How to factor 2048 bit RSA integers with less than a million noisy qubits

Craig Gidney

Google Quantum AI, Santa Barbara, California 93117, USA

May 23, 2025

Planning the transition to quantum-safe cryptosystems requires understanding the cost of quantum attacks on vulnerable cryptosystems. In Gidney+Ekerå 2019, I co-published an estimate stating that 2048 bit RSA integers could be factored in eight hours by a quantum computer with 20 million noisy qubits. In this paper, I substantially reduce the number of qubits required. I estimate that a 2048 bit RSA integer could be factored in less than a week by a quantum computer with less than a million noisy qubits. I make the same assumptions as in 2019: a square grid of qubits with nearest neighbor connections, a uniform gate error rate of 0.1%, a surface code cycle time of 1 microsecond, and a control system reaction time of 10 microseconds.

The qubit count reduction comes mainly from using approximate residue arithmetic (Chevignard+Fouque+Schrottenloher 2024), from storing idle logical qubits with yoked surface codes (Gidney+Newman+Brooks+Jones 2023), and from allocating less space to magic state distillation by using magic state cultivation (Gidney+Shutty+Jones 2024). The longer runtime is mainly due to performing more Toffoli gates and using fewer magic state factories compared to Gidney+Ekerå 2019. That said, I reduce the Toffoli count by over 100x compared to Chevignard+Fouque+Schrottenloher 2024.

Data availability: *Code and assets created for this paper are available on Zenodo [Gid25].*

1 Introduction **3**

2 Methods **3**
2.1 Approximate Residue Arithmetic . . . 3
2.2 Approximate Period Finding . . . 7
2.3 Ekerå-Håstad Period Finding . . . 13
2.4 Arithmetic Optimizations . . . 14

3 Results **16**
3.1 Logical Costs . . . 16
3.2 Physical Costs . . . 18

4 Conclusion **21**

5 Acknowledgments **21**

A Detailed Mock-ups **26**
A.1 Reference Python Implementation of Algorithm . . . 26
A.2 Addition Operation . . . 30
A.3 Phaseup Operation . . . 34
A.4 Lookup Operation . . . 35
A.5 Frequency Basis Measurement . . . 38

