right\rfloor \right) \leq \left(\sum_{p \in P} [A_X]_p \frac{w_p}{p} \right) \left(1 - 2^{-u}\right) \quad . \tag{24}$$

Asymptotically in n, we have seen that the fractional part of $\sum_{p \in P} [A_X]_p \frac{w_p}{p}$ is very small, so that, if $u = \mathsf{polylog}(n)$:

$$\left\{ \sum_{p \in P} [A_X]_p \frac{w_p}{p} \right\} < 2^{-u} \left(\sum_{p \in P} [A_X]_p \frac{w_p}{p} \right)$$

Consequently this inequality entails:

$$\frac{1}{2^u} \left(\sum_{p \in P} [A_X]_p w_p \left\lfloor 2^u/p \right\rfloor \right) < \sum_{p \in P} [A_X]_p \frac{w_p}{p} - \left\{ \sum_{p \in P} [A_X]_p \frac{w_p}{p} \right\} = q_M \quad . \tag{25}$$

Thus, the result of the sum needed to be corrected by 1. □

At this point, we notice that q_M can be computed on the fly (by computing the $[A_X]_p$ again) using low space. The value of q_M is important both for having $[q_M]_2$, but also because it appears in Q_N. We can now turn ourselves towards the second step, i.e., the computation of $[Q_N]_2$.

Remark 2. Strictly speaking, this first approximation is not necessary. We could choose an RNS representing $n(m+1)$-bit numbers, and the computation of q_M by Equation 20 would then work with certainty for all inputs X. However, this source of errors is negligible, and the gain in practice is significant enough to justify taking a smaller RNS.

4.3 Reduction Modulo N and Second Approximation

For the computation of $[Q_N]_2$, we will use Barrett reduction. We introduce a parameter t_N to satisfy the condition of Proposition 1, i.e., $2^{t_N} \geq A_X N$, so we can choose $t_N = \lceil n(\frac{m}{2} + m^{2/3}) \rceil + n$. We also remark that:

Lemma 6. *For any choice of X, A_X is not a multiple of N.*

Which follows from the fact that all numbers in the multi-product are powers of A, which is (assumed to be) prime with N. Therefore, Proposition 1 applies and Q_N has the expression:

$$Q_N = \left\lfloor \frac{1}{2^{t_N}} \left(\sum_{p \in P} [A_X]_p M_p w_p \left\lfloor 2^{t_N}/N \right\rfloor - q_M M \left\lfloor 2^{t_N}/N \right\rfloor \right) \right\rfloor \tag{26}$$

We have:

$$[Q_N]_2 = \left(\sum_{p \in P} [A_X]_p M_p w_p \left\lfloor 2^{t_N}/N \right\rfloor - q_M M \left\lfloor 2^{t_N}/N \right\rfloor \right)_{t_N} , \tag{27}$$

i.e., we need to compute the bit at position t_N in this quantity, where q_M has been computed at the previous step. For convenience, we replace the difference by a sum, working modulo 2^{t_N+1}. We have:

$$[Q_N]_2 = \left(\sum_{p \in P} [A_X]_p M_p w_p \left\lfloor 2^{t_N}/N \right\rfloor + q_M \left(2^{t_N+1} - M \left\lfloor 2^{t_N}/N \right\rfloor \right) \right)_{t_N} \tag{28}$$

Intuitively, there is no need to compute the entire sum, as the least significant bits only have a very small influence on the value of the t_N-th bit. To view this however, we first decompose the modular residues $[A_X]_p$ and q_M on individual bits, and express $[Q_N]_2$ as the t_N-th bit in a large sum:

$$[Q_N]_2 = \Bigg(\sum_{p \in P} \sum_i \left([A_X]_p\right)_i 2^i M_p w_p \left\lfloor 2^{t_N}/N \right\rfloor + \sum_i (q_M)_i 2^i \left(2^{t_N+1} - M \left\lfloor 2^{t_N}/N \right\rfloor \right) \Bigg)_{t_N}$$

Therefore we can see $[Q_N]_2$ as a sum of $\leq \sum_{p \in P} \lceil \log_2 p \rceil + \lceil \log_2 q_M \rceil$ integers. How many depends on the current values of $[A_X]_p$ and q_M (but we prefer to overestimate). These integers behave as if they were drawn uniformly at random[1].

We introduce a parameter u' and approximate $[Q_N]_2$ by truncating these integers to u' bits, and taking the bit u' in the sum:

$$[Q_N]_2 \simeq \Bigg(\sum_{p \in P} \sum_i ([A_X]_p)_i \left\lfloor 2^i M_p w_p \left\lfloor 2^{t_N}/N \right\rfloor / 2^{t_N - u'} \right\rfloor \\ + \sum_i (q_M)_i \left\lfloor 2^i \left(2^{t_N+1} - M \left\lfloor 2^{t_N}/N \right\rfloor\right) / 2^{t_N - u'} \right\rfloor \Bigg)_{u'}$$

The amount of precision we need is related to the number of integers we are summing. In a sum of β random integers we need a precision of at least $\lceil \log_2 \beta \rceil$ bits to succeed with constant probability. The reason is the same as in *oblivious carry runways* [18]. When truncating with $\lceil \log_2 \beta \rceil + \nu$ bits, the probability that carries propagate all the way from the truncated part of the integers to the wanted bit (and the bit is flipped as a result) is $2^{-\nu}$.

