How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits

Craig Gidney[1, *] and Martin Ekerå[2]
[1] *Google Inc., Santa Barbara, California 93117, USA*
[2] *KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden*
Swedish NCSA, Swedish Armed Forces, SE-107 85 Stockholm, Sweden

(Dated: May 24, 2019)

We significantly reduce the cost of factoring integers and computing discrete logarithms over finite fields on a quantum computer by combining techniques from Griffiths-Niu 1996, Zalka 2006, Fowler 2012, Ekerå-Håstad 2017, Ekerå 2017, Ekerå 2018, Gidney-Fowler 2019, Gidney 2019. We estimate the approximate cost of our construction using plausible physical assumptions for large-scale superconducting qubit platforms: a planar grid of qubits with nearest-neighbor connectivity, a characteristic physical gate error rate of 10^{-3}, a surface code cycle time of 1 microsecond, and a reaction time of 10 microseconds. We account for factors that are normally ignored such as noise, the need to make repeated attempts, and the spacetime layout of the computation. When factoring 2048 bit RSA integers, our construction's spacetime volume is a hundredfold less than comparable estimates from earlier works (Fowler et al. 2012, Gheorghiu et al. 2019). In the abstract circuit model (which ignores overheads from distillation, routing, and error correction) our construction uses $3n + 0.002n \lg n$ logical qubits, $0.3n^3 + 0.0005n^3 \lg n$ Toffolis, and $500n^2 + n^2 \lg n$ measurement depth to factor n-bit RSA integers. We quantify the cryptographic implications of our work, both for RSA and for schemes based on the DLP in finite fields.

1. INTRODUCTION

Peter Shor's introduction in 1994 of polynomial time quantum algorithms for factoring integers and computing discrete logarithms [1, 2] was a historic milestone that greatly increased interest in quantum computing. Shor's algorithms were the first quantum algorithms that achieved an exponential speedup over classical algorithms, applied to problems outside the field of quantum mechanics, and had obvious applications. In particular, Shor's algorithms may be used to break the RSA cryptosystem [3] based on the hardness of factoring integers that are the product of two similarly-sized primes (hereafter "RSA integers"), and cryptosystems based on the discrete logarithm problem (DLP), such as the Diffie-Hellman key agreement protocol [4] and the Digital Signature Algorithm [5].

The most expensive operation performed by Shor's factoring algorithm is a modular exponentiation. Modern classical computers can perform modular exponentiations on numbers with thousands of bits in under a second. These two facts may at first glance appear to suggest that factoring a thousand bit number with Shor's algorithm should only take seconds, but unfortunately (or perhaps fortunately), that is not the case. The modular exponentiation in Shor's algorithm is performed over a superposition of exponents, meaning a quantum computer is required, and quantum hardware is expected to be many orders of magnitude noisier than classical hardware [6–8]. This noise necessitates the use of error correction, which introduces overheads that ultimately make performing reliable arithmetic on a quantum computer many orders of magnitude more expensive than on classical computers [9, 10].

Although Shor's algorithms run in polynomial time, the constant factors hidden by the asymptotic notation are substantial. These constant factors must be overcome, by heavy optimization at all levels, in order to make the algorithms practical.

A. Our contributions and a summary of our results

In this work, we combine several novel and existing optimizations to reduce the cost of implementing Shor's algorithms. The main hurdle to overcome is to implement one or more modular exponentiations efficiently, as these exponentiations dominate the overall cost of Shor's algorithms.

We use the standard square-and-multiply approach to reduce the exponentiations into a sequence of modular multiplications. We apply optimizations to reduce the number of multiplications and the cost of each multiplication.

[*] craiggidney@google.com

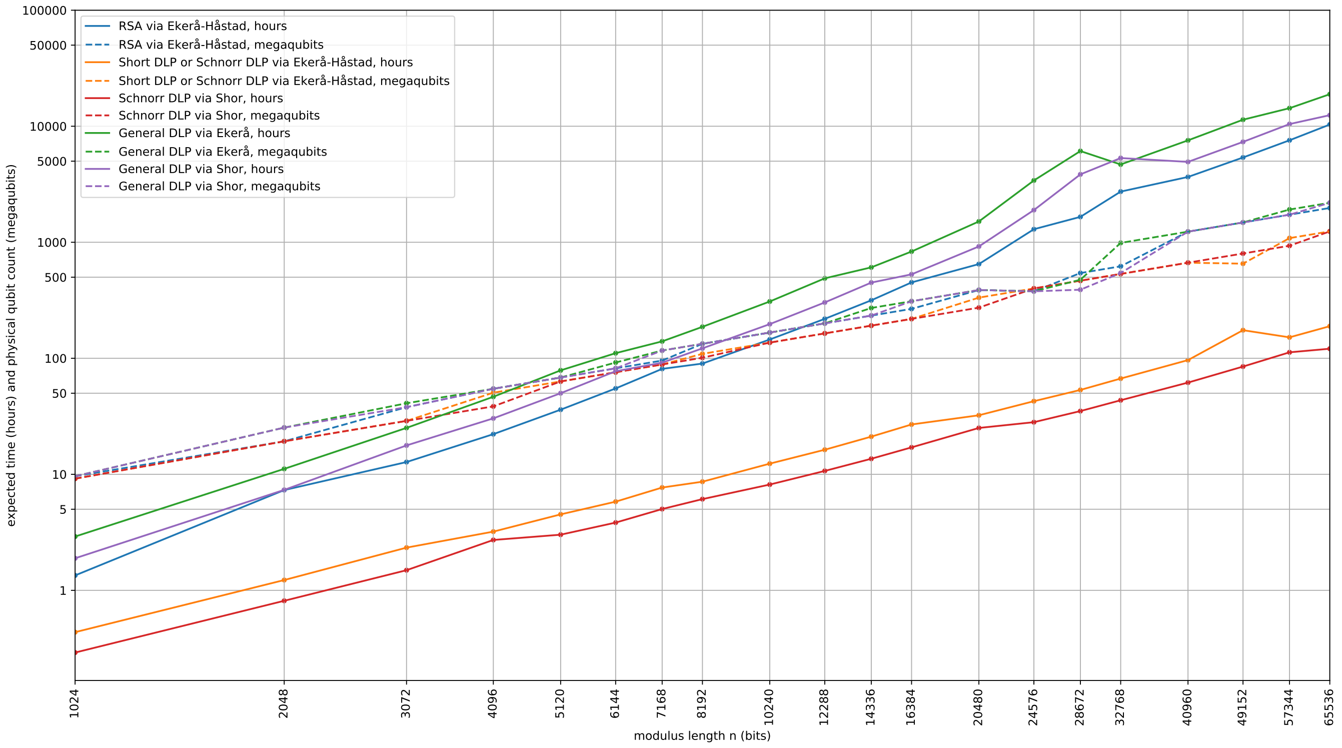


FIG. 1. Log-log plot of estimated space and expected-time costs, using our parallel construction, for various problems and problem sizes. See Section 3 for additional details. The jumps in space around $n = 32786$ occur as the algorithm exceeds the error budget of the CCZ factory from [11] and switches to the T factory from [12]. Generated by ancillary file "estimate_costs.py".

	Abstract Qubits	Measurement Depth	Toffoli+T/2 Count	Toffoli+T/2 Count (billions)			Min Volume (megaqubitdays)		
RSA Factoring Construction	Asymptotic			$n = 1024$	$n = 2048$	$n = 3072$	$n = 1024$	$n = 2048$	$n = 3072$
Vedral et al. 1996 [13]	$7n + 1$	$80n^3 + O(n^2)$	$80n^3 + O(n^2)$	86	690	2300	240	4100	23000
Zalka 1998 (basic) [14]	$3n + O(1)$	$12n^3 + O(n)$	$12n^3 + O(n^2)$	13	100	350	16	250	1400
Zalka 1998 (log add) [14]	$5n + O(1)$	$600n^2 + O(n)$	$52n^3 + O(n^2)$	56	450	1500	16	160	540
Zalka 1998 (fft mult) [14]	$\approx 96n$	$\approx 2^{17}n^{1.2}$	$\approx 2^{17}n^2$	140	550	1200	62	260	710
Beauregard 2002 [15]	$2n + 3$	$144n^3 \lg n + O(n^2 \lg n)$	$576n^3 \lg^2 n + O(n^3 \lg n)$	62000	600000	2200000	32000	380000	1700000
Fowler et al. 2012 [9]	$3n + O(1)$	$40n^3 + O(n^2)$	$40n^3 + O(n^2)$	43	340	1200	53	850	4600
Häner et al. 2016 [16]	$2n + 2$	$52n^3 + O(n^2)$	$64n^3 \lg n + O(n^3)$	580	5200	19000	230	2800	13000
(ours) 2019	$\mathbf{3n + 0.002n \lg n}$	$\mathbf{500n^2 + n^2 \lg n}$	$\mathbf{0.3n^3 + 0.0005n^3 \lg n}$	**0.4**	**2.7**	**9.9**	**0.5**	**5.9**	**21**
ECC Log Construction	Asymptotic			$n = 160$	$n = 224$	$n = 256$	$n = 160$	$n = 224$	$n = 256$
Roetteler et al 2017 [17]	$9n + O(\lg n)$	$448n^3 \lg n + 4090n^3$	$448n^3 \lg n + 4090n^3$	30	84	130	13	52	83

TABLE I. Expected costs of factoring RSA integers using various constructions. Includes an elliptic curve DLP, with similar classical security, for comparison. The estimated minimum spacetime volumes assume modern surface code constructions, even for older papers. See Appendix A for details on each entry in this table.

	Physical assumptions				Approach		Estimated costs		
Historical cost estimate at $n = 2048$	Physical gate error rate	Cycle time (microseconds)	Reaction time (microseconds)	Physical connectivity	Distillation strategy	Execution strategy	Physical qubits (millions)	Expected runtime (days)	Expected volume (megaqubitdays)
Fowler et al. 2012 [9]	0.1%	1	0.1	planar	1200 T	single threaded	1000	1.1	1100
O'Gorman et al. 2017 [18]	0.1%	10	1	arbitrary	block CCZ	single threaded	230	3.7	850
Gheorghiu et al. 2019 [19]	0.1%	0.2	0.1	planar	1100 T	single threaded	170	1	170
(ours) 2019 (1 factory)	0.1%	1	10	planar	1 CCZ	serial distillation	16	6	90
(ours) 2019 (1 thread)	0.1%	1	10	planar	14 CCZ	single threaded	19	0.36	6.6
(ours) 2019 (parallel)	**0.1%**	**1**	**10**	**planar**	**28 CCZ**	**double threaded**	**20**	**0.31**	**5.9**

TABLE II. Historical estimates of the expected costs of factoring $n = 2048$ bit RSA integers, and the assumptions they used. Our spacetime volumes can be directly compared to the volume from Fowler et al. (we achieve a 165x improvement), because Fowler et al's estimate is dominated by distillation and changing the reaction time doesn't affect this volume. It is unclear how to compare O'Gorman et al.'s volume to ours, because of the difference in connectivity. Multiplying the volume from Gheorghiu et al. by 5, to account for the difference in cycle time, allows comparison to our volume (we achieve a 140x improvement). See Appendix B for details on each entry in this table.

The number of multiplications is reduced by using windowed arithmetic [20], which uses small table lookups to fuse several multiplications together. It is also reduced by using Ekerå and Håstad's derivatives [21–24] of Shor's algorithms, that require fewer multiplications to be performed compared to Shor's original algorithms.

The cost of each multiplication is reduced by combining several optimizations. We use Zalka's coset representation of modular integers [25], which allows the use of cheaper non-modular arithmetic circuits to perform modular additions. We use oblivious carry runways [26] to split registers into independent pieces that can be worked on in parallel when performing additions. We bound the approximation error introduced by using oblivious carry runways and the coset representation of modular integers by analyzing them as approximate encoded permutations [26]. We use windowed arithmetic (again) to fuse multiple additions into individual lookup additions [20]. We use a layout of the core lookup addition operation where carry propagation is limited by the reaction time of the classical control system, and where the lookups are nearly reaction limited [27]. Finally, we optimize the overall computation by trying many possible parameter sets (e.g. window sizes and code distances) for each problem size and selecting the best parameter set.

We estimate the approximate cost of our construction, both in the abstract circuit model, and in terms of its runtime and physical qubit usage in an error corrected implementation under plausible physical assumptions for large-scale superconducting qubit platforms with nearest neighbor connectivity (see Figure 1). We provide concrete cost estimates for several cryptographically relevant problems, such as the RSA integer factoring problem, and various parameterizations of the DLP in finite fields. These cost estimates may be used to inform decisions on when to mandate migration from currently deployed vulnerable cryptosystems to post-quantum secure systems or hybrid systems.

Compared to previous works, we reduce the Toffoli count when factoring RSA integers by over 10x (see Table I). To only compare the Toffoli counts as in Table I may prove misleading, however, as it ignores the cost of routing, the benefits of parallelization, etc. Ideally, we would like to compare our runtime and physical qubit usage to previous works in the literature. However, this is only possible when such estimates are actually reported and use physical assumptions similar to our own. The number of works for which this requirement is met is limited.

The works by Fowler et al. [9] and Gheorgiu et al. [19] stand out in that they use the same basic cost model as we use in this paper, enabling fair comparisons to be made. We improve on their estimates by over 100x (see Table II) when accounting for slight remaining differences in the cost model. We also improve upon the cost estimates given by O'Gorman et al. [18], even if a proper comparison is not possible as their physical assumptions are different from ours (they assume slower operations where all qubits are interconnected, which is typical of ion trap architectures, whilst we assume faster operations with nearest neighbor connectivity, which is typical of superconducting architectures).

As most previous works focus either on factoring, or on the DLP in elliptic curve groups, it is hard to find references against which to compare our cost estimates for solving the DLP in multiplicative groups of finite fields. In general, the improvements we achieve for factoring RSA integers are comparable to the improvements we achieve for solving the general DLP in finite fields. As may be seen in Figure 1, the choice of parameterization has a significant impact on the costs. For the short DLP, and the DLP in Schnorr groups, we achieve significant improvements. These are primarily due to our use of derivatives of Shor's algorithm that are optimized for these parameterizations.