Craig Gidney: craig.gidney@gmail.com

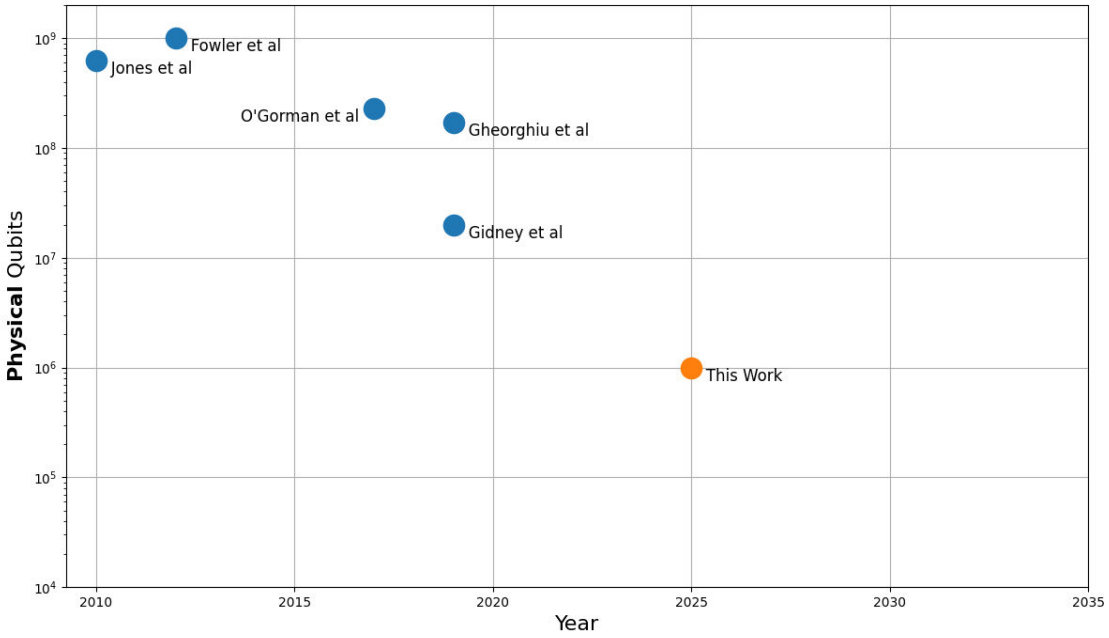

Figure 1: Historical estimates, with comparable physical assumptions, of the physical qubit cost of factoring 2048 bit RSA integers. Includes overheads from fault tolerance, routing, and distillation. Results are from [Jon+12; Fow+12; OC17; GM19; GE21]. Results such as [Van+10] and [LN22] aren't included because they target substantially different assumptions or cost models.

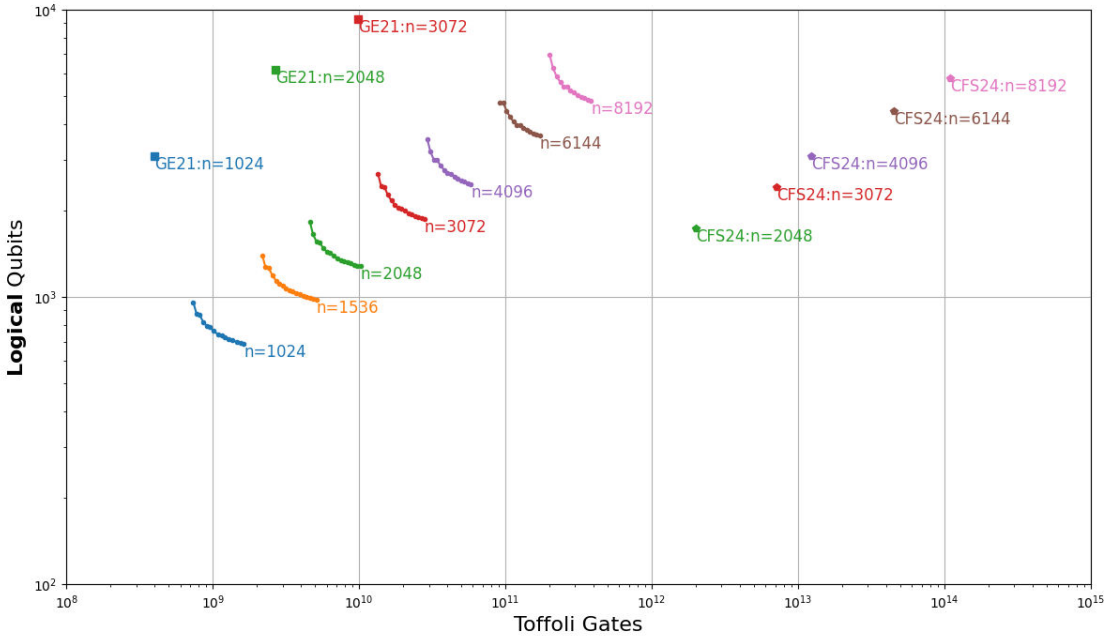

Figure 2: Pareto frontiers achieved by this paper for the Toffoli and logical qubit cost of factoring n bit RSA integers, for various values of n. This paper uses notably fewer logical qubits than [GE21] (points labeled "GE21") and notably fewer Toffolis than [CFS24] (points labeled "CFS24").

1 Introduction

In 1994, Peter Shor published a paper showing quantum computers could efficiently factor integers [Sho94], meaning the RSA cryptosystem [RSA78] wasn't secure against quantum computers. Understanding the cost of quantum factoring is important for planning and coordinating the transition away from RSA, and other cryptosystems vulnerable to quantum computers. Correspondingly, since Shor's paper, substantial effort has gone into understanding the cost of quantum factoring [Kni95; Bec+96; VBE96; Zal98; CW; Bea03; Zal06; Whi+09; Van+10; Fow+12; Jon+12; PG14; Eke16; HRS16; Gid17; OC17; GM19; Eke19; Eke21; GE21; LN22; MS22; Reg24; CFS24] (and more). A key metric is the number of qubits used by the algorithm, since this bounds the required size of quantum computer.