This is what happens if the integers are drawn at random. This is not the case for us, as these integers are precomputed cofactors depending on the RNS. Experimentally, we observe that this property remains satisfied, and formulate it as a heuristic.

Heuristic 2. When choosing $u' = \left\lceil \log_2 (\sum_{p \in P} \lceil \log_2 p \rceil + \lceil \log_2 q_M \rceil) \right\rceil + \nu$, for random inputs X, the truncation at Step 2 succeeds with probability $\geq 1 - 2^{-\nu}$.

At this point, the only subroutine that remains is the computation of $[A_X]_p$ for primes p in the RNS. This computation can be done in $\mathcal{O}(\log n)$ space and time $\mathcal{O}(m \log^2 n)$ using a series of m multiplications modulo p. We give another strategy in Subsection 4.5 which is overall more efficient.

4.4 Summary

Thanks to the approximations, both q_M and $[Q_N]_2$ are computed with $o(n)$ space. More precisely, let $\gg$ be a bitwise right-shift, then we have:

$$q_M = \left(\left(\sum_{p \in P} [A_X]_p \underbrace{w_p \left\lfloor 2^u / p \right\rfloor}_{\text{Precomputed}} \right) \gg u \right) + 1 \tag{29}$$

[1] This is not exactly the case because they have some zero LSBs, but since we are looking at the coordinate t_N (which is quite far in the sum), the LSBs are insignificant.

and:

$$[Q_N]_2 = \left(\sum_{p\in P} \sum_i ([A_X]_p)_i \underbrace{\left\lfloor \left\lfloor 2^i M_p w_p \left\lfloor 2^{t_N}/N \right\rfloor / 2^{t_N - u'} \right\rfloor \right\rfloor_{2^{u'+1}}}_{\leq 2^{u'+1} \text{ and precomputed}} \right.$$
$$\left. + \sum_i (q_M)_i \underbrace{\left\lfloor \left\lfloor 2^i \left(2^{t_N+1} - M \left\lfloor 2^{t_N}/N \right\rfloor \right) / 2^{t_N - u'} \right\rfloor \right\rfloor_{2^{u'+1}}}_{\leq 2^{u'+1} \text{ and precomputed}} \right) \gg u'$$

Therefore, we only need to compute large (controlled) sums of precomputed cofactors, depending on the bits of $[A_X]_p$ and q_M. The maximal amount of storage required remains in $\mathcal{O}(\log n)$. The cofactors are the following:

$$\begin{cases} C_{p,i} := \left\lfloor 2^i w_p \left\lfloor 2^u/p \right\rfloor \right\rfloor_{2^{u+\lceil \log_2 q_M \rceil+1}} \\ D_{0,i} := \left\lfloor \left\lfloor 2^i \left(2^{t_N+1} - M \left\lfloor 2^{t_N}/N \right\rfloor \right) / 2^{t_N-u'} \right\rfloor \right\rfloor_{2^{u'+1}} \\ D_{p,i} := \left\lfloor \left\lfloor 2^i M_p w_p \left\lfloor 2^{t_N}/N \right\rfloor / 2^{t_N-u'} \right\rfloor \right\rfloor_{2^{u'+1}} \end{cases} \tag{30}$$

We summarize the resulting algorithm as Algorithm 1, with a layout close to how it will be implemented as a quantum circuit.

Under the heuristics above, our algorithm will use about $\mathcal{O}(n^3)$ gates and succeed with constant probability.

Theorem 2. *Assume that $m = \mathcal{O}(n)$. There exist a reversible logical circuit for compressed multi-product which, on input $(X, 0)$, returns $(X, [[A_X]_N]_2)$, and succeeds:*

- *for inputs X satisfying $hw(X) \leq \frac{m}{2} + m^{2/3}$,*
- *with probability $1 - \varepsilon$ for a chosen $\varepsilon > 0$, under Heuristic 2.*

It uses $\mathcal{O}(n^3)$ gates, depth $\mathcal{O}(n^2 \log^3 n)$ and $\mathcal{O}\left(\frac{n}{\log n}\right)$ ancillas.

Proof. The circuit layout follows Algorithm 1.

Throughout the computation we maintain a register for q_M, a register for $[A'_X]_2$ and a register for Q_N. Each time a new residue is computed, we perform $\mathcal{O}(\log n)$ controlled additions by constants in the registers for q_M and Q_N. We also update the register for $[A'_X]_2$ with a CNOT. By Lemma 1, there are $\mathcal{O}(n^2/\log n)$ primes and we compute each residue only once. We use the multi-product circuit of Lemma 8, either with the sequential sum or its parallel version.

Once the whole sum for q_M has been computed, we shift it by u bits and increment. We have obtained q_M.