B. Notation and conventions

Throughout this paper, we refer to the modulus as N. The modulus is the composite integer to be factored, or the prime characteristic of the finite field when computing discrete logarithms. The number of bits in N is $n = \lceil \lg N \rceil$ where $\lg(x) = \log_2(x)$. The number of modular multiplication operations to perform (i.e. the combined exponent length in the modular exponentiations) is denoted n_e. Our construction has a few adjustable parameters, which we refer to as c_{exp} (the exponent window length), c_{mul} (the multiplication window length), c_{sep} (the runway separation), and c_{pad} (the padding length used in approximate representations).

In the examples and figures, we consider moduli of length $n = 2048$ bits when n needs to be explicitly specified. This is because $n = 2048$ bits is the default modulus length in several widely used software programs [28–30]. Our optimizations are not specific to this choice of n. Section 3 provides cost estimates for a range of cryptographically relevant modulus lengths n.

We often quote costs as a function of both the number of exponent qubits n_e and the problem size n. We do this because the relationship between n_e and n changes from problem to problem, and optimizations that improve n_e are orthogonal to optimizations that improve the cost of individual operations.

C. On the structure of this paper

The remainder of this paper is organized as follows. In Section 2, we describe our construction and the optimizations it uses, and show how to estimate its costs. We then proceed in Section 3 to describe how existing widely deployed cryptosystems are impacted by our work. In Section 4, we present several ideas and possible optimizations that we believe are worth exploring in the future. Finally, we summarize our contributions and their implications in Section 5.

2. OUR CONSTRUCTION

A. Quantum algorithms

In Shor's original algorithm [1], composite integers N that are not pure powers are factored by computing the order r of a randomly selected element $g \in \mathbb{Z}_N^*$. Specifically, period finding is performed against the function $f(e) = g^e$. This involves computing a modular exponentiation with $n_e = 2n$ qubits in the exponent e. If r is even and $g^{r/2} \not\equiv -1$, this yields non-trivial factors of N. To see why, lift g to $\mathbb{Z}$. As $g^r - 1 = (g^{r/2} - 1)(g^{r/2} + 1) \equiv 0 \pmod{N}$ it suffices to compute $\gcd((g^{r/2} \pm 1) \bmod N, N)$ to factor N. The order r is even with probability at least $1/2$.

In [21–23], Ekerå and Håstad explain how to factor RSA integers $N = pq$ in a different way; namely by computing a short discrete logarithm. This algorithm proceeds as follows: First $y = g^{N+1}$ is computed classically, where as before g is randomly selected from $\mathbb{Z}_N^*$ and of unknown order r. Then the discrete logarithm $d = \log_g y \equiv p + q \pmod{r}$ is computed quantumly. To see why $d \equiv p + q \pmod{r}$, note that $\mathbb{Z}_N^*$ has order $\phi(N) = (p-1)(q-1)$ by Euler's totient theorem, so $d \equiv pq + 1 \equiv p + q \pmod{r}$ as r divides $\phi(N)$ and $pq + 1 \equiv p + q \pmod{\phi(N)}$, with equality if $r > p + q$.

For large random RSA integers, the order $r > p+q$ with overwhelming probability. Hence, we may assume $d = p+q$. By using that $N = pq$ and $d = p + q$, where N and d are both known, it is trivial to deterministically recover the factors p and q as the roots of the quadratic equation $p^2 - dp + N = 0$.

The quantum part of Ekerå and Håstad's algorithm is similar to the quantum part of Shor's algorithm, except for the following important differences: there are two exponents e_1, e_2, of lengths $2m$ and m qubits, respectively, for m a positive integer such that $p + q < 2^m$. Period finding is performed against the function $f(e_1, e_2) = g^{e_1} y^{-e_2}$ rather than the function $f(e) = g^e$. The total exponent length is hence $n_e = 3m = 1.5n + O(1)$ qubits, compared to $2n$ qubits in Shor's algorithm. It is this reduction in the exponent length that translates into a reduction in the overall number of multiplications that need to be performed on the quantum computer.

The two exponent registers are initialized to uniform superpositions of all 2^m and 2^{2m} values, respectively, and two quantum Fourier transforms are applied independently to the two registers. This implies that standard optimization techniques, such as the square-and-multiply technique, the semi-classical Fourier transform of Griffiths and Niu [31], recycling of qubits in the exponent registers [32], and so forth, are directly applicable.

The classical post-processing algorithm used to extract the logarithm from the observed frequencies is by necessity different from Shor's. It uses lattice-based techniques, and critically does not require r to be known. Ekerå shows in [23] that the post-processing algorithm has probability above 99% of recovering d. As the factors p and q are recovered deterministically from d and N, this implies that it in general suffices to run the quantum algorithm once.

In summary, Ekerå and Håstad's algorithm is similar to Shor's algorithm from an implementation perspective: a sequence of multiplications is computed, where one operand is in a quantum register and one operand is a classical constant. The multiplications are interleaved with the Fourier transform. It is of no significance to our construction what sequence of classical constants are used, or if one or more independent Fourier transforms need to be applied. This fact, coupled with the fact that the algorithms of Ekerå and Håstad may be used to solve other relevant problems where n_e is a different function of n (see Section 3), lead us to describe our construction in terms of n and n_e.

Note that the above description of the algorithm has been somewhat simplified compared to the algorithm described in [22, 23] in the interest of improved legibility. Furthermore, there are technical conditions that need to be respected for the analysis in [23] to be applicable. For the full details, see section A.2.1 of [23].

B. Reference implementation

To avoid overloading the reader, we will describe our factoring construction by starting from a simple reference implementation of the quantum part of Shor's original algorithm and then apply optimizations one by one.

The reference implementation works the way most implementations of Shor's algorithm do, by decomposing exponentiation into iterative controlled modular multiplication [13–16, 25, 33]. A register x is initialized into the $|1\rangle$ state, then a controlled modular multiplication of the classical constant $g^{2^j} \pmod{N}$ into x is performed, controlled by the qubit e_j from the exponent e, for each integer j from 0 to $n_e - 1$. After the multiplications are done, x is storing $g^e \pmod{N}$ and measuring x completes the hard part of Shor's algorithm.

Controlled modular multiplication is still not a primitive operation, so it must also be decomposed. It can be performed by introducing a work register y initialized to $|0\rangle$ and then performing the following two controlled scaled additions: $y \mathrel{+}= x \cdot k \pmod N$ then $x \mathrel{+}= y \cdot (-k^{-1}) \pmod N$. After these two operations, y is storing the result and x has been cleared to the $|0\rangle$ state. Swapping the two registers, so that the result is in x, completes the multiplication.

Performing a controlled scaled addition with classical scale factor k can be done with a series of n controlled modular additions. For each qubit q_j in the input register, you add $k \cdot 2^j \pmod N$ into the target, controlled by q_j. Controlled modular addition in turn is performed via a series of non-modular additions and comparisons. For example, [13] uses five additions for this purpose. Finally, using the Cuccaro adder [34], uncontrolled non-modular additions can be performed with no additional workspace using $2n$ Toffolis.

Combining the numbers in the preceding two paragraphs implies a Toffoli count of $n_e \cdot 2n \cdot 5 \cdot 2n = 20 n_e n^2$ for the reference implementation.

C. The coset representation of modular integers

Following Zalka [25], the first improvement we make over the reference implementation is to switch from the usual representation of integers to the coset representation of modular integers. The usual representation of an integer k in a quantum computer is the computational basis state $|k\rangle$. The coset representation is different: it uses a periodic superposition with period N and offset k. In ideal conditions, the integer $k \pmod N$ is represented by the state $\sqrt{2^{-c_{\text{pad}}}} \sum_{j=0}^{2^{c_{\text{pad}}}} |jN + k\rangle$ where c_{pad} is the number of additional padding qubits placed at the end of the register. The key idea is that the periodicity of the superposition causes a non-modular adder to perform approximate modular addition on this representation, and the error of the approximation can be exponentially suppressed by adding more padding to the register.

As we will discuss later, the amount of padding required is logarithmic in the number of operations. This small cost enables the large benefit of using non-modular adders instead of modular adders. It is possible to perform a controlled non-modular addition in $4n$ Toffolis [34, 35], significantly cheaper than the $10n$ we assumed for a controlled modular adder. Therefore switching to the coset representation reduces the leading asymptotic term of the Toffoli count of the reference implementation from $20 n_e n^2$ to $8 n_e n^2$.

D. Windowed arithmetic

The next optimization we use is windowed arithmetic. Specifically, we follow the modular exponentiation construction from [20] and use windowing at two levels.

First, at the level of a multiplication, we window over the controlled additions. We fuse groups of controlled additions into single lookup additions. A lookup addition is an addition where the value to add into a register is the result of a table lookup. Small windows of the qubits that would have been used as controls for the additions are instead treated as addresses into classically precomputed tables of values to unconditionally add into the target.

Second, at the level of the exponentiation, we window over the controlled multiplications. This is done by including exponent qubits in the addresses of all lookups being performed within the multiplications. We refer the reader to [20] for the exact details of this nested windowed arithmetic construction.

The cost of windowed arithmetic depends on the size of the windows. Let c_{exp} be the size of the window over exponent qubits that are being used to control multiplications. Let c_{mul} be the size of the window over factor qubits being used to control additions. Using these parameters, the n_e controlled multiplications we needed to perform become n_e / c_{exp} uncontrolled multiplications while the $2n$ controlled additions we needed to perform within each multiplication become $2n / c_{\text{mul}}$ uncontrolled additions. The tradeoff is that each addition is now accompanied by a lookup with an address of size $c_{\text{exp}} + c_{\text{mul}}$.

Using Cuccaro et al's adder [34], each n-bit addition has a Toffoli count and measurement depth of $2n$. Using Babbush et al's QROM read [36], each table lookup has a Toffoli count and measurement depth of $2^{c_{\text{mul}} + c_{\text{exp}}}$. The cost of uncomputing the lookup is negligible due to using measurement based uncomputation [37]. The overhead due to the logarithmic padding introduced by the coset representation of modular integers is also negligible (for now).

Thus, by using windowed arithmetic, the leading term of the Toffoli count of the reference implementation has been reduced to $\frac{2 n_e n}{c_{\text{mul}} c_{\text{exp}}}(2n + 2^{c_{\text{mul}} + c_{\text{exp}}})$. If we set $c_{\text{mul}} = c_{\text{exp}} = \frac{1}{2} \lg n$ then we achieve a Toffoli count with leading term $\frac{24 n_e n^2}{\lg^2 n}$. In terms of space, we are using $3n + O(\lg n)$ logical qubits. The qubits are distributed as follows: $n + O(\lg n)$ for the accumulation register storing the product, $n + O(\lg n)$ for a workspace register during the multiplication, $n + O(\lg n)$ for the lookup output register, and $O(\lg n)$ qubits to hold the part of the exponent needed for the current stage of the semi-classical Fourier transform [31].

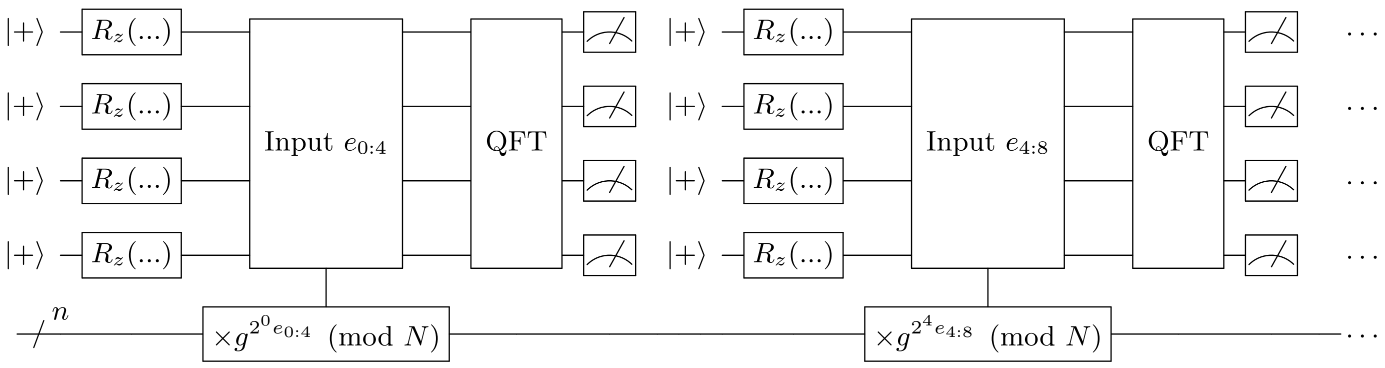

$|+\rangle$
$R_z(\ldots)$
Input $e_{0:4}$
QFT
Input $e_{4:8}$
n
$\times g^{2^0 e_{0:4}} \pmod N$
$\times g^{2^4 e_{4:8}} \pmod N$

FIG. 2. Working through the qubits representing an exponent in Shor's algorithm with a window size c_{exp} of four, while using a semi-classical Fourier transform [31]. Each $R_z(\ldots)$ gate rotates by an amount determined by previous measurements. All 16 possible values of the expression $g^{e_{4:8} 2^4} \pmod N$ (and similar expressions) can be precomputed classically, and looked up on demand within the multiplication circuit. This reduces the number of multiplications by a factor of the window size, at the cost of some additional lookup work within the multiplication.

E. Oblivious carry runways

The last major algorithmic optimization we apply is the use of oblivious carry runways [26]. The basic problem addressed by runways is that, normally, a carry signal generated at the bottom of a register must propagate all the way to the top of the register. This process can be short-circuited by instead terminating the carry propagation into an appropriately constructed runway. Runways allow large additions to be performed piecewise, with each piece being worked on in parallel, by terminating carries into appropriately placed runways at the end of each piece.

As with the coset representation of modular integers, circuits using oblivious carry runways approximate the original circuit instead of perfectly reproducing it. But, as we will discuss later, increasing the runway length c_{pad} exponentially suppresses the approximation error.