Historically, there was no known way to factor an n bit number using fewer than $1.5n$ logical qubits [Zal06; Bea03; HRS16; Gid17]. Anecdotally, it was widely assumed that n was the minimum possible because doing arithmetic modulo an n bit number "required" an n qubit register. May and Schlieper had shown in 2019 that in principle only a single *output* qubit was needed [MS22], but there was no known way to prepare a relevant output that didn't involve intermediate values spanning n qubits. In 2024, Chevignard and Fouque and Schrottenloher (CFS) solved this problem [CFS24]. They found a way to compute approximate modular exponentiations using only small intermediate values, destroying the anecdotal n qubit arithmetic bottleneck. However, their method is incompatible with "qubit recycling" [PP00; ME99], reviving an old bottleneck on the number of *input* qubits. Shor's original algorithm used $2n$ input qubits [Sho94], but Ekerå and Håstad proved in 2017 that $(0.5 + \epsilon)n$ input qubits was sufficient for RSA integers [EH17]. So, by combining May et al's result with Ekerå et al's result, the CFS algorithm can factor n bit RSA integers using only $(0.5 + \epsilon)n$ logical qubits [CFS24].

A notable downside of [CFS24] was its gate count. In [GE21], it was estimated that a 2048 bit RSA integer could be factored using 3 billion Toffoli gates and a bit more than $3n$ logical qubits. Whereas [CFS24] uses 2 trillion Toffoli gates and a bit more than $0.5n$ logical qubits. So [CFS24] is paying roughly 1000x more Toffolis for a 6x reduction in space. This is a strikingly inefficient spacetime trade-off. However, as I'll show in this paper, the trade-off can be made orders of magnitude more forgiving. In addition to optimizing the Toffoli count of the algorithm, I'll provide a physical cost estimate showing it should be possible to factor 2048 bit RSA integers in less than a week using less than a million physical qubits (under the assumptions mentioned in the abstract).

The paper is structured as follows. In Section 2, I describe a streamlined version of the CFS algorithm. In Section 3, I estimate its cost. I first estimate Toffoli counts and logical qubit counts, and then convert these into physical qubit cost estimates accounting for the overhead of fault tolerance. Finally, in Section 4, I summarize my results. The paper also includes Appendix A, which shows more detailed mock-ups of the algorithm and its physical implementation.

2 Methods

In this section, I present a variation of the CFS algorithm. The underlying ideas are the same, but many of the details are different. For example, I use fewer intermediate values and I extract the most significant bits of the result rather than the least significant bits.

Beware that, as in [GE21], for simplicity, I will describe everything in terms of period finding against $f(e) = g^e \bmod N$ (as in Shor's original algorithm) despite actually intending to use Ekerå-Håstad-style period finding [EH17]. Ultimately everything decomposes into a series of quantum controlled multiplications, which fundamentally is what is actually being optimized, so what I describe will trivially translate.

2.1 Approximate Residue Arithmetic

Consider an integer L equal to the product of many ℓ-bit primes from a set P:

$$L = \prod_{p \in P} p \tag{1}$$

Notation	Equivalent To	Description
len a	$\lceil \log_2 \max(1, a) \rceil$	Min bits needed to store an integer in the range $[0, a)$.
len q	-	Number of qubits in a quantum register q.
$a//b$	$\lfloor a/b \rfloor$	Floored division of a by b.
$a \bmod b$	$a - b \cdot (a//b)$	Remainder of a divided by b, canonicalized into $[0, b)$.
$a \bmod b \bmod c$	$(a \bmod b) \bmod c$	mod is left-associative.
$a \gg b$	$a//2^b$	Right shift of a by b bits.
$a \ll b$	$a2^b$	Left shift of a by b bits.
$\|a : b : c\rangle$	$\sum_{k=0}^{\lceil (b-a)/c \rceil - 1} \frac{\|kc + a\rangle}{\sqrt{\lceil (b-a)/c \rceil}}$	Uniform superposition, from a to $b-1$, stepping by c.
GRAD_a^b	$\sum_{k=0}^{a-1} \|k\rangle\langle k\| e^{ik2\pi b/a}$	Phase gradient modulo a, scaled by b.
QFT_a	$\frac{1}{\sqrt{a}} \sum_{k=0}^{a-1} \sum_{j=0}^{a-1} \|j\rangle\langle k\| e^{ijk2\pi/a}$	Quantum Fourier Transform modulo a.
$\Delta_N(a)$	$\min(a \bmod N, -a \bmod N)/N$	Modular deviation of a, relative to N.
$\text{int}(a)$	$\begin{cases} a & \to 1 \\ \neg a & \to 0 \end{cases}$	Indicator function for a boolean expression.
$\text{negif}(a)$	$(-1)^{\text{int}(a)}$	Phase flip indicator function for a boolean expression.
a_b	$(a \gg b) \mod 2$	Access bit (or qubit) b within integer (or quint) a.
$a \,\&\, b$	$\sum_k a_k b_k \ll k$	Bitwise AND of a and b.
$\text{MultiplicativeInverse}_a(b)$	$b^{-1} \pmod a$	Multiplicative inverse of b modulo a.

Table 1: Various bits of mostly-Python-inspired notation used in this paper.

$$\forall p \in P : (\text{len } p = \ell) \wedge \text{IsPrime(p)} \tag{2}$$

A simple way to do arithmetic on a value modulo L is to store the value as a 2s complement integer in a len L bit register, and when operating on this register add or subtract multiples of L as appropriate to keep it in the range $[0, L)$. This requires $\Omega(\log L)$ bits of storage.

Residue arithmetic performs addition and multiplication modulo L by separately performing the operations modulo each of the primes from P. The final result can then be recovered using the Chinese remainder theorem. By the prime number theorem, P can be chosen so that the bit length of primes in P is $\ell = \Theta(\log \log L)$. Therefore, using residue arithmetic introduces the possibility of arithmetic being exponentially more space efficient:

$$|P| = \Theta\left(\frac{\log L}{\log \log L}\right) \tag{3}$$

$$\ell = \Theta(\log \log L) \tag{4}$$

In the context of Shor's algorithm, the key operation we want to compute is a modular exponentiation $V = g^e \bmod N$. Here g is a randomly chosen classical constant in $[2, N)$, e is a uniformly superposed $m = O(\log N)$ qubit integer, and N is the number to factor. The computation of V can be decomposed into a series of multiplications controlled by the qubits of e:

$$V = g^e \bmod N = \left(\prod_{k=0}^{m-1} M_k^{e_k}\right) \bmod N \tag{5}$$

$$m = O(\log N) \tag{6}$$

In the above equation, e_k is the qubit at little-endian offset k within the register e and M_k is the classically precomputed constant

$$M_k = g^{(2^k)} \bmod N \tag{7}$$

Because the factors of N aren't known, and because in practice those factors would be large, it's not viable to perform residue arithmetic modulo N. However, performing the arithmetic modulo

any other modulus $L \neq N$ creates an issue. If the accumulating product exceeds L, then the register will wrap around. This shifts its value by a multiple of L and, since $L \bmod N \neq 0$, this offsets the tracked value relative to the true result mod N. The following multiplications would then amplify this error out of control, ruining the computation. I fix this in a simple way, the same way Chevignard et al fix it, by picking L to be larger than the largest possible product. This guarantees the wraparound issue never occurs:

$$L \geq N^m \tag{8}$$

With this promise about the size of L, we can rewrite the computation of V to use residue arithmetic. Instead of computing the product directly, we can compute it using a dot product between each residue r_j (the product modulo a prime from L) and its contribution factor u_j (the smallest positive integer satisfying $u_j \bmod p_i = \delta_{i,j}$):

$$r_j = \left(\prod_{k=0}^{m-1} M_k^{e_k} \right) \bmod p_j \tag{9}$$

$$u_j = (L/p_j) \cdot \text{MultiplicativeInverse}_{p_j}(L/p_j) \tag{10}$$

$$\begin{aligned} V &= \left(\prod_{k=0}^{m-1} M_k^{e_k} \right) \bmod L \bmod N \\ &= \left(\sum_{j=0}^{|P|-1} r_j u_j \right) \bmod L \bmod N \end{aligned} \tag{11}$$