We use the bits of q_M to complete the sum for Q_N, by making new controlled additions for each one with their own precomputed cofactors. This step is computationally negligible with respect to the other additions that we just performed.

When we have finished the sum for Q_N, we select its u'-th bit, XOR it with $[A'_X]_2$ and output the result. Then, we erase the accumulator registers by recomputing the residues and performing a series of controlled subtractions.

Algorithm 1 Approximate multi-product algorithm.

```
    Input: $X$
    Output: $[[A_X]_N]_2$
    Precomputed: cofactors $C_{p,i}, D_{0,i}, D_{p,i}$
 1: qM ← 0                                        ▷ value of $q_M$
 2: QN2 ← 0                                       ▷ value of $[Q_N]_2$
 3: AXp2 ← 0                                      ▷ value of $[A'_X]_2$
 4: for all $p$ in the RNS do
 5:     $x \leftarrow [A_X]_p$                    ▷ using the subroutine
 6:     $x := x_0 + 2x_1 + ...$
 7:     for all $0 \leq i \leq \lceil \log_2 p \rceil$ do
 8:         if $x_i = 1$ then
 9:             qM ← qM + $C_{p,i}$   ▷ The computation is done on $u + \lceil \log_2 q_M \rceil + 1$ bits
10:             QN2 ← QN2 + $D_{p,i}$           ▷ The computation is done on $u' + 1$ bits
11:         end if
12:     end for
13:     AXp2 ← AXp2 $\oplus [x]_2[M_p w_p]_2$
14: end for
15: qM ← qM $\gg u$
16: qM ← qM + 1
17: AXp2 ← AXp2 $\oplus$ [qM]$_2[M]_2$
18: $q_M := b_0 + 2b_1 + \ldots$
19: for all $0 \leq i \leq \lceil \log_2 q_M \rceil$ do
20:     if $b_i = 1$ then
21:         QN2 ← QN2 + $D_{0,i}$
22:     end if
23: end for
24: QN2 ← QN2 $\gg u'$
25: Return AXp2 $\oplus$ [QN2]$_2$
```

Since the accumulator registers are also of logarithmic size in n, the circuit uses $\mathcal{O}(\log n)$ working bits in addition of those required for the RNS residues. The gate count is dominated by the computation of the residues, which we do using Lemma 8. □

Remark 3. Since the size of intermediate registers is logarithmic, the dominating space complexity term $\frac{n}{\log n}$ comes entirely from our implementation of the multi-sum circuit in Lemma 8. This can be reduced down to $\mathcal{O}(\log n)$ qubits by sacrificing the depth.

4.5 Modular Multi-Product in the RNS

The main problem with the computation of $[A_X]_p$ is its sequentiality. Indeed, if we follow the blueprint of modular multi-product in Shor's algorithm, we do a sequence of m controlled modular multiplications by $[A_i]_p$. This takes depth $\widetilde{\mathcal{O}}(m)$, and gates are applied only on a work register of size $\mathcal{O}(\log p) = \mathcal{O}(\log n)$, while the control qubits remain idle for the most part.

In the following, since we work at low scales, the asymptotic formulas that we give assume the use of schoolbook addition and multiplication circuits: on $\log n$ bits, addition costs $\mathcal{O}(\log n)$ gates and depth, and multiplication costs $\mathcal{O}\big(\log^2 n\big)$ gates and depth.

Our first improvement on the naive multi-product is to replace the controlled modular multiplications by a sequence of controlled additions.

Lemma 7. *Let $a_0, \ldots, a_{m-1} \in \mathbb{Z}_p$, and p be a prime, with $m = \mathcal{O}(n)$ and $p = \mathcal{O}(n)$. There exists a circuit performing the multi-product modulo p:*

$$|x_0, \ldots, x_{m-1}\rangle\, |0\rangle \mapsto |x_0, \ldots, x_{m-1}\rangle\, |[\prod_i a_i^{x_i}]_p\rangle \quad ,$$

using $\mathcal{O}(m \log n)$ gates and $\mathcal{O}(m \log n)$ depth.

Proof. We choose a generator g of $\mathbb{Z}_p^*$ and precompute the discrete logarithms of the a_i modulo p. Let $\alpha_i \leq p-1$ be such that: $a_i \equiv g^{\alpha_i} \bmod p$. Then we have:

$$\prod_i a_i^{x_i} \equiv g^{\sum_{i=0}^{m-1} x_i \alpha_i} \bmod p \ . \tag{31}$$

Instead of requiring $\mathcal{O}\big(m \log^2 n\big)$ gates, we will need:

- $\mathcal{O}(m \log n)$ gates for computing $\sum_{i=0}^{m-1} x_i \alpha_i$, which is smaller than $(p-1)m$ (thus on $\mathcal{O}(\log n)$ bits)
- a $\mathcal{O}(\log n)$-bit exponentiation circuit modulo p: we precompute the $g^{2^i} \bmod p$, we read off the $\mathcal{O}(\log n)$ bits of $\sum_{i=0}^{m-1} x_i \alpha_i$, and perform $\mathcal{O}(\log n)$ modular multiplications in gate count: $\mathcal{O}\big(\log^3 n\big)$.