The major benefit of oblivious carry runways, compared to previous techniques for reducing the depth of addition such as Draper et al's logarithmic depth carry-lookahead adder [38], is that oblivious carry runways can be introduced gradually without incurring large overheads. The Toffoli count and workspace overheads are linear in the number of pieces $\lceil n/c_{\text{sep}} \rceil$ (where c_{sep} is the runway spacing) but only logarithmic in n. For example, if you place a single runway at the midpoint of a 2048-qubit register, then the number of qubits and the number of Toffolis needed to perform an addition will increase by a couple percent but the depth of the additions is nearly halved.

F. Interactions between optimizations

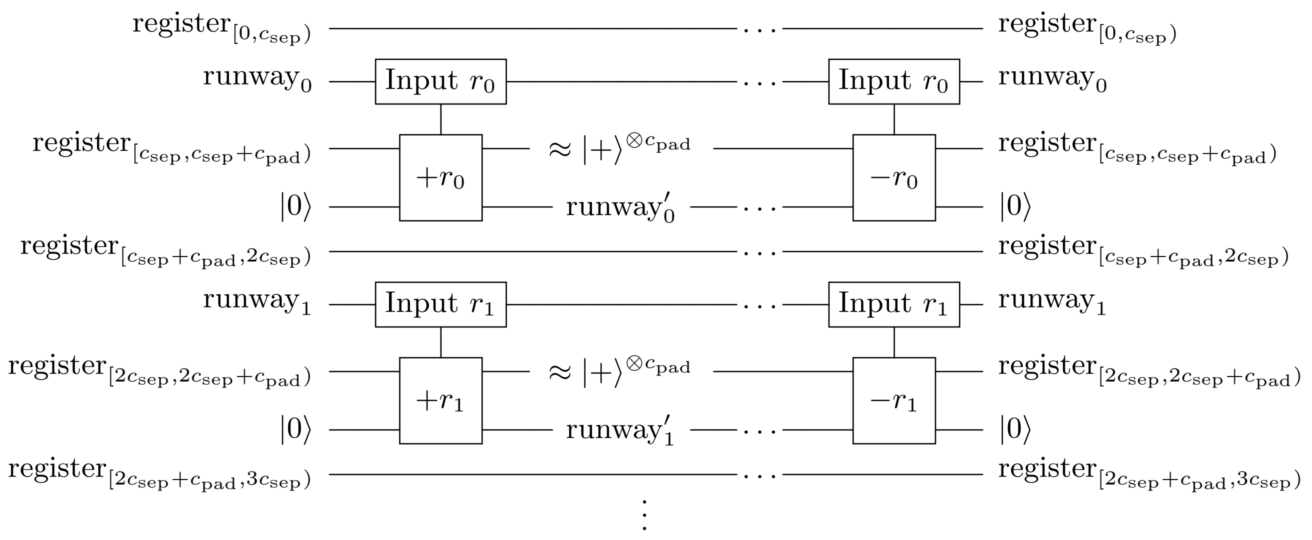
A major benefit of the set of optimizations we have chosen to use in this paper is that they complement each other. They make orthogonal improvements that compound or reinforce, instead of conflicting.

For example, when using the coset representation of modular integers, it is important that additions only use offsets less than N. Larger offsets cause larger approximation error. However, because we are using windowed arithmetic, every addition we perform has an offset that is being returned by a table lookup. Since the entries in the tables are classically precomputed, we can classically ensure all offsets are canonicalized into the $[0, N)$ range.

There are two cases where the optimizations do not mesh perfectly.

First, using oblivious runways reduces the depth of addition but not the depth of lookups. This changes their relative costs, which affects the optimal window sizes to use in windowed arithmetic. When addition is suddenly four times faster than it used to be, the optimal window sizes c_{exp} and c_{mul} decrease by 1 (approximately).

Second, when iterating over input qubits during a multiply-add, it is necessary to also iterate over runway qubits and padding qubits. This increases the number of lookup additions that need to be performed in order to complete a windowed multiplication. This issue is partially solved by temporarily adding the runways back into the main register before performing a multiply-add where that register is used as one of the factors (instead of the target). This temporarily reduces each runway to a single carry qubit (see Figure 3). We could reduce the runways to zero qubits, but this would require propagating the carry qubits all the way across the register, so we do not.



$\text{register}_{[0,c_\text{sep})}$
runway_0
Input r_0
$\text{register}_{[c_\text{sep},c_\text{sep}+c_\text{pad})}$
$+r_0$
$\approx |+\rangle^{\otimes c_\text{pad}}$
$|0\rangle$
runway'_0
$-r_0$
$\text{register}_{[c_\text{sep}+c_\text{pad},2c_\text{sep})}$
runway_1
Input r_1
$\text{register}_{[2c_\text{sep},2c_\text{sep}+c_\text{pad})}$
$+r_1$
runway'_1
$-r_1$
$\text{register}_{[2c_\text{sep}+c_\text{pad},3c_\text{sep})}$

FIG. 3. How to temporarily reduce oblivious carry runways to a single qubit, in preparation for being used as the input factor in a multiply-add operation that must iterate over all qubits in the register. The multiply-add should occur during the "..." section in the middle.

G. Abstract circuit model cost estimate

We have now described all the details necessary to estimate the cost of our implementation in the abstract circuit model. The cost depends on the two parameters specified by the problem (the input size n and the number of exponent qubits n_e) and also the four controllable parameters we have discussed (the exponentiation window size c_exp, the multiplication window size c_mul, the runway separation c_sep, and the padding/runway length c_pad).

Although we do consider alternate values when producing tables and figures, in general we have found that the settings $c_\text{exp} = c_\text{mul} = 5, c_\text{sep} = 1024, c_\text{pad} = 2\lg n + \lg n_e + 10 \approx 3\lg n + 10$ work well. In this overview we will focus on these simple, though slightly suboptimal, values.

Recall that an exponentiation may be reduced to a sequence of n_e multiplications, which we process in groups of size c_exp. For each multiplication, we do two multiply-adds. Each multiply-add will use several small additions to temporarily reduce the runway registers to single qubits. The multiply-add then needs to perform a sequence of additions controlled by the n main register qubits, the $O(\lg n)$ coset padding qubits, and the n/c_sep reduced runway qubits. Using windowed arithmetic, these additions are done in groups of size c_mul with each group handled by one lookup addition. Therefore the total number of lookup additions we need to perform is

$$\begin{aligned}\text{LookupAdditionCount}(n, n_e) &= \frac{2nn_e}{c_\text{exp}c_\text{mul}} \cdot \frac{c_\text{sep}+1}{c_\text{sep}} + O\left(\frac{n_e \lg n}{c_\text{exp}c_\text{mul}}\right) \\ &\to \frac{2nn_e}{25} \cdot \frac{1025}{1024} + O\left(n_e \lg n\right) \\ &\approx 0.1 n_e n\end{aligned} \tag{1}$$

The lookup additions make up essentially the entire cost of the algorithm, i.e. the total Toffoli count is approximately equal to the Toffoli count of a lookup addition times the lookup addition count. This works similarly for the measurement depth. Ignoring the negligible cost of uncomputing the lookup, the Toffoli count of a lookup addition is $2n + nc_\text{pad}/c_\text{sep} + 2^{c_\text{exp}+c_\text{pad}}$ and the measurement depth is $2n + 2c_\text{pad} + 2^{c_\text{exp}+c_\text{mul}}$. Therefore

$$\begin{aligned}\text{ToffoliCount}(n, n_e) &\approx \text{LookupAdditionCount}(n, n_e) \cdot \left(2n + c_\text{pad}\frac{n}{c_\text{sep}} + 2^{c_\text{exp}+c_\text{pad}}\right) \\ &\approx 0.2 n_e n^2 + 0.0003 n_e n^2 \lg n\end{aligned} \tag{2}$$

$$\begin{aligned}\text{MeasurementDepth}(n, n_e) &\approx \text{LookupAdditionCount}(n, n_e) \cdot \left(2c_\text{sep} + 2c_\text{pad} + 2^{c_\text{exp}+c_\text{mul}}\right) \\ &\approx 300 n_e n + 0.5 n_e n \lg n\end{aligned} \tag{3}$$

These approximate upper bounds, with n_e set to $1.5n$, are the formulae we report in the abstract and in Table I for the cost of factoring RSA integers.

H. Approximation error

Because we are using oblivious carry runways and the coset representation of modular integers, the computations we perform are not exact. Rather, they are approximations. This is not a problem in practice, as we may bound the approximation error using concepts and results from [26].

The oblivious runways and the coset representation of modular integers are both examples of "approximate encoded permutations" which have a "deviation". When using a padding/runway length of c_{pad}, and a runway separation of c_{sep}, the deviation per addition is at most $\frac{n}{c_{\text{sep}} 2^{c_{\text{pad}}}}$. Deviation is subadditive under composition, so the deviation of the entire modular exponentiation process is at most the number of lookup additions times the deviation per addition.

We can use this to check how much deviation results from using $c_{\text{pad}} = 2 \lg n + \lg n_e + 10$, which we so far have merely claimed would be sufficient:

$$\begin{aligned} \text{TotalDeviation}(n, n_e) &\leq \text{LookupAdditionCount}(n, n_e) \cdot \frac{n}{c_{\text{sep}} 2^{c_{\text{pad}}}} \\ &\approx 0.1 n_e n \frac{n}{1024 \cdot 2^{2 \lg n + \lg n_e + 10}} \\ &= 0.1 \frac{1}{1024 \cdot 2^{10}} \\ &\approx 10^{-7} \end{aligned} \tag{4}$$

When an approximate encoded permutation has a deviation of ϵ, the trace distance between its output and the ideal output is at most $2\sqrt{\epsilon}$. Therefore the approximation error (i.e. the trace distance due to approximations), using the parameter assignments we have described, is roughly 0.1%. This is significantly lower than other error rates we will calculate.

I. Spacetime layout

Our implementation of Shor's algorithm uses a layout that is derived from the (nearly) reaction limited layouts presented in [27]. In these layouts, a lookup addition is performed as follows (see Figure 4).

All data qubits are arranged into rows. The rows of qubits making up the target register of the addition are spread out into row pairs with gaps five rows wide in between. In these gaps there will be two rows for the lookup output register, and three rows for access hallways. This arrangement allows the lookup computation two ways to access each output row, doubling the speed at which it can run, while simultaneously ensuring the target register and the lookup output register are interleaved in a convenient fashion.

After the lookup is completed, the rows of the lookup output register and the addition target register are packed tightly against the top (or equivalently bottom) of the available area. An operating area is prepared below them, and the data is gradually streamed through the operating area while the operating area gradually shifts upward. This performs the MAJ sweep of Cuccaro's ripple carry adder [34]. Everything then begins moving in the opposite direction in order to perform the UMA sweep on the return stroke.

The lookup register is quickly uncomputed using measurement based uncomputation [37], and the system is returned to a state where another lookup addition can be performed.

In order to perform piecewise additions separated by oblivious carry runways, we simply partition the computation horizontally as shown in Figure 5 and Figure 7. The lookup before each addition prepares registers across the pieces, as shown in Figure 6, but the additions themselves stick to their own piece.

The width of each piece is determined by the number of CCZ factories needed to run at the reaction limited rate. This number is 14, assuming a code distance of 27 and using the CCZ factories from [11, 27]. Also, assuming a level 1 code distance of 17, the footprint of the factory is 15x8 [27]. The factories are laid out into 2 rows of 7, with single logical qubit gaps to allow space for data qubits to be routed in and out. The total width of a piece is $15 \cdot 7 + 7 = 113$ logical qubits.

The height of the operating area is 33 rows of logical qubits ($2 \cdot 8$ for the CCZ factories, 3 for the ripple-carry operating area, 6 for AutoCCZ fixup boxes, and 8 for routing data qubits). The $c_{\text{pad}} + c_{\text{sep}}$ qubits from each register add another 30 rows (approximately). So, overall, when working on a problem of size n, the computation covers a $113w \times 63$ grid of logical qubits where $w = n/c_{\text{sep}} = n/1024$.

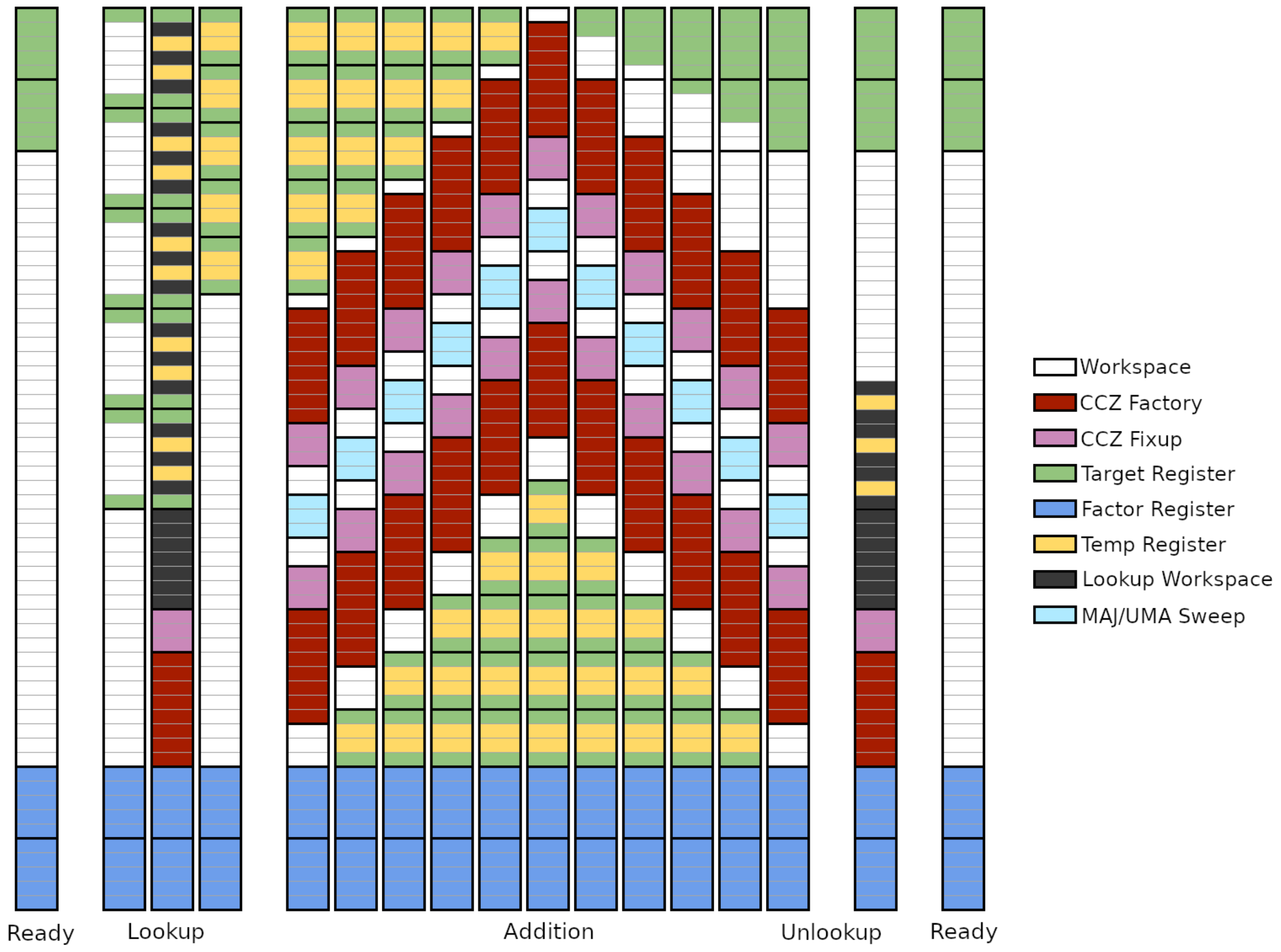


FIG. 4. How the computation is roughly laid out over time during the lookup-add-unlookup inner loop of our construction, as generalized from [27]. During lookup, the target register is spread out to make room for the temporary register that will hold the lookup's output. During addition the target register and lookup output register are squeezed through a moving operating area that sweeps up then down, applying the MAJ and UMA sweeps of Cuccaro's adder. Uncomputing the lookup is done with measurement based uncomputation [37], which is overlapped slightly with the UMA sweep (this is why the yellow rows disappear early).