Only $\mathcal{O}(\log n)$ ancilla space is needed for these two operations. □

However this circuit is still sequential. We modify it further by optimizing the multi-sum, which is the dominating cost.

Lemma 8. *There exists a circuit for the multi-product modulo p using $\mathcal{O}(m \log n)$ gates, $\mathcal{O}\Big(\frac{m}{\log n}\Big)$ ancilla qubits and depth $\mathcal{O}\big(\log^4 n\big)$.*

Proof. We change the way we compute $\sum_{i=0}^{m-1} x_i \alpha_i$, as follows. We select a parameter $2 < k \leq m$ and cut the numbers into k groups.

For each group, we compute its sum using an addition tree. As there are (m/k) numbers to add, the tree has depth $\log_2(m/k)$. We need $\mathcal{O}\big(\frac{m}{k} \log n\big)$ ancilla qubits to write all its nodes (starting from the first numbers), and $\mathcal{O}\big(\frac{m}{k} \log n\big)$ gates to compute all of them.

The result of each smaller sum is added into an accumulator register, which contains the result of the whole sum.

Since there are k groups, the total depth is: $\mathcal{O}(k \times \log(m/k) \times \log n) = \mathcal{O}\big(k (\log n)^2\big)$ and the total gate count is: $\mathcal{O}(m \log n)$.

By selecting $k = (\log n)^2$, we achieve a space in $\mathcal{O}\Big(\frac{m}{\log n}\Big)$ and a depth $\mathcal{O}\big((\log n)^4\big)$ (which becomes dominating), for the same gate count. □

Remark 4. The choice of $\frac{m}{\log n}$ ancillas is not entirely arbitrary: the goal is to match the asymptotic space increase of the Ekerå-Håstad algorithm (see Subsection 3.2).

Since $\mathcal{O}\big((\log n)^4\big)$ is still quite a large factor for a typical $n = 2048$, we propose an alternative optimization which gives slightly better results in practice, even though it increases the asymptotic gate count.

Lemma 9. *There exists a circuit for the multi-product mod p using $\mathcal{O}\big(m \log^2 n\big)$ gates, $\mathcal{O}\Big(m\frac{\log\log n}{\log n}\Big) = o(m)$ ancilla qubits and depth $\mathcal{O}\big(\log^3 n\big)$.*

Proof. Let us decompose α_i as a sequence of bits: $\forall i, \alpha_i = \sum_k (\alpha_i)_k 2^k$. We then have:

$$\sum_{i=0}^{m-1} x_i \alpha_i = \sum_k \left(\sum_{i=0}^{m-1} x_i (\alpha_i)_k \right) 2^k \ . \tag{32}$$

A strategy to compute the sum in $\mathcal{O}(\log n)$ steps follows: we initialize a $\mathcal{O}(\log n)$-qubit accumulator. At each step, we compute $\sum_{i=0}^{m-1} (\alpha_i)_k$, shift the value by k bits and add it to our accumulator.

We now focus on the computation of $\sum_{i=0}^{m-1} x_i (\alpha_i)_k$ for a fixed k. Since $(\alpha_i)_k$ are precomputed values, this is equivalent to computing $\sum_{i \in I} x_i$ for a fixed subset $I \subseteq \{0, \ldots, m-1\}$. To do this with a minimal use of ancillas, we adopt the following strategy.

We cut the bits of I (less than m) into groups of size $k = \mathcal{O}(\log n)$. To each of these groups, we append an ancillary register of $\mathcal{O}(\log\log n)$ qubits. We use a sequence of $\log n$ in-place incrementor circuits to sum the bits into this register one by one. Then, we sum these ancillary registers two by two. Each time we sum two registers, we use an addition circuit and append two more qubits for the inevitable carries. The total ancilla space is therefore:

$$\frac{m}{k} \log k + \frac{m}{k} \left(1 + \frac{1}{2} + \ldots \right) = \mathcal{O}\left(m \frac{\log\log n}{\log n} \right) \ .$$

The first step (local and sequential) uses $\frac{m}{\log n} \times \log n \times \log n = \mathcal{O}(m \log n)$ gates and depth $\mathcal{O}\big(\log^2 n\big)$. The second step (global and parallel) needs depth $\mathcal{O}\big(\log \frac{m}{k} \times \log n\big) = \mathcal{O}\big(\log^2 n\big)$ as well under the same assumption, and uses $\mathcal{O}(m)$ gates.

Finally, we need to do this $\mathcal{O}(\log n)$ times. The gate count increases to $\mathcal{O}\big(m \log^2 n\big)$ and the total depth is $\mathcal{O}\big(\log^3 n\big)$, which is the same as the modular exponentiation circuit. $\square$

4.6 Other Applications

While the main application of our method remains factorization of RSA moduli with the Ekerå-Håstad method, other applications of Theorem 2 are immediate. Essentially any application of Shor's algorithm where the group exponentiation can be reduced to an integer multi-product, and which uses an input register of size $m = \mathcal{O}(n)$, can be performed in time $\mathcal{O}(n^3)$ with $m + o(n)$ qubits.