J. Runtime

Because our implementation is dominated almost entirely by the cost of performing lookup additions, its duration is approximately equal to the number of lookup additions times the duration of a single lookup addition.

During the lookup phase of a lookup addition, the computation is code depth limited. Assuming a code depth of $d = 27$ and a surface code cycle time of 1 microsecond, it takes $1\mu s \cdot d/2 \cdot 2^{c_{\text{exp}}+c_{\text{mul}}} \approx 14$ milliseconds to perform the lookup using double-access hallways. During the addition phase of a lookup addition, the computation is reaction limited. Given a reaction time of 10 microseconds, it takes $2(c_{\text{sep}} + c_{\text{pad}}) \cdot 10\mu s \approx 22$ milliseconds to perform the addition. The remaining bits of a lookup addition, such as uncomputing the lookup and rearranging the rows, take approximately 1 millisecond.

Thus one lookup addition takes approximately 37 milliseconds. Given this fact, i.e. that we perform quantum lookup additions slower than most video games render entire frames, we can approximate the total duration:

$$\begin{aligned} \text{TotalRuntime}(n, n_e) &\approx \text{LookupAdditionCount}(n, n_e) \cdot 37\mu s \\ &\approx 4 n_e n \mu s \end{aligned} \tag{5}$$

Though we caution the reader that this estimate ignores the fact that, at larger problem sizes, lookups become

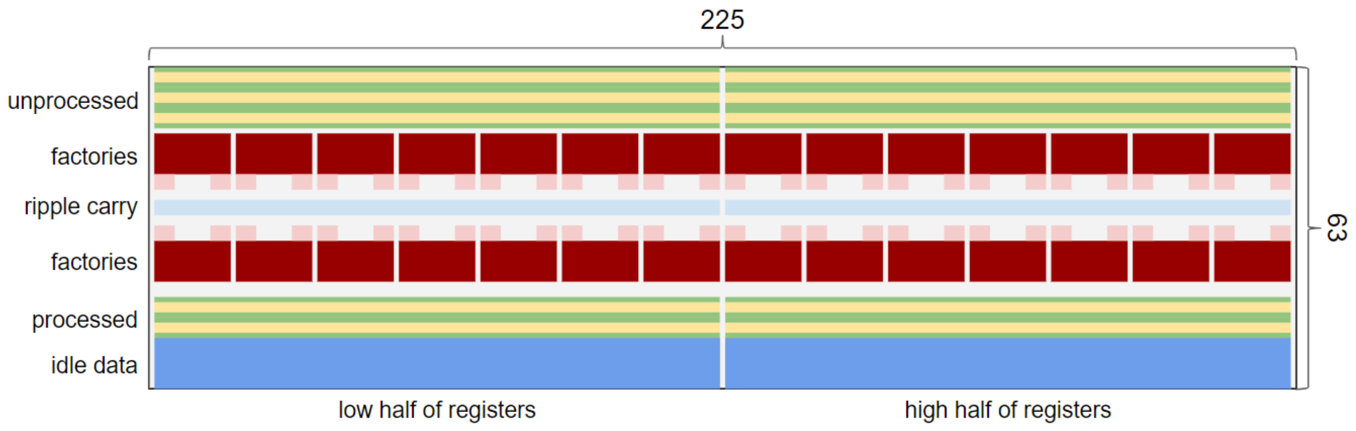


FIG. 5. Data layout during a parallel addition, as generalized from [27]. To scale, assuming a 2048 bit number is being factored. The left and right halfs of the computer run completely independently and in parallel. The factories (red boxes and pink boxes) are feeding AutoCCZ states into the blue area, which is rippling the carry bit back and forth as the offset and target register data (green and yellow rows) is routed through gaps between the factories to the other side.

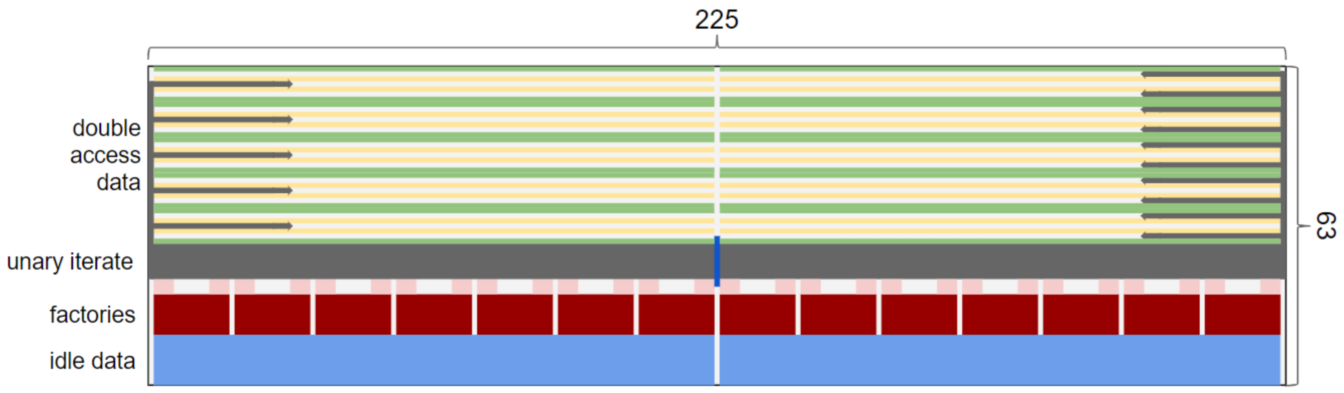


FIG. 6. Data layout during a table lookup, as generalized from [27]. To scale, assuming a 2048 bit number is being factored. The factories (red boxes and pink boxes) are feeding AutoCCZ states into the dark gray region, which is performing the unary iteration part of a table lookup computation. There are enough factories, and enough work space, to run the lookup at double speed, made possible by the fact that every qubit in the lookup output register (yellow) is adjacent to two access hallways. The target register (green rows) and factor register (blue rows) are idle, except that a few qubits from the factor register are being used as address bits in the table lookup.

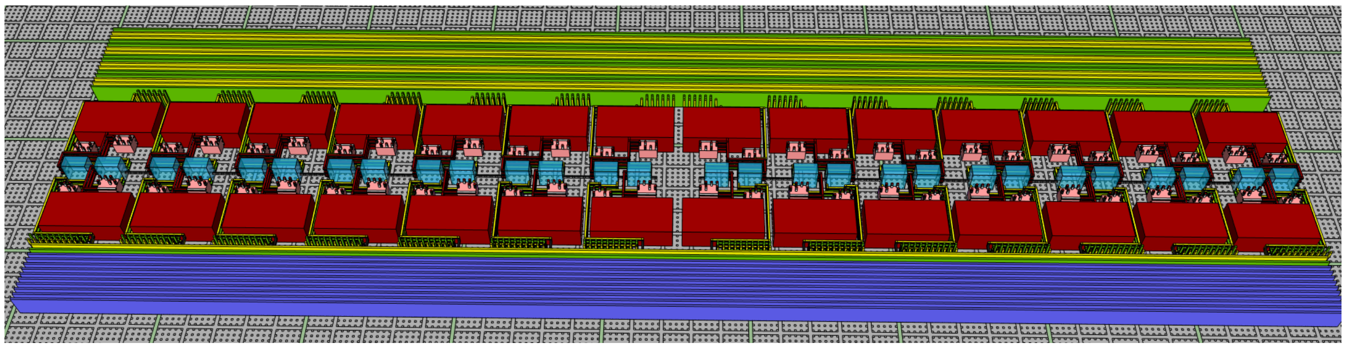


FIG. 7. Surface code activity during an addition, as generalized from [27]. Yellow and green rows are qubits from the target and input registers of the addition. Dark blue rows are qubits from an idle register. Red boxes are CCZ magic state factories, turned into AutoCCZ states by the pink boxes. Light blue boxes are the MAJ operation of Cuccaro's adder, arranged into a spacelike sweep to keep ahead of the control software). The full adder is formed by repeating this pattern, with the operating area gradually sweeping up and then down through the green/yellow data. The diagram is to scale for $n = 2048$ with a level 2 code distance of $d = 27$ and a level 1 code distance of 17, and fits on a 225×63 rectangular grid of logical qubits.

slower due to the minimum code distance increasing.

This estimate implies that factoring a 2048 bit integer will take approximately 7 hours, assuming only one run of the quantum part of the algorithm is needed. Note that in our reported numerical estimates we achieve lower per-run numbers by using more precise intermediate values and by more carefully selecting parameters.

K. Distillation error

In order to perform the $0.2 n_e n^2 + 0.0003 n_e n^2 \lg n$ Toffoli gates our implementation uses to factor an $n = 2048$ bit RSA integer, we need to distill approximately 3 billion CCZ states. According to the spreadsheet included in [11], using a level 1 code distance 17 and a level 2 code distance of 27, this corresponds to a total distillation error of 6.4%.

This quantity is computed by considering the initial error rate of injecting physical T states, topological error within the factory, and the likelihood of the various stages of distillation producing a false negative.

L. Topological error

Now that we know the number of logical qubits, and the duration of the algorithm, we can approximate the probability of a topological error occurring within the surface code during the algorithm. This will allow us to verify our initial assumption that a code distance of 27 is sufficient in the case where $n = 2048$. Larger computations will require larger code distances.

Assuming a physical gate error rate of 10^{-3}, the probability of error in a logical qubit of distance d, per surface code cycle time, is approximately $10^{-\lceil d/2+1 \rceil}$ (see equation 11 from [39]). When factoring an $n = 2048$ bit RSA integer we are using a board of $226 \cdot 63$ logical qubits. Approximately 25% of these qubits are being used for distillation, which we already accounted for, and so we do not count them in this calculation. The remaining qubits are kept through $4 n_e n \approx 50$ billion surface code cycles, implying the chance of topological error is approximately $10^{-\lceil 27/2+1 \rceil} \cdot 226 \cdot 63 \cdot 0.75 \cdot 50 \cdot 10^9 \approx 50\%$.

This is a large error rate. Using a code distance of 27 is pushing the limits of feasibility. We would need to repeat the computation twice, on average, doubling the expected runtime. If our goal is to minimize the expected spacetime volume of the computation, perhaps we should increase the code distance to 29. Doing so would increase the physical qubit count by 15%, but the error rate would drop by approximately a factor of 10 and so the expected number of retries would be much closer to 1. Ultimately the choice comes down to one's preferences for using more space versus taking more time.

Note that, in the more precise estimates computed by ancillary file "estimate_costs.py", the total error rate at distance 27 is estimated to be closer to 30%. This lowers the gains from increasing the code distance.

M. Physical qubit count

In lattice surgery, a logical qubit covers $2(d+1)^2$ physical qubits where d is the code distance (see Figure 8). Assuming we push the limits and use a code distance of 27 at $n = 2048$, each logical qubit will cover 1568 physical qubits. Therefore the total physical qubit count is the number of logical qubits $226 \cdot 63$ times 1568; approximately 23 million qubits.

Attentive readers will note that this number disagrees with the estimate in the title of the paper. This is because, throughout this section, we have been sloppily rounding quantities up and choosing fixed parameters in order to keep things simple. The estimate in the title is produced by the ancillary file "estimate_costs.py", which does not make these simplifications. (In particular, "estimate_costs.py" realizes that the level 1 code distance used during distillation can be reduced from 17 to 15 when $n = 2048$ and this adjusts the layout in several fortuitous ways.)

N. Summary

We now review the basic flow of our implementation of Shor's algorithm, including some details we would have otherwise left unstated.

The quantum part of the algorithm starts by preparing two registers, storing zero and one respectively, into the coset representation of modular integers with oblivious carry runways. The cost of this preparation is negligible compared to the cost of the rest of the algorithm. The register storing one is the accumulation register that will ultimately store the modular exponentiation, while the zero register is a workspace register.

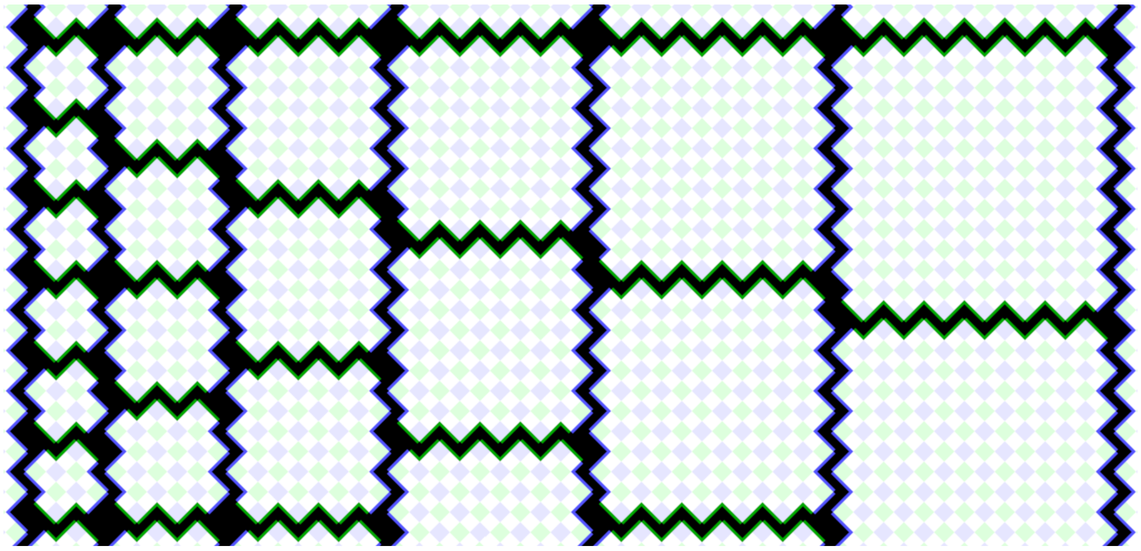


FIG. 8. Lattice surgery qubits, in the rotated surface code, with code distances d of (from left to right) 3, 5, 7, 9, 11, and 13. Each green (blue) square correspond to physical X (Z) measurement qubit. Each white square corresponds to a physical data qubit. Blacked out regions correspond to disabled qubits. The two types of boundaries are highlighted with blue and green borders. There are $2(d+1)^2$ physical qubits per logical qubit (including the dividers). With a physical error rate of 10^{-3}, each increase of the code distance by 2 suppresses logical errors per surface code cycle by a factor of 10.

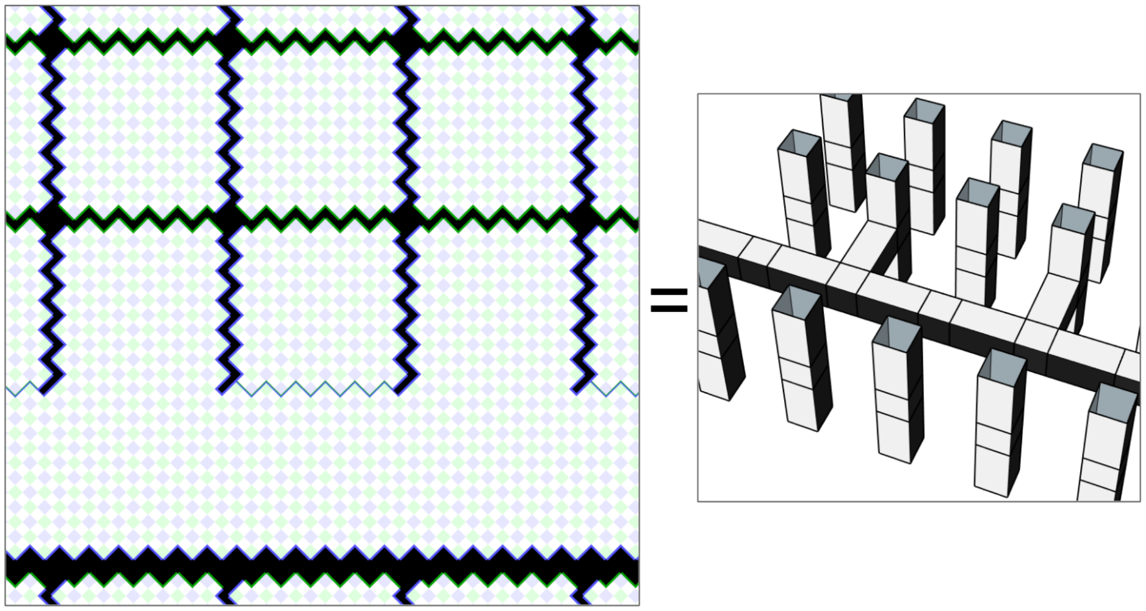


FIG. 9. Physical view and logical view of lattice surgery qubits during a computation. The vertical columns (right) are logical qubits, corresponding to patches in the physical layout (left). The horizontal bar/region corresponds to a stabilizer measurement of the product of the two connected qubits (times whatever else is connected) [12]. The connections shown in the 3d diagram correspond to missing boundaries in the physical layout, and indicate that physical stabilizer measurements are crossing between the two regions instead of staying separated.

The bulk of the execution time is then spent performing the modular exponentiation(s). Exponent qubits are iteratively introduced in groups of size c_{exp}. We are performing a semi-classical Fourier transform [31], so the exponent qubits must be phased according to measurements on previous exponent qubits. Performing the phasing operations requires using T factories instead of CCZ factories, but there is so little phasing work compared to CCZ work that we ignore this cost in our estimates. The phased exponent qubits are used as address qubits during a windowed modular multiplication, then measured in the frequency basis and discarded. See Figure 2. Within this step, almost all of the computation time is spent on lookup additions.

After all of the modular multiplications have been completed, the accumulation register is storing the result of the modular exponentiation. Note that this register still has oblivious carry runways and is still using the coset representation of modular integers. We could decode the register before measuring it but, because the decoding operations are all classical, it is more efficient to simply measure the entire register and perform the decoding classically. As with the cost of initialization of the registers, this cost is completely negligible.

This completes the quantum part of the algorithm. The exponent qubit measurement results, and the decoded accumulator measurement result, are fed into the classical post-processing code which uses them to derive the solution. If this fails, which can occur e.g. due to a distillation error during quantum execution, the algorithm is restarted.

Note that there are many minor implementation details that we have not discussed. For example, during a windowed multiplication, one of the registers is essentially sitting idle except that its qubits are being used (c_{mul} at a time) as address qubits for the lookup additions. We have not discussed how these qubits are routed into and out of the lookup computation as they are needed. It is simply clear that there is enough leeway, in the packing of the computation into spacetime, for this routing task to be feasible and contribute negligible cost.

3. CRYPTOGRAPHIC IMPLICATIONS OF OUR CONSTRUCTION

In this section, we consider how the security of RSA, and of cryptographic schemes based on the intractability of the DLP in finite fields, is affected when the optimized construction described in the previously section is used to implement the currently most efficient derivatives of Shor's algorithms.

Our goal throughout this section is to minimize the overall expected spacetime volume, including expected repetitions, when factoring RSA integers or computing discrete logarithms. That is to say, we minimize the spacetime volume in each run of the quantum algorithm times the expected number of runs required.

A. Methodology

To minimize the spacetime volume for a given choice of n and n_e, we consider all combinations of level 1 and 2 surface code distances $d_1 \in \{15, 17, \ldots, 23\}$ and $d_2 \in \{25, 27, \ldots, 51\}$ used during distillation and computation, window sizes $c_{\mathrm{mul}} \in \{4, 5, 6\}$ and $c_{\mathrm{exp}} \in \{4, 5, 6\}$, runway spacings $c_{\mathrm{sep}} \in \{512, 768, 1024, 1536, 2048\}$, and padding offsets $\delta_{\mathrm{off}} \in \{2, 3, \ldots, 10\}$ where $\delta_{\mathrm{off}} = c_{\mathrm{pad}} - 2 \lg n - \lg n_e$. Furthermore, we consider two different magic state distillation strategies: the CCZ factory from [11, 27] and the T factory from [12]. For each combination of parameters we estimate the execution time t and physical qubit count s, and upper bound the overall probability of errors occurring (to obtain the "retry risk" ϵ).

To derive an upper bound on the overall probability of errors occurring, we separately estimate the probabilities of topological errors occurring due to a failure of the surface code to correct errors, of approximation errors occurring due to using oblivious carry runways and the coset representation of modular integers, of magic state distillation errors, and of the classical post-processing algorithm failing to recover the solution from a correct run of the quantum algorithm. We combine these error probabilities, assuming independence, to derive an upper bound on the overall probability of errors occurring.

We chose to optimize the quantity $s^{1.2} \cdot t/(1-\epsilon)$, which we refer to as the "skewed expected spacetime volume". The $t/(1-\epsilon)$ factor is the expected runtime, and the $s^{1.2}$ factor grows slightly faster than the space usage. We skew the space usage when optimizing because we have a slight preference for decreasing space usage over decreasing runtime. We consider all combination of parameters (d_1, d_2, δ_{off}, c_{mul}, c_{exp}, c_{sep}), choose the set that minimizes the skewed expected spacetime volume, and report the corresponding estimated costs.

B. Implications for RSA

Today, the arguably most commonly used modulus size for RSA is $n = 2048$ bits. Larger moduli are however in widespread use and smaller moduli have been used historically. The best current academic record is the factorization of a 768 bit RSA modulus in 2009, see [40] for details. In Table III and Figure 1, we provide estimates for the resource and time requirements for attacking RSA for various cryptographically relevant modulus lengths n. The estimates are for factoring RSA integers with Ekerå-Håstads's algorithm [22, 23] that computes a short discrete logarithm, see the appendix to [23] for full technical details. As is explained in [23], a single correct run of this quantum algorithm suffices for the RSA integer to be factored with at least 99% success probability in the classical post-processing.

		Parameters						Retry	Volume (megaqubitdays)		Qubits (megaqubits)	Runtime (hours)
n	n_e	d_1	d_2	δ_{off}	c_{mul}	c_{exp}	c_{sep}	Risk	per run	expected	per run	per run
1024	$3(n/2-1)-40$	15	27	5	5	5	1024	6%	0.5	0.5	9.7	1.3
2048		15	27	4	5	5	1024	31%	4.1	5.9	20	5.1
3072		17	29	6	4	5	1024	9%	19	21	38	12
4096		17	31	9	4	5	1024	5%	48	51	55	22
8192		19	33	4	4	5	1024	5%	480	510	140	86
12288		19	33	3	4	5	1024	12%	1700	1900	200	200
16384		19	33	4	4	5	1024	24%	3900	5100	270	350

TABLE III. Factoring an n bit RSA integer by computing a short discrete logarithm. This table was produced by the script in the ancillary file "estimate_costs.py".

C. Implications for finite field discrete logarithms

Given a generator g of an order r subgroup to $\mathbb{Z}_N^*$, where the modulus N is prime, and an element $x = g^d$, the finite field discrete logarithm problem is to compute $d = \log_g x$. In what follows, we assume r to be prime. If r is composite, the discrete logarithm problem may be decomposed into problems in subgroups of orders dividing r, as shown by Pohlig and Hellman [41]. For this reason, prime order subgroups are used in cryptographic applications.

As $\mathbb{Z}_N^*$ has order $N-1$, it must be that r divides $N-1$, so $N = 2rk+1$ for some integer $k \geq 1$. The asymptotically best currently known classical algorithms for computing discrete logarithms in subgroups of this form are generic cycle-finding algorithms, such as Pollard's ρ- and λ-algorithms [42], that run in time $O(\sqrt{r})$ and $O(\sqrt{d})$, respectively, and the general number field sieve (GNFS), that runs in subexponential time in the bit length n of N.

The idea of factoring via algebraic number fields was originally introduced by Pollard [43] for integers on special forms. It was over time generalized, by amongst others Buhler et al. [44], Lenstra et al. [45] and Pollard [46], to factor general integers, and modified by amongst others Gordon [47] and Schirokauer [48, 49] to compute discrete logarithms in finite fields. Much research effort has since been devoted to optimizing the GNFS in various respects. For a proper in-depth historical account of the development of the GNFS, see [50, 51].

Let z be number of bits of security provided by the modulus with respect to classical attacks using the GNFS. Let n_d and n_r be the lengths in bits of d and r, respectively. It then suffices to pick $n_d, n_r \geq 2z$ to achieve z bits of classical security, as the generic cycle-finding algorithms are then not more efficient than the GNFS.

When instantiating schemes based on the intractability of the finite field discrete logarithm problem, one may hence choose between using a Schnorr group, for which $n_d = n_r = 2z$, or a safe-prime group, for which $n_r = n - 1$. In the latter case, one may in turn choose between using a short exponent, such that $n_d = 2z$, or a full length exponent, such that $n_d = n_r = n - 1$. All three parameterization options provide z bits of classical security.

1. *What finite field groups are used in practice?*

In practice, Schnorr groups or safe-prime groups with short exponents are often preferred over safe prime groups with full length exponents, as the comparatively short exponents yield considerable performance improvements.

The discrete logarithm problem in Schnorr groups is of standard form, unlike the short discrete logarithm problem in safe-prime groups, and Schnorr groups are faster to generate than safe-prime groups. A downside to using Schnorr groups is that group elements received from untrusted parties must be tested for membership of the order r subgroup. This typically involves exponentiating the element to the power of r, which is computationally expensive. Safe-prime groups are more flexible than Schnorr groups, in that the exponent length may be adaptively selected depending on the performance requirements. The reader is referred to [52] for a more in-depth comparison and historical recommendations. In more recent years, the use of safe-prime groups would appear to have become increasingly prevalent. Some cryptographic schemes, such as the Diffie-Hellman key agreement protocol, are agnostic to the choice of group, whereas other schemes, such as DSA, use Schnorr groups for efficiency reasons.

The National Institute of Standards and Technology (NIST) in the United States standardizes the use of cryptology in unclassified applications within the federal government. Up until April of 2018, NIST recommended the use of randomly selected Schnorr groups with moduli of length 2048 bits for Diffie-Hellman key agreement. NIST changed this recommendation in the 3rd revision of SP800-56A [53], and are now advocating using a fixed set of safe-prime groups with moduli of length up to 8192 bits, with short or full length exponents. These groups were originally developed for TLS [54] and IKE [55] where, again, they are used either with short of full length exponents.

2. *Complexity estimates*

To estimates the resource and time requirements for computing discrete logarithms in finite fields for various modulus lengths n, and for the aforementioned parameterizations options, we need to decide on what model to use for estimating z as a function of n. Various models have been proposed over the years, see for instance [56, 57]. For simplicity, we use the same model that NIST uses in SP 800-56A [53]. It is described on page 110 in FIPS 140-2 IG [58]. Note that NIST rounds z to the closest multiple of eight bits.