One of these cases is the factorization of general integers, where the space can be reduced to $n + o(n)$ as follows.[2]

First, one uses the classical reduction from [12], which factors an integer N completely given the order of a random element g in $\mathbb{Z}_N^*$. To solve this order-finding problem, one uses Seifert's variant of Shor's algorithm [33] combined with the analysis of App. A in [13]. At this point, the input register length is of $m + \lceil m/s \rceil$ where $m \leq n$ is an upper bound on the bit-length of the order of g, and about s runs are required to find the entire order.

Finally, we can use the exact same procedure to compute discrete logarithms in prime fields. For the short discrete logarithm setting we can use the Ekerå-Håstad method again (here we refer to [11] for more details).

5 Experiments

Since Algorithm 1 is a classical algorithm, we tested it with the RSA-2048 instance of the RSA factoring challenge. The code of our experiments is available at:

https://gitlab.inria.fr/capsule/quantum-factoring-less-qubits

Success Probability. We used $\nu = 20$ additional bits for Heuristic 2 (truncation of the sum). This means that we expect our algorithm to succeed with probability $\geq 1 - 2^{-20}$ on random inputs. Following the discussion in Subsection 3.2 and Subsection 3.3, we take $s = 17$ in the Ekerå-Håstad variant, i.e., we perform $60 = 2^{5.91}$ runs of the algorithm, find 20 results which are non-zero, and use the post-processing of Ekerå [11] to factor N with success probability close to 1. This means that $m = 1146$ in the multi-product circuit.

Note that the number of independent runs and measurements is not negligible and should be factored in the total time complexity, especially at these scales.

RNS. By Lemma 4, we chose to limit the RNS to represent numbers of size $N^{m/2+m^{2/3}}$, i.e., 1 398 784 bits. We chose primes of similar bit-sizes for more uniformity, and chose 72 199 primes of bit-sizes 19 to 21 (included). Their product is of size 1 398 786 bits. The most important resulting parameters are given in Table 2.

5.1 Classical Implementation

We implemented Algorithm 1 on a desktop computer, by separating the pre-computation and the actual computation. We verified that on randomly chosen inputs X, the algorithm succeeds with large probability. Our implementation is quite slow, and computes about one value per second. This allowed us to verify Heuristic 2 for larger failure probabilities (but not for 2^{-20}).

[2] The details that follow here were explained to us by Martin Ekerå.

Table 2. Parameters for running our algorithm with RSA-2048.

Parameter	Value
$s, \ell = \left\lceil \frac{n}{2s} \right\rceil, m = \frac{n}{2} + 2\ell$	17, 61, 1146
Number of primes in the RNS	72 199
Maximal bit-size of primes in the RNS	21
u	57
u'	41
$\lceil \log_2 q_M \rceil$	57
Size of q_M register $= \lceil \log_2 q_M \rceil + u + 1$	115
Size of Q_N register $= u' + 1$	42

5.2 Quantum Implementation

We did not write down entirely the quantum circuit for this instance, as it has above 2^{30} gates and our implementations in Python would not support this. However, we estimated the gate count and depth from its basic building blocks using Qiskit [29]. This is a preliminary estimate which might still be significantly improved.

The main computational cost (in gates and in depth) of our circuit is located in the computation of $[A_X]_p$ for all primes p in the RNS. The circuit is constructed from a precomputed list $a_0 = [A_0]_p, \ldots, a_{m-1} = [A_{m-1}]_p$ and computes:

$$|x_0, \ldots, x_{m-1}\rangle |0\rangle \mapsto |x_0, \ldots, x_{m-1}\rangle |[\prod a_i^{x_i}]_p\rangle$$

by summing the discrete logarithms d_i of the a_i relative to an arbitrary generator of $\mathbb{Z}_p^*$:

$$|x_0, \ldots, x_{m-1}\rangle |0\rangle \mapsto |x_0, \ldots, x_{m-1}\rangle |\sum_i (x_i d_i)\rangle$$

and then computing a small modular exponentiation.

We implemented a circuit for this operation following the blueprint of Lemma 9, computed its costs on random instances and took the average.

Basic Arithmetic Operations. Our circuit relies on arithmetic operations like additions, multiplications or Euclidean divisions on less than 60 bits. We built them from a simple in-place sequential adder from [9] (and a controlled version of it, when necessary). This adder is suited to optimize the qubit count, since it requires at most one ancilla.

Multi-Sum. As expected, most of the cost in the multi-product comes from the multi-sum circuit. However, the small exponentiation modulo p is also not negligible.

In the multi-sum circuit, we needed 21×2 multi-bit sums (once for each bit forwards and backwards). For each multi-bit sum, we used groups of 15 bits and a 4-bit controlled incrementor circuit using an ancilla qubit. This circuit was optimized in an *ad hoc* manner.