To compute short logarithms in safe-prime groups, the best option is to use Ekerå-Håstad's algorithm [22, 23] that is specialized for this purpose. To compute general discrete logarithms in safe-prime or Schnorr groups, one option is to use Ekerå's algorithm [24]. As is explained in [23, 24], a single correct run of these quantum algorithms suffices for the logarithm to be recovered with $\geq 99\%$ success probability in the classical post-processing. These algorithms do not require the order of the group to be known. See Table IV and Figure 1 for complexity estimates.

		n	n_e	n_d	n_r	z	Parameters d_1	d_2	δ_{off}	c_{mul}	c_{exp}	c_{sep}	Retry Risk	Volume (megaqubitdays) per run	Volume (megaqubitdays) expected	Qubits (megaqubits) per run	Runtime (hours) per run
Schnorr		1024	$3n_r = 6z$	$2z$	$2z$	80	15	25	4	5	5	1024	10%	0.2	0.2	9.2	0.4
		2048				112	15	27	3	5	5	1024	9%	0.9	1.0	20	1.2
		3072				128	15	27	9	5	5	1024	18%	2.4	2.9	29	2.0
		4096				152	17	29	7	4	5	1024	4%	6.5	6.8	51	3.1
		8192				200	17	31	3	4	5	1024	5%	38	40	110	8.3
		12288				240	17	31	9	4	5	1024	9%	110	120	170	15
		16384				272	17	31	5	4	5	1024	17%	210	250	220	23
Safe-prime	Short	1024	$3n_d = 6z$	$2z$	$n-1$	80	15	25	4	5	5	1024	10%	0.2	0.2	9.2	0.4
		2048				112	15	27	3	5	5	1024	9%	0.9	1.0	20	1.2
		3072				128	15	27	9	5	5	1024	18%	2.4	2.9	29	2.0
		4096				152	17	29	7	4	5	1024	4%	6.5	6.8	51	3.1
		8192				200	17	31	3	4	5	1024	5%	38	40	110	8.3
		12288				240	17	31	9	4	5	1024	9%	110	120	170	15
		16384				272	17	31	5	4	5	1024	17%	210	250	220	23
	Full Length	1024	$3n_r = 3(n-1)$	$n-1$	$n-1$	80	15	27	9	5	5	1024	10%	1.1	1.2	9.7	2.7
		2048				112	17	29	6	4	5	1024	6%	12	12	26	11
		3072				128	17	31	5	4	5	1024	5%	41	43	41	24
		4096				152	17	31	7	4	5	1024	9%	97	110	55	43
		8192				200	19	33	4	4	5	1024	8%	960	1100	140	180
		12288				240	19	33	3	4	5	1024	21%	3300	4100	200	390
		16384				272	21	35	4	4	5	1024	16%	9100	11000	320	700

TABLE IV. Computing discrete logarithms using Ekerå-Håstad's [22, 23] and Ekerå's [24] algorithms. This table was produced by the script in the ancillary file "estimate_costs.py".

If the group order is known, a better option for computing general discrete logarithms in safe-prime groups and Schnorr groups when not making tradeoffs is to use Shor's original algorithm [1], modified to work in the order r subgroup rather than in the whole multiplicative group $\mathbb{Z}_N^*$, and to start with a uniform superposition of all exponent values, as opposed to superpositions of r values. Note that the latter modification is necessary to enable the use of the semi-classical Fourier transform, qubit recycling and the windowing technique. The modified algorithm is described by Ekerå in [21] in the section on general logarithms, and in [59] where a heuristic analysis is also provided.

When using this modified version of Shor's algorithm to compute discrete logarithms, the heuristic analysis [59] shows that a single correct run suffices to compute the logarithm with $\geq 99\%$ success probability, assuming each of the two exponent registers is padded with 5 bits, and assuming a small search is performed in the classical post-processing. This implies that Shor's algorithm outperforms Ekerå's algorithm, as Shor's algorithm performs only approximately $2n_r$ group operations per run, compared to $3n_r$ operations in Ekerå's algorithm, see Table V for complexity estimates. This is because Ekerå's algorithm does not require r to be known. In fact, it computes both d and r.

Note that for safe-prime groups, $r = (N-1)/2$, so when N is known to the adversary then so is r. For Schnorr groups, it may be that r is unknown to the adversary, especially if the group is randomly selected. It may be hard to compute r classically, as it amounts to finding a $n_r = 2z$ bit prime factor of $(N-1)/2$.

	n	n_e	n_d	n_r	z	d_1	d_2	δ_{off}	c_{mul}	c_{exp}	c_{sep}	Retry Risk	Volume (megaqubitdays) per run	Volume (megaqubitdays) expected	Qubits (megaqubits) per run	Runtime (hours) per run
Schnorr	1024	$2(n_r+5)$	$2z$	$2z$	80	15	25	6	5	5	1024	8%	0.1	0.1	9.2	0.3
Schnorr	2048	$2(n_r+5)$	$2z$	$2z$	112	15	27	8	5	5	1024	6%	0.6	0.7	20	0.8
Schnorr	3072	$2(n_r+5)$	$2z$	$2z$	128	15	27	6	5	5	1024	13%	1.6	1.8	29	1.3
Schnorr	4096	$2(n_r+5)$	$2z$	$2z$	152	15	27	3	5	5	1024	25%	3.3	4.4	39	2.1
Schnorr	8192	$2(n_r+5)$	$2z$	$2z$	200	17	29	5	4	5	1024	11%	23	26	110	5.5
Schnorr	12288	$2(n_r+5)$	$2z$	$2z$	240	17	31	4	4	5	1024	8%	68	74	170	10
Schnorr	16384	$2(n_r+5)$	$2z$	$2z$	272	17	31	5	4	5	1024	12%	140	160	220	16
Safe-prime	1024	$2(n_r+5)$	$n-1$	$n-1$	80	15	27	3	5	5	1024	8%	0.7	0.8	9.7	1.8
Safe-prime	2048	$2(n_r+5)$	$n-1$	$n-1$	112	17	29	3	4	5	1024	5%	7.4	7.8	26	7.0
Safe-prime	3072	$2(n_r+5)$	$n-1$	$n-1$	128	17	29	4	4	5	1024	12%	25	29	38	16
Safe-prime	4096	$2(n_r+5)$	$n-1$	$n-1$	152	17	31	4	4	5	1024	8%	65	70	55	29
Safe-prime	8192	$2(n_r+5)$	$n-1$	$n-1$	200	19	33	3	4	5	1024	6%	640	680	140	120
Safe-prime	12288	$2(n_r+5)$	$n-1$	$n-1$	240	19	33	4	4	5	1024	15%	2200	2600	200	260
Safe-prime	16384	$2(n_r+5)$	$n-1$	$n-1$	272	21	35	2	4	5	1024	12%	6100	6900	320	470

TABLE V. Computing discrete logarithms using Shor's algorithm [1] modified as described in [21, 59]. This table was produced by the script in the ancillary file "estimate_costs.py".

D. Implications for elliptic curve discrete logarithms

Over the past decades cryptographic schemes based on the intractability of the DLP in finite fields and the RSA integer factoring problem have gradually been replaced by cryptography based on the intractability of the DLP in elliptic curve groups. This is reflected in standards issued by organization such as NIST.

Not all optimizations developed in this paper are directly applicable to arithmetic in elliptic curve groups. It is an interesting topic for future research to study to what extent the optimizations developed in this paper may be adapted to optimize such arithmetic operations (see Section 4 E). This paper should not be perceived to indicate that the RSA integer factoring problem and the DLP in finite fields is in itself less complex than the DLP in elliptic curve groups on quantum computers. The feasibility of optimizing the latter problem must first be properly studied.

E. On the importance of complexity estimates

It is important to estimate the complexity of attacking widely deployed asymmetric cryptographic schemes using future large-scale quantum computers. Such estimates enable informed decisions to be made on when to mandate migration from existing schemes to post-quantum secure schemes.

For cryptographic schemes that are used to protect confidentiality, such as encryption and key agreement schemes, a sufficiently long period must elapse inbetween the point in time when the scheme ceases to be used, and the point in time when the scheme is projected to become susceptible to practical attacks. This is necessary so as to ensure that the information that has been afforded protection with the scheme is no longer sensitive once the scheme becomes susceptible to practical attacks. This is because one must assume that encrypted information may be recorded and archived for decryption in the future. If the information you seek to protect is to remain confidential for 25 years, you must hence stop using asymmetric schemes such as RSA and Diffie-Hellman at least 25 years before quantum computers capable of breaking these schemes become available to the adversary. For cryptographic schemes that are used to protect authenticity, or for authentication, such as signature schemes, it suffices to migrate to post-quantum secure schemes before the schemes become susceptible to practical attacks. This is an important distinction.

F. On early adoption of post-quantum secure schemes

The process of transitioning to post-quantum secure schemes has already begun. However, no established or universally recognized standards are as of yet available. Early adopters may therefore wish to consider implementing schemes conjectured to be post-quantum secure alongside existing classically secure schemes, in such a fashion that both schemes must be broken for the combined hybrid scheme to be broken.

4. FUTURE WORK

A. Investigate asymptotically efficient multiplication

The multiplication circuits that we are using have a Toffoli count that scales quadratically (up to polylog factors). There are multiplication circuits with asymptotically better Toffoli counts. For example, the Karatsuba algorithm [60] has a Toffoli count of $O(n^{\lg 3})$ and the Schönhage–Strassen algorithm [61] has a Toffoli count of $O(n \lg n \lg \lg n)$. However, there are difficulties when attempting to use these asymptotically efficient algorithms in the context of Shor's algorithm.

The first difficulty is that efficient multiplication algorithms are typically classical, implemented with non-reversible computation in mind. They need to be translated into a reversible form. This is not trivial. For example, a naive translation of Karatsuba multiplication will result in twice as many recursive calls at each level (due to the need to uncompute), and increase the asymptotic Toffoli count from $O(n^{\lg 3})$ to $O(n^{\lg 6})$. Attempting to fix this problem can result in the space complexity increasing [62], though it is possible to solve this problem [63].

The second difficulty is constant factors, both in workspace and in Toffoli count. Clever multiplication circuits have better Toffoli counts for sufficiently large n but, when we do back-of-the-envelope estimates, "sufficiently large" is beyond $n = 2048$. This difficulty is made worse if the multiplier is incompatible with optimizations that work on naive multipliers, such as windowed arithmetic and the coset representation of modular integers. Clever multiplication circuits also tend to use additional workspace, and it is necessary to contrast using the better multiplier against the opportunity cost of using the space for other purposes (such as distillation of magic states). For example, the first step of the Schönhage–Strassen algorithm is to pad the input up to double length, then split the padded register into $O(\sqrt{n})$ pieces and pad each piece up to double length. The target register quadruples in size before even getting into the details of performing the number theoretic transform! This large increase in space means that the quantum variant of the Schönhage–Strassen algorithm is competing with the many alternative opportunities one has when given 6000 more logical qubits of workspace (at $n = 2048$). For example, that's enough space for fifty additional CCZ factories.

Can multipliers with asymptotically lower Toffolis counts help at practical sizes, such as $n = 2048$? We believe that they don't, but only a careful investigation can properly determine the answer to this question.

B. Optimize distillation

To produce our magic states, we use slightly-modified copies of the CCZ factory from [11] as explained in [27]. We have many ideas for improving on this approach.

First, the factory we are using is optimized for the case where it is working in isolation. But, generally speaking, error correcting codes get better when working over many objects instead of one object. It is likely that a factory using a block code to produce multiple CCZ states at once would perform better than the factory we used. For example, [64] presents a distillation protocol that produces good CCZ states ten at a time.

Second, since publishing [11], we have realized there are two obvious-in-hindsight techniques that could be used to reduce the volume of the factories in that paper. First, we assumed that topological errors that can occur within the factories were undetected, but actually many of them are heralded as distillation failures. By taking advantage of this heralding, it should be possible to reduce the code distance used in many parts of the factories. Second, when considering the set of S gates to apply to correct T gate teleportations performed by the factories, there are redundancies that we were not previously aware of. In particular, for each measured X stabilizer, one can toggle whether or not every qubit in that stabilizer has an S gate applied to it. This freedom makes it possible to e.g. apply dynamically chosen corrections while guaranteeing that the number of S gate fixups in the 15-to-1 T factory is at most 5 (instead of 15), or to ensure a particular qubit will never need an S gate fixup (removing some packing constraints).

Third, we believe it should be possible to use detection events produced by the surface code to estimate how likely it is that an error occurred, and that this information can be use to discard "risky factory runs" in a way that increases reliability. This would allow us to trade the increased reliability for a decreased code distance. As an extreme example, suppose that instead of attempting to correct errors during a factory run we simply discarded the run if there were any detection events where a local stabilizer flipped. Then the probability that a factory run would complete without being discarded would be approximately zero, but when a run did pass the chance of error would be amazingly low. We believe that by picking the right metric (e.g. number of detections or diameter of alternating tree during matching), then interpolating a rejection threshold between the extreme no-detections-anywhere rule and the implicit hope-all-errors-were-corrected rule that is used today, there will be a middle ground with lower expected volume per magic state. (Even more promisingly, this thresholded error estimation technique should work on any small state production task and almost all quantum computation can be reformulated as a series of small state production tasks.)

Reducing the volume of distillation would improve the space and time estimates in this paper. But turning some combination of the above ideas into a concrete factory layout, with understood error behavior, is too large of a task for one paper. Of course, the problem of finding small factories is a problem that the field has been exploring for some time. In fact, in the week before we first released this paper, Daniel Litinsky made [65] available. He independently arrived at the idea of using distillation to herald errors within the factory, and provided rough numerics indicating this may reduce the volume of distillation by as much as a factor of 10. Therefore we leave optimizing distillation not as future work, but as ongoing work.

C. Optimize qubit encoding

Most of the logical qubits in our construction spend most of their time sitting still, waiting for other qubits to be processed. It should be possible to store these qubits in a more compact, but less computationally convenient, form. A simple example is that resting qubits don't need the "padding layer" between qubits shown in Figure 9, because this layer is only there to make surgery easier. Another simple example is that there appear to be ways to pack lattice surgery qubits that use less area while preserving the code distance (e.g. the "houses" shown in Figure 10).

One could also imagine encoding the resting qubits into a block code. The difficulty is in finding a block code that a) works with small groups of qubits, b) can be encoded and decoded fast enough and compactly enough to fit into the existing computation, and c) is sufficiently better than the surface code that the benefits (reduced code distance within the surface code) outweigh the costs (redundant additional qubits).

Finally, there are completely different ways of storing information in the surface code. For example, qubits stored using dislocations [66] could be denser than qubits stored using lattice surgery. However, dislocations also create runs of stabilizers over not-quite-adjacent physical qubits. Measuring these stabilizers requires additional physical operations, creating more opportunities for errors, and so errors will propagate more quickly along these runs.

Do the "qubit houses" in Figure 10 actually work, or is there some unexpected error mechanism? Can dislocation qubits be packed more tightly than lattice surgery qubits while achieving an equivalent logical error rate? Is there a block code that is sufficiently beneficial when layered over surface code qubits? Until careful simulations are done it will be unclear what the answer to these questions is, and so we leave the answers to future work.

D. Distribute the computation

An interesting consequence of how we parallelize addition, by segmenting registers into pieces terminated by carry runways, is that there is limited interaction between the different pieces. In fact, the additions themselves require *no* communication between the pieces; it is only the lookup procedure preparing the input of the additions that requires communication. If we place each piece on a separate quantum computer, only a small amount of communication is needed to seed the lookups and keep the computation progressing.

Given the parameters we chose for our construction, each lookup is controlled by ten address qubits. Five of those qubits come from the exponent and are re-used hundreds of times. The other five come from the factor register, which is being iterated over during multiplication. If the computation was to be distributed over multiple machines, the communication cost would be dominated by broadcasting the successive groups of five qubits from the factor register.

Recall from Section 2 that each lookup addition takes approximately 37 milliseconds. Five qubits must be broadcast per lookup addition. Therefore, if each quantum computer had a 150 qb/s quantum channel, the necessary qubits could be communicated quickly enough to keep up with the lookup additions. This would distribute the factoring computation.

For example, a network topology where the computers were arranged into a linear chain and were constantly forwarding the next set of factor qubits to each other would be sufficient. Each quantum computer in the distributed computation would have a computational layout basically equivalent to the left half (or right half) of Figure 7. Factoring an $n = 2048$ bit number could be performed with two machines each having perhaps 11 million physical qubits, instead of one machine with 20 million physical qubits.

We can decrease the size of each individual quantum computer by decreasing the piece size we use for the additions. For example, suppose we used an addition piece size of $c_{\text{sep}} = 256$ and distributed an $n = 2048$ RSA factoring computation over 8 machines. Normally using smaller pieces would proportionally accelerate the addition, but the number of magic state factories has not changed (they have simply been distributed) so each lookup addition will still take approximately the same amount of time. So a 150 qb/s quantum channel per computer should still be sufficient. One caveat here is that each of these much smaller machines has to run its own copy of the lookup operation, and because there are so few factories per machine the lookup will take much longer. The optimal windows sizes of the lookups will decrease, and the total computation time will approximately double.

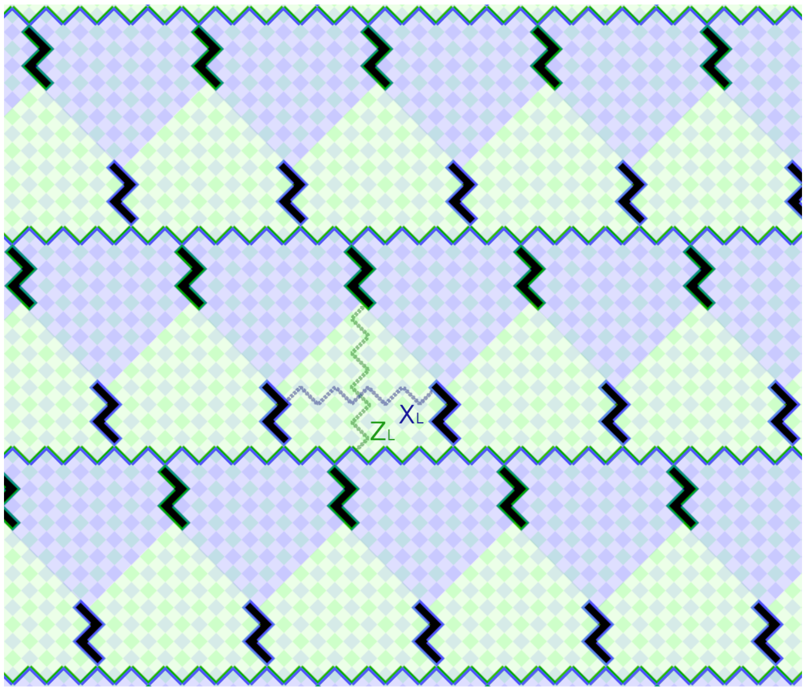


FIG. 10. A possible dense packing for idle lattice surgery qubits, using $1.5d^2 + O(d)$ physical qubits instead of $2(d+1)^2$. In the diagram, there is one distance 9 logical qubit per "house" (triangular roof plus rectangular base). The logical X/Z observable chains of one of the qubit houses is shown. The two region highlight colors correspond to different operation orders for performing stabilizer measurements. This is important information because every stabilizer measurement has one particular physical error, the "long error", that can produce detection events a distance 2 apart. By choosing the operation order, we determine if the long errors of each type are oriented vertically or horizontally. The goal is to avoid pointing long errors towards boundaries of the same type, because that reduces the effective code distance. In green regions, the long blue errors are vertical and the long green errors are horizontal (by "blue error" and "green error" we mean an error that terminates on a boundary of the same color). In blue regions, the long errors have the opposite orientation.

Based on this surface analysis it seems that, instead of using 1 machine with 20 million qubits, we could use 8 machines each with perhaps 4 million qubits, as long as they were connected by quantum channels with bandwidths of 150qb/s. But we have not carefully explored this question, and perhaps there are even better ways to distribute the computation with even lower overheads. Only future work can tell.

E. Revisit elliptic curves

Many of the optimization techniques that we use in this paper generalize to other contexts where arithmetic is performed. In particular, consider the Toffoli count for computing discrete logarithms over elliptic curves reported by [17]. It can likely be improved substantially by using windowed arithmetic. On the other hand, because of the need to compute modular inverses, it is not clear if the coset representation of modular integers is applicable.

Which of our optimizations can be ported over, and which ones cannot? How much of an improvement would result? These are interesting questions for future research work.

5. CONCLUSION

In this paper, we combined several techniques and optimizations into an efficient construction for factoring integers and computing discrete logarithms over finite fields. We estimated the approximate cost of our construction, both in the abstract circuit model and under plausible physical assumptions for large-scale quantum computers based on superconducting qubits. We presented concrete cost estimates for several cryptographically relevant problems. Our estimated costs are orders of magnitude lower than in previous works with comparable physical assumptions.

In [67], Mosca poses the hypophoric question: "How many physical qubits will we need to break RSA-2048? [...]

Current estimates range from tens of millions to a billion physical qubits". The upper bound of "a billion physical qubits" is likely from [9]. Our physical assumptions are more pessimistic than the physical assumptions used in that paper (see Table II), so our results can be directly compared. Doing so shows that, in the four years since 2015, the worst case estimate of how many qubits will be needed to factor 2048 bit RSA integers has dropped nearly two orders of magnitude; from a billion to twenty million.

Clearly the low end of Mosca's estimate should also drop. However, the low end of the estimate is highly sensitive to advances in the design of quantum error correcting codes, the engineering of physical qubits, and the construction of quantum circuits. As predicting these advances is out of the scope of this paper, we leave the task of estimating new lower bounds to others.

Post-quantum cryptosystems are in the process of being standardized [68], and small-scale experiments with deploying such systems on the internet have been performed [69]. However, a considerable amount of work remains to be done to enable large-scale deployment of post-quantum cryptosystems. We hope that this paper informs the rate at which this work needs to proceed.

6. CONTRIBUTIONS

Craig Gidney designed the efficient construction for modular exponentiation, produced initial cost estimates for RSA, and assembled results from other papers for comparison. Martin Ekerå did the cryptographic impact analysis, and extended the construction and cost estimates to problems beyond factoring RSA integers and to algorithms beyond Shor's factoring algorithm.

7. ACKNOWLEDGEMENTS

We thank Adam Langley, Ilya Mironov, Ananth Raghunathan, and Ryan Babbush for reading a draft of this paper and providing useful feedback which improved it. We thank Austin Fowler and Johan Håstad for useful feedback and discussions. Craig Gidney thanks Hartmut Neven for creating an environment where this research was possible in the first place.

[1] P. W. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)* (IEEE, 1994) pp. 124–134.
[2] Wikipedia, "Timeline of quantum computing," `https://en.wikipedia.org/wiki/Timeline_of_quantum_computing`, Accessed: 2018-12-18.
[3] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM **21**, 120–126 (1978).
[4] W. Diffie and M. E. Hellman, "New Directions in Cryptography," IEEE Transactions on Information Theory **IT-22**, 644–654 (1976).
[5] NIST, "FIPS 186-4: Digital Signature Standard (DSS)," (2013).
[6] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," SIGMETRICS Performance Evaluation Review **37**, 193–204 (2009).
[7] R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. C. White, J. Mutus, A. G. Fowler, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, C. Neill, P. O'Malley, P. Roushan, A. Vainsencher, J. Wenner, A. N. Korotkov, A. N. Cleland, and J. M. Martinis, "Superconducting quantum circuits at the surface code threshold for fault tolerance," Nature **508**, 500–503 (2014), arXiv:1402.4848.
[8] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (2014) pp. 361–372.
[9] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," Physical Review A **86**, 032324 (2012), arXiv:1208.0928.
[10] E. Campbell, A. Khurana, and A. Montanaro, "Applying quantum algorithms to constraint satisfaction problems," arXiv preprint arXiv:1810.05582 (2018).
[11] C. Gidney, "Efficient magic state factories with a catalyzed $|\mathrm{CCZ}\rangle$ to $2|\mathrm{T}\rangle$ transformation," Quantum **3** (2019), 10.22331/q-2019-04-30-135.
[12] A. G. Fowler and C. Gidney, "Low overhead quantum computation using lattice surgery," arXiv preprint arXiv:1808.06709 (2018).

[13] V. Vedral, A. Barenco, and A. Ekert, "Quantum networks for elementary arithmetic operations," Physical Review A **54**, 147–153 (1996), arXiv:1208.0928.
[14] C. Zalka, "Fast versions of Shor's quantum factoring algorithm," arXiv preprint quant-ph/9806084 (1998).
[15] S. Beauregard, "Circuit for Shor's algorithm using $2n + 3$ qubits," arXiv preprint quant-ph/0205095 (2002).
[16] T. Häner, M. Roetteler, and K. M. Svore, "Factoring using $2n + 2$ qubits with Toffoli based modular multiplication," arXiv preprint arXiv:1611.07995 (2016).
[17] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter, "Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms," in *Proceedings of the 23rd International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT 2017)*, Lecture Notes in Computer Science (LNCS), Vol. 10625 (Springer, 2017) pp. 241–270.
[18] J. O'Gorman and E. T. Campbell, "Quantum computation with realistic magic-state factories," Physical Review A **95**, 032338(1–19) (2017).
[19] V. Gheorghiu and M. Mosca, "Quantum cryptanalysis of symmetric, public-key and hash-based cryptographic schemes," arXiv preprint arXiv:1902.02332 (2019).
[20] C. Gidney, "Windowed quantum arithmetic," arXiv preprint arXiv:1905.07682 (2019).
[21] M. Ekerå, "Modifying Shor's algorithm to compute short discrete logarithms," Cryptology ePrint Archive, Report 2016/1128 (2016).
[22] M. Ekerå and J. Håstad, "Quantum Algorithms for Computing Short Discrete Logarithms and Factoring RSA Integers," in *Proceedings of the 8th International Workshop on Post-Quantum Cryptography (PQCrypto 2017)*, Lecture Notes in Computer Science (LNCS), Vol. 10346 (Springer, 2017) pp. 347–363.
[23] M. Ekerå, "On post-processing in the quantum algorithm for computing short discrete logarithms," Cryptology ePrint Archive, Report 2017/1122 (2019), (2nd revision).
[24] M. Ekerå, "Quantum algorithms for computing general discrete logarithms and orders with tradeoffs," Cryptology ePrint Archive, Report 2018/797 (2018).
[25] C. Zalka, "Shor's algorithm with fewer (pure) qubits," arXiv preprint quant-ph/0601097 (2006).
[26] C. Gidney, "Approximate encoded permutations and piecewise quantum adders," arXiv preprint arXiv:1905.08488 (2019).
[27] C. Gidney and A. G. Fowler, "Flexible layout of surface code computations using AutoCCZ states," arXiv preprint arXiv:1905.08916 (2019).
[28] The OpenSSH Project, "Linux documentation: Man page for `ssh-keygen(1)`," `https://linux.die.net/man/1/ssh-keygen` (2018), Accessed: 2018-12-11.
[29] The GnuPG Project, "GnuPG Frequently Asked Questions: Why does GnuPG default to 2048 bit RSA-2048?" `https://www.gnupg.org/faq/gnupg-faq.html#default_rsa2048` (2018), Accessed: 2018-12-11.
[30] OpenSSL Software Foundation, "Openssl source code: Line 32 of `apps/dhparam.c`," `https://github.com/openssl/openssl/blob/07f434441e7ea385f975e8df8caa03e62222ca61/apps/dhparam.c#L32` (2018), Accessed: 2018-12-11.
[31] R. B. Griffiths and C.-S. Niu, "Semiclassical Fourier transform for quantum computation," Physical Review Letters **76**, 3228–3231 (1996).
[32] M. Mosca and A. Ekert, "The Hidden Subgroup Problem and Eigenvalue Estimation on a Quantum Computer," in *Proceeding from the First NASA International Conference: Quantum Computing and Quantum Communications (QCQC 1998)*, Vol. 1509 (Springer, 1999) pp. 174–188.
[33] C. Gidney, "Factoring with $n + 2$ clean qubits and $n - 1$ dirty qubits," arXiv preprint arXiv:1706.07884 (2017).
[34] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, "A new quantum ripple-carry addition circuit," arXiv preprint quant-ph/0410184 (2004).
[35] C. Gidney, "Halving the cost of quantum addition," Quantum **2**, 74 (2018).
[36] R. Babbush, C. Gidney, D. W. Berry, N. Wiebe, J. McClean, A. Paler, A. Fowler, and H. Neven, "Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity," Physical Review X **8**, 041015(1–36) (2018).
[37] D. W. Berry, C. Gidney, M. Motta, J. R. McClean, and R. Babbush, "Qubitization of Arbitrary Basis Quantum Chemistry by Low Rank Factorization," arXiv preprint arXiv:1902.02134 (2019).
[38] T. G. Draper, S. A. Kutin, E. M. Rains, and K. M. Svore, "A Logarithmic-Depth Quantum Carry-Lookahead Adder," arXiv preprint quant-ph/0406142 (2004).
[39] A. G. Fowler and S. J. Devitt, "A bridge to lower overhead quantum computation," arXiv preprint arXiv:1209.0510 (2012).
[40] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, t. R. Herman, A. Timofeev, and P. Zimmermann, "Factorization of a 768-Bit RSA Modulus," in *Proceedings of the 30th Annual Cryptology Conference: Advances in Cryptology (CRYPTO 2010)*, Lecture Notes in Computer Science (LNCS), Vol. 6223 (Springer, 2010) pp. 333–350.
[41] S. C. Pohlig and M. E. Hellman, "An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance," IEEE Transactions on Information Theory **IT-24**, 106–110 (1978).
[42] J. M. Pollard, "Monte Carlo Methods for Index Computation (mod p)," Mathematics of Computation **32**, 918–924 (1978).
[43] J. M. Pollard, "Factoring with cubic integers," in *The Development of the Number Field Sieve*, Lecture Notes in Mathematics (LNM), Vol. 1554 (Springer, 1993) pp. 4–10, (Original Manuscript Date: 1988-08-31).
[44] J. P. Buhler, H. W. Lenstra Jr., and C. Pomerance, "Factoring integers with the number field sieve," in *The Development of the Number Field Sieve*, Lecture Notes in Mathematics (LNM), Vol. 1554 (Springer, 1993) pp. 50–94, (Original Manuscript Date: 1992-05-07).
[45] A. K. Lenstra, H. W. Lenstra Jr., M. S. Manasse, and J. M. Pollard, "The number field sieve," in *Proceedings of the 22nd Annual ACM Symposium on the Theory of Computing (STOC)* (ACM, 1990) pp. 564–572.