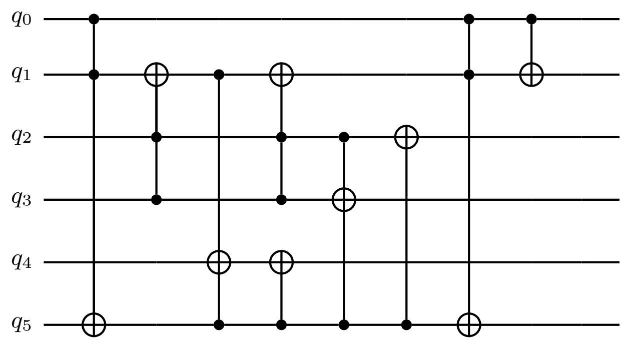

Fig. 2. 4-bit controlled incrementor circuit. Qubit q_0 is the control; qubit q_5 is the ancilla.

As it can be seen in Figure 2, our incrementor contains 6 Toffoli and 3 CNOT gates, and it has depth 8. Each multi-bit sum first sums the bits inside each group of 15 inputs, calling 15 times this circuit in sequence. Then, it uses a tree of additions. Counting only the incrementors, we would need: $(2 \times 21) \times (1146/2) \times 9 = 216\,594$ gates, assuming that we only have $(1146/2)$ bits to sum at each level (half of them being 0). Besides, the depth should be around $2 \times 21 \times 8 \times 15 = 5040$.

Trying some random instances, we obtain gate counts around 160 000 Toffolis and 110 000 CNOTs, and a depth around 17 000. The difference in depth comes from the sequential adders, both in the addition trees and also during the computation of the sum.

Modular Exponentiation. For the modular exponentiation circuit, we need to compute: $|x\rangle |0\rangle \mapsto |x\rangle |[a^x]_p\rangle$ where a is a predetermined constant, $x \leq p-1$ and p is about 21 bits. We could use a sequence of 21 modular multiplications for this, but we obtained better results using windowed arithmetic. We cut the exponent x into three parts: $x = x_1 + 2^7 x_2 + 2^{14} x_3$, and we compute $[a^{x_1}]_p, [(a^{2^7})^{x_2}]_p$ and $[(a^{2^{14}})^{x_3}]_p$ using table lookup circuits (see e.g. [1]). Afterwards, we do non-modular multiplications and Euclidean divisions to reduce modulo p. This allows us to compute the full modular exponentiation in typically less than 50 000 gates. However, the depth is large (and mostly comes from the table lookups).

Modular Multi-Product. The modular multi-product circuit uses both components. While the exact costs depend on the a_i and on p, we tried a handful random choices and observe that the variations are quite small at these scales.

Full Quantum Circuit. The quantum circuit follows the layout given in Algorithm 1. We need to keep track of the qM, QN2 and AXp2 registers, which represent:

$$(u + \lceil \log_2 q_M \rceil + 1) + (u' + 1) + 1 = 158 \text{ qubits.}$$

Table 3. Average counts of modular multi-product. The operation considered here is the computation of $[A_X]_p$, and the multi-sum of discrete logarithms is kept along to avoid an unnecessary level of uncomputation.

Qubits (incl. ancilla)	Toffoli	CNOT	X	Depth
1475 (329)	$2^{17.47}$	$2^{16.99}$	$2^{10.13}$	$2^{15.28}$

The cost in depth and gates of the sums is negligible with respect to the modular multi-product sub-circuits. For each prime of the RNS, we will compute a modular multi-product, update the accumulators, and then uncompute it. When we have computed the output bit, we copy it, and then we uncompute everything to erase the accumulators. This means that we compute $72\,199 \times 4 = 288\,796 = 2^{18.14}$ modular multi-product circuits.

Table 4. Expected gate and qubit counts for our RSA-2048 instance. These are the costs for a single exponentiation circuit: 60 runs must be performed in the compressed Ekerå-Håstad algorithm.

Qubits (incl. ancilla)	Toffoli	CNOT	X	Depth
1633 (487)	$2^{35.61}$	$2^{35.13}$	$2^{28.27}$	$2^{33.42}$

We summarize the costs of the full quantum circuit in Table 4. These numbers count a single run of the algorithm. As we have remarked in Subsection 3.3, since we use the Ekerå-Håstad variant we need $60 \simeq 2^{5.91}$ independent runs.

5.3 Future Optimizations and Remarks

The main desirable improvement of our algorithm would be a reduction in depth. While the depth is asymptotically $\mathcal{O}\big(n^2 \times \mathsf{polylog}(n)\big)$, in practice, the factor $\mathsf{polylog}(n)$ is far from negligible. A first step in this direction would be to use shallower circuits for arithmetic operations (e.g., additions), and to optimize these circuits for small-scale operations, since we typically sum numbers of 20 to 60 bits.

Next, the trade-off between depth and width needs to be analyzed in a more refined way. Again, while our algorithm requires $\frac{n}{2} + o(n)$ qubits only, in practice, the $o(n)$ cannot be neglected. A better algorithm for the multi-sum operation could significantly reduce the ancilla overhead of our current implementation.

Finally, some arithmetic circuits make use of dirty ancillas (which start in an uncontrolled state, and are returned to this state). Since we are essentially computing with $o(n)$ qubits, and keeping the $\frac{n}{2}$ controls along the way, such components would certainly be more suited to our algorithm.