[46] J. M. Pollard, "The lattice sieve," in *The Development of the Number Field Sieve*, Lecture Notes in Mathematics (LNM), Vol. 1554 (Springer, 1993) pp. 43–49, (Original Manuscript Date: September 1991).

[47] D. M. Gordon, "Discrete logarithms in GF(p) using the Number Field Sieve," SIAM Journal on Discrete Mathematics **6**, 124–138 (1993).

[48] O. Schirokauer, *On pro-finite groups and on discrete logarithms*, Ph.D. thesis, University of California, Berkeley (1992).

[49] O. Schirokauer, "Discrete Logarithms and Local Units," Philosophical Transactions of the Royal Society of London A **345**, 409–423 (1993).

[50] A. K. Lenstra and H. W. Lenstra Jr., eds., *The Development of the Number Field Sieve*, Lecture Notes in Mathematics (LNM), Vol. 1554 (1993).

[51] C. Pomerance, "A Tale of Two Sieves," Notices of the AMS **43**, 1473–1485 (1996).

[52] P. C. van Oorschot and M. J. Wiener, "On Diffie-Hellman Key Agreement with Short Exponents," in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques, Advanced in Cryptology (EUROCRYPT 1996), LNCS*, Vol. 1070 (Springer, 1996) pp. 332–343.

[53] NIST, "SP 800-56A (3rd revision): Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography," (2018), Accessed: 2019-03-11.

[54] D. Gillmor, "RFC 7919: Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)," (2016).

[55] T. Kivinen and M. Kojo, "RFC 3526: More Modular Exponentiation (MODP) Diffie-Hellman groups for Internet Key Exchange," (2003).

[56] A. K. Lenstra and E. R. Verheul, "Selecting Cryptographic Key Sizes," Journal of Cryptology **14**, 225–293 (2001).

[57] A. K. Lenstra, "Key Lengths," in *The Handbook of Information Security* (2004) Chap. 6.

[58] NIST and CCCS, "Implementation Guidance for FIPS 140-2 and the Cryptographic Module Validation Program," (2019), Accessed: 2019-05-10, Document Revision: 2019-05-07.

[59] M. Ekerå, "Revisiting Shor's quantum algorithm for computing general discrete logarithms," arXiv preprint arXiv:1905.09084 (2019).

[60] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," Doklady Akademii Nauk SSSR **145**, 293–294 (1962).

[61] A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen," Computing **7**, 281–292 (1971).

[62] A. Parent, M. Roetteler, and M. Mosca, "Improved reversible and quantum circuits for Karatsuba-based integer multiplication," in *Proceedings of the 12th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 73 (Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018) pp. 7:1–7:15, arXiv:1706.03419.

[63] C. Gidney, "Asymptotically Efficient Quantum Karatsuba Multiplication," arXiv preprint arXiv:1904.07356 (2019).

[64] J. Haah and M. B. Hastings, "Codes and Protocols for Distilling T, controlled-S, and Toffoli Gates," Quantum **2**, 71 (2018).

[65] D. Litinski, "Magic state distillation: Not as costly as you think," arXiv preprint arXiv:1905.06903 (2019).

[66] M. B. Hastings and A. Geller, "Reduced Space-Time and Time Costs Using Dislocation Codes and Arbitrary Ancillas," arXiv preprint arXiv:1408.3379 (2014).

[67] M. Mosca, "Cybersecurity in an Era with Quantum Computers: Will We Be Ready?" IEEE Security & Privacy Magazine **16**, 38–41 (2018).

[68] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, *NIST IR 8240: Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*, Tech. Rep. (NIST, 2019).

[69] M. Braithwaite, "Experimenting with post-quantum cryptography," `https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html` (2016).

[70] BlueKrypt, "Cryptographic key length recommendation," `https://www.keylength.com`, Accessed: 2019-03-03.

Appendix A: Notes on Table I

1. Columns

- **Abstract Qubits**: The number of logical qubits used in the abstract circuit model. Ignores qubits used for distillation and routing.

- **Measurement Depth**: The length of the longest chain of dependent measurements, which determines the reaction limited runtime of an algorithm. These numbers are not adjusted to account for the chance of retrying.

- **Toffoli+T/2**: Number of magic states required by the algorithm. The "/2" adjustment is intended to account for the fact that T states take less volume to distill than Toffoli states. These numbers are not adjusted to account for the chance of retrying.

- **Min volume**: Expected spacetime cost of the computation, including retries. The papers included in the table span decades, and a range of assumptions about the architecture of quantum computers, and generally do not provide spacetime volumes. To assign volumes to these papers, we plugged their asymptotic formulas into various possible realizations of that algorithm (serial, parallel, and intermediate) using ancillary file "fill-in-table.py". Each volume entry is the minimum volume achieved by the different possible realizations. We assumed all papers but our own had a negligible retry chance.

2. Entries

Some entries in the table are directly from a paper, others had to be inferred by hand, and others were filled in using the output of the ancillary file "fill-in-table.py". Here is where each entry in the table came from:

- Vedral et al. 1996 [13]:
  - Toffoli+T/2 Count: $80n^3$ derived from figures at end of the paper.
  - Abstract Qubits: Paper says $7n + 1$ (end of section IV). Verified using figures at end of paper. Paper mentions this can be improved to $4n + 3$, but the described method would radically increase the Toffoli count.
  - Measurement Depth: Equal to the Toffoli count. The circuit construction is serial.
  - Numbers at specific n: from "fill-in-table.py".
- Zalka 1998 (basic) [14]:
  - Toffoli+T/2 Count: Paper says $12n^3$ (title of section 1.4.1).
  - Abstract Qubits: Paper says $3n$ (title of section 1.4.1).
  - Measurement Depth: Equal to the Toffoli count. The circuit construction is serial.
  - Numbers at specific n: from "fill-in-table.py".
- Zalka 1998 (log add) [14]:
  - Toffoli+T/2 Count: Paper says $52n^3$ (section 5, bottom of page 18).
  - Abstract Qubits: Paper says $5n$ (section 5, bottom of page 18).
  - Measurement Depth: Paper says $600n^2$ (section 5, top of page 19).
  - Numbers at specific n: from "fill-in-table.py".
- Zalka 1998 (fft mult) [14]:
  - Toffoli+T/2 Count: Paper says $2^{17}n^2$ (section 5, middle of page 19).
  - Abstract Qubits: Paper says $96n$ (section 5, middle of page 19).
  - Measurement Depth: Paper says $2^{17}n^{1.2}$ (section 5, middle of page 19).
  - Numbers at specific n: from "fill-in-table.py".
- Beauregard 2002 [15]:
  - Toffoli+T/2 Count: Derived from figures in the paper. Our estimate has an additional factor of $\lg n$ to account for the need to approximate arbitrary phase rotations using T states.
  - Abstract Qubits: Stated in the title of the paper.
  - Measurement Depth: Derived from figures in the paper. Our estimate has an additional factor of $\lg n$ to account for the need to approximate arbitrary phase rotations using T states.
  - Numbers at specific n: from "fill-in-table.py".
- Fowler et al. 2012 [9]:
  - Toffoli+T/2 Count: From table I in the paper.
  - Abstract Qubits: The paper erroneously claims $2n$ in table I due to overlooking a necessary workspace register. We corrected this to $3n + O(1)$.

  - Measurement Depth: From table I in the paper.
  - Numbers at specific n: from “fill-in-table.py”.

- Häner et al. 2016 [16]:
  - Toffoli+T/2 Count: From table 2 of [17].
  - Abstract Qubits: From paper’s title.
  - Measurement Depth: Derived $52n^3$ from the circuit diagrams included in the paper.
  - Numbers at specific n: from “fill-in-table.py”.
- (ours) 2019: Asymptotic bounds explained in Section 2. Concrete numbers at specific sizes come from ancillary file “estimate_costs.py”. The Toffoli count does not account for the chance of retrying (because it is mostly insensitive to changes that lower this chance), but the volume does account for it (i.e. it is the expected total volume to factor, not the per-run volume).
- Roetteler et al 2017 [17]: Note that we used different values of n in order to compare against ECC keys with comparable security. We used the same size pairs as in [17] after double-checking them using keylength.com [70].
  - Toffoli+T/2 Count (asymptotic): From the paper’s abstract.
  - Abstract Qubits: From the paper’s abstract.
  - Measurement Depth: Assumed same as Toffoli count.
  - Toffoli+T/2 Count (at $n = 224$): From Table 2.
  - Numbers at specific n: from “fill-in-table.py”.

Appendix B: Notes on Table II

1. Columns

- **Physical gate error rate**: The probability that executing a physical gate will introduce Pauli errors onto targeted qubits.
- **Cycle time**: The amount of time it takes to measure all of the surface code’s stabilizers once.
- **Reaction time**: The amount of time it takes the classical control system to trigger a logical measurement, collect and error-correct the result, and decide on which measurement basis to use for the next set of measurements.
- **Physical connectivity**: Which physical qubits can interact with each other. Planar means only adjacent qubits on a planar grid can interact (typical of superconducting qubits). Arbitrary means qubits can interact with other qubits as needed by the construction (typical of ion traps).
- **Distillation strategy**: The dominant kind of magic states being distilled, and the number or type of factory being used to distill them.
- **Execution strategy**: How the computation progresses. Serial distillation means there is one factory producing magic states one by one, and this determines the cadence of the computation. Threaded means reaction limited computation is being used, with the number of threads corresponding to the number of distinct chains of dependent Toffolis being executed in parallel.
- **Physical qubits**: The number of physical qubits used by the computation, using the original historical assumptions.
- **Expected runtime**: The average amount of time before the computation has completed successfully, using the original historical assumptions.
- **Expected volume**: The average number of physical qubit-rounds before the computation has completed successfully, using the original historical assumptions.

2. Entries

- Fowler et al. 2012 [9]: Numbers are from the background section of [9]. Their expected volume is directly comparable to ours, despite the different reaction time assumption, because their volume estimate is dominated by distillation and changing the reaction time does not affect distillation volume.

- O'Gorman et al. (2017) [18]: Numbers are derived from table I of [18]. The physical qubit count of $2.18 \cdot 10^8$ from their table only includes distillation; we increased it by $3 \cdot 2048 \cdot (2 \cdot 32^2) \approx 0.13 \cdot 10^8$ to account for three n-logical-qubit data registers. It is difficult to directly compare O'Gorman et al's numbers to ours, because their estimates were made with ion traps in mind. Their underlying physical assumptions differ in ways that have non-trivial effects on the physical layout. In particular, their factories interact remote logical qubits without routing them towards each other.

- Gheorghiu et al. (2019) [19] The physical gate error rate, qubit count, and time are stated in the caption of fig 45 ("B. RSA-2048") of the paper. The surface code cycle time is stated to be 200ns in Section B, soon after equation 3. The T factory count was inferred by dividing the stated T count of $2.4 \cdot 10^{12}$ by the production rate of the T factory from [12], which is approximately 25kHz for the given surface code cycle time. The paper does not state a reaction time, but the T states are being used to perform chains of dependent Toffolis. This implies the reaction time must be faster than the number factories times the per-factory production rate divided by 4 (the number of Ts needed to perform a Toffoli). So a 0.2 microsecond reaction time would not be sufficient, but a 0.1 microsecond reaction time would be, and so we state the reaction time as 0.1 microseconds.

- Ours (2019): The parallel entry is the implementation described in Section 2 and produced by ancillary file "estimates_costs.py". The single threaded and serial distillation entries use the same basic architecture, but with fewer factories and without piecewise additions or double-speed lookups.