6 Conclusion

In this paper, we reduced the number of logical qubits for quantum factoring to $\frac{n}{2} + o(n)$ for the case of RSA moduli. In particular, the number of qubits goes below the size of the RSA modulus, and for RSA-2048 we could propose a circuit with less than 1700 qubits. More generally, the space required for factoring will depend on the *input register* rather than the workspace register, as the latter can be compressed down to size $\mathcal{O}(\log n)$.

While this result may be counter-intuitive at first sight, it follows from a classical algorithm. Using an arithmetic circuit based on the RNS, we can compute (efficiently) a single bit of a modular exponentiation on n bits using $o(n)$ space. This realizes the *compression* initially proposed by May and Schlieper [26]. Then, we can use the Ekerå-Håstad algorithm for computing short discrete logarithms [16,11]. This algorithm reduces the size of the input to $\frac{n}{2} + o(n)$ bits in the case of RSA moduli. Since the input register is all that remains in terms of asymptotic complexity, this technique becomes particularly important in our context.

The arithmetic circuit itself contains $\mathcal{O}\big(n^3\big)$ gates and makes few errors. Our preliminary estimates indicate that the constant factor in the $\mathcal{O}$ is small. Compared to state-of-the art benchmarks for quantum factoring [21], our circuit currently increases the gate count by a factor about 2^5 for a single run. However, multiple runs are needed for the Ekerå-Håstad method, so overall the factor is about a thousand. Since our circuit architecture differs from previous ones, further work is required to understand precisely the possible optimizations and trade-offs.

An interesting question is whether our strategy could be applied to the recent work of Regev [31], who modified Shor's algorithm to split the work into $\sqrt{n}+4$ independent runs (while keeping the overall gate count similar). Current estimates [30] indicate that the space requirement in Regev's algorithm, while it can be made linear in n, is quite large. Therefore, it would be even more useful to compress the work registers in this case. However, the classical subroutine in Regev's algorithm is a *multi-exponentiation* instead of a single one, which cannot be reduced easily to a multi-product. This is the same reason for which the algorithm needs a quantum exponentiation circuit where the *input* is quantum and not only the exponent. Finally, another prominent target of Shor's algorithm is the discrete logarithm problem on elliptic curves. Since the notion of RNS does not exist in their case, they remain outside the scope of our work. Whether there is another way to compress their workspace register (while keeping the circuit size polynomial in n) is an interesting open question.

Acknowledgments. The authors would like to express their gratitude to Martin Ekerå for many insightful comments and remarks, which improved significantly the first version of this paper. We also thank Daniel J. Bernstein for pointing us to the references [5,6] for the Explicit CRT. A.S. wants to thank Xavier Bonnetain for discussions on Shor's algorithm. This work has been supported by the French Agence Nationale de la Recherche through the France 2030 program under grant agreement No. ANR-22-PETQ-0008 PQ-TLS.

References

1. Babbush, R., Gidney, C., Berry, D.W., Wiebe, N., McClean, J., Paler, A., Fowler, A., Neven, H.: Encoding electronic spectra in quantum circuits with linear t complexity. Physical Review X **8**(4), 041015 (2018). https://doi.org/10.1103/PhysRevX.8.041015
2. Beame, P., Cook, S.A., Hoover, H.J.: Log depth circuits for division and related problems. SIAM J. Comput. **15**(4), 994–1003 (1986). https://doi.org/10.1137/0215070
3. Beauregard, S.: Circuit for Shor's algorithm using $2n + 3$ qubits. arXiv preprint quant-ph/0205095 (2002)
4. Bennett, C.H.: Time/space trade-offs for reversible computation. SIAM J. Comput. **18**(4), 766–776 (1989). https://doi.org/10.1137/0218053
5. Bernstein, D.J.: Multidigit modular multiplication with the explicit chinese remainder theorem. Chapter 4 in "Detecting perfect powers in essentially linear time, and other studies in computational number theory", Ph.D. dissertation, University of California at Berkeley (1995), https://cr.yp.to/papers/mmecrt-19950518-retypeset20220326.pdf
6. Bernstein, D.J., Sorenson, J.P.: Modular exponentiation via the explicit chinese remainder theorem. Math. Comput. **76**(257), 443–454 (2007). https://doi.org/10.1090/S0025-5718-06-01849-7
7. Bernstein, E., Vazirani, U.V.: Quantum complexity theory. SIAM J. Comput. **26**(5), 1411–1473 (1997)
8. Cleve, R., Watrous, J.: Fast parallel circuits for the quantum fourier transform. In: FOCS. pp. 526–536. IEEE Computer Society (2000). https://doi.org/10.1109/SFCS.2000.892140
9. Cuccaro, S.A., Draper, T.G., Kutin, S.A., Moulton, D.P.: A new quantum ripple-carry addition circuit. arXiv preprint quant-ph/0410184 (2004)
10. Draper, T.G.: Addition on a quantum computer. arXiv preprint quant-ph/0008033 (2000)
11. Ekerå, M.: On post-processing in the quantum algorithm for computing short discrete logarithms. Des. Codes Cryptogr. **88**(11), 2313–2335 (2020). https://doi.org/10.1007/S10623-020-00783-2
12. Ekerå, M.: On completely factoring any integer efficiently in a single run of an order-finding algorithm. Quantum Inf. Process. **20**(6), 205 (2021). https://doi.org/10.1007/S11128-021-03069-1, https://doi.org/10.1007/s11128-021-03069-1
13. Ekerå, M.: Quantum algorithms for computing general discrete logarithms and orders with tradeoffs. J. Math. Cryptol. **15**(1), 359–407 (2021). https://doi.org/10.1515/JMC-2020-0006, https://doi.org/10.1515/jmc-2020-0006
14. Ekerå, M.: On the success probability of the quantum algorithm for the short DLP. CoRR **abs/2309.01754** (2023). https://doi.org/10.48550/ARXIV.2309.01754, https://doi.org/10.48550/arXiv.2309.01754
15. Ekerå, M., Gärtner, J.: Extending Regev's factoring algorithm to compute discrete logarithms. CoRR **abs/2311.05545** (2023)

16. Ekerå, M., Håstad, J.: Quantum algorithms for computing short discrete logarithms and factoring RSA integers. In: PQCrypto. Lecture Notes in Computer Science, vol. 10346, pp. 347–363. Springer (2017). https://doi.org/10.1007/978-3-319-59879-6_20
17. Gidney, C.: Factoring with $n+2$ clean qubits and $n-1$ dirty qubits. arXiv preprint arXiv:1706.07884 (2017)
18. Gidney, C.: Approximate encoded permutations and piecewise quantum adders. arXiv preprint arXiv:1905.08488 (2019)
19. Gidney, C.: Asymptotically efficient quantum karatsuba multiplication. arXiv preprint arXiv:1904.07356 (2019)
20. Gidney, C.: Windowed quantum arithmetic. arXiv preprint arXiv:1905.07682 (2019)
21. Gidney, C., Ekerå, M.: How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. Quantum **5**, 433 (2021). https://doi.org/10.22331/q-2021-04-15-433
22. Griffiths, R.B., Niu, C.S.: Semiclassical fourier transform for quantum computation. Physical Review Letters **76**(17), 3228 (1996). https://doi.org/10.1103/PhysRevLett.76.3228
23. Häner, T., Roetteler, M., Svore, K.M.: Factoring using $2n+2$ qubits with toffoli based modular multiplication. arXiv preprint arXiv:1611.07995 (2016)
24. Harvey, D., Van Der Hoeven, J.: Integer multiplication in time $\mathcal{O}(n \log n)$. Annals of Mathematics **193**(2), 563–617 (2021)
25. Kahanamoku-Meyer, G.D., Yao, N.Y.: Fast quantum integer multiplication with very few ancillas (2023), https://gregdmeyer.github.io/files/2023-10-26_Google_Kahanamoku-Meyer.pdf
26. May, A., Schlieper, L.: Quantum period finding is compression robust. IACR Trans. Symmetric Cryptol. **2022**(1), 183–211 (2022). https://doi.org/10.46586/TOSC.V2022.I1.183-211
27. Montgomery, P.L., Silverman, R.D.: An FFT extension to the $p-1$ factoring algorithm. Mathematics of Computation **54**(190), 839–854 (1990)
28. Nielsen, M.A., Chuang, I.: Quantum computation and quantum information (2002)
29. Qiskit contributors: Qiskit: An open-source framework for quantum computing (2023). https://doi.org/10.5281/zenodo.2573505
30. Ragavan, S., Vaikuntanathan, V.: Optimizing space in Regev's factoring algorithm. IACR Cryptol. ePrint Arch. p. 1501 (2023)
31. Regev, O.: An efficient quantum factoring algorithm. CoRR **abs/2308.06572** (2023)
32. Rötteler, M., Steinwandt, R.: A quantum circuit to find discrete logarithms on ordinary binary elliptic curves in depth $o(\log^2 n)$. Quantum Inf. Comput. **14**(9-10), 888–900 (2014). https://doi.org/10.26421/QIC14.9-10-11
33. Seifert, J.: Using fewer qubits in shor's factorization algorithm via simultaneous diophantine approximation. In: CT-RSA. Lecture Notes in Computer Science, vol. 2020, pp. 319–327. Springer (2001). https://doi.org/10.1007/3-540-45353-9_24
34. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997). https://doi.org/10.1137/S0097539795293172
35. Simon, D.R.: On the power of quantum computation. SIAM J. Comput. **26**(5), 1474–1483 (1997)
36. Takahashi, Y., Kunihiro, N.: A quantum circuit for Shor's factoring algorithm using $2n+2$ qubits. Quantum Information & Computation **6**(2), 184–192 (2006)

37. Zalka, C.: Fast versions of Shor's quantum factoring algorithm. arXiv preprint quant-ph/9806084 (1998)
38. Zalka, C.: Shor's algorithm with fewer (pure) qubits. arXiv preprint quant-ph/0601097 (2006)