To simplify later steps, I decompose r_j into its ℓ bits $r_{j,k}$, and write V in those terms:

$$r_j = \sum_{k=0}^{\ell-1} r_{j,k} \ll k \tag{12}$$

$$V = \left(\sum_{j=0}^{|P|-1} \sum_{k=0}^{\ell-1} r_{j,k} \Big[u_j \ll k \Big] \right) \bmod L \bmod N \tag{13}$$

The computation of V is now a sum modulo L then modulo N, instead of a product modulo N. Crucially, this fixes the issue where errors early in the process would be amplified by later operations. As a result, we're in a position to start using approximations.

Our goal now is to use approximations to extract the most significant bits of V, without computing the entirety of V. To quantify the error in these approximations, we'll track its "modular deviation" $\Delta_N(a-b)$. The modular deviation is the minimum number of increments or decrements needed to turn a into b modulo N, divided by N:

$$\Delta_N(a-b) = \frac{\min((a-b) \bmod N, (b-a) \bmod N)}{N} \tag{14}$$

To approximate additions modulo N, we'll truncate all values down to f bits. When asked to add an offset s into an accumulator, we'll instead add $(s \bmod N) \gg t$ where $t = \text{len } N - f$. The accumulator will be truncated to f bits, and will be operated modulo $N \gg t$ instead of modulo N. At any time, the approximate result can be extracted by left shifting the accumulator's value by t.

Truncating introduces two sources of error. First, during additions, carries that would have propagated out of the removed part of the register into the kept part of the register will no longer happen. Second, the accumulating additions wrap around slightly too quickly due to $(N \gg t) \ll t$ being less than N. For an individual truncated addition, both of these errors offset the approximate result by an amount $O(2^t)$, which is a modular deviation of $O(2^{-f})$. So the modular deviation is exponentially small in the number of kept bits, and will accumulate linearly with the number of operations, meaning a series of A truncated additions has a modular deviation of at most $O(A \cdot 2^{-f})$:

$$t = (\text{len } N) - f \tag{15}$$

$$\begin{aligned} S &= \left(\sum_{k=0}^{A-1} s_k\right) \bmod N \\ \widetilde{S} &= \left(\sum_{k=0}^{A-1} (s_k \bmod N) \gg t\right) \bmod (N \gg t) \\ \Delta_N(S - (\tilde{S} \ll t)) &\leq O(A \cdot 2^{-f}) \end{aligned} \tag{16}$$

This kind of approximated accumulation isn't quite directly applicable to the current expression for V. The issue is that V isn't being accumulated modulo N, it's being accumulated modulo L and only at the end is the modulo N performed. If we were to accumulate only modulo N, instead of modulo L then modulo N, then each time the accumulator would have wrapped modulo L we'd miss an offset of $L \bmod N$. However, L is the product of the primes in our residue system, and we get to choose these primes, so we can pull a trick. We can optimize L such that $L \bmod N$ has negligible modular deviation.

Numerically, it seems to be the case that picking random sets of small primes results in values of $L \bmod N$ uniformly distributed over the range $[1, N)$. In cases I've tested, I'm consistently able to find an L with deviation below 2^{-f} with high probability by randomly sampling $O(2^f)$ sets of small primes. I conjecture this is true in general (see Assumption 1). I'll show later in Equation 23 that the required value of f grows logarithmically with the problem size len N. Assuming the conjecture and the promise that f will grow like $O(\log \log N)$, an L with sufficiently small deviation can be found in polynomial time:

$$\Delta_N(L) < 2^{-f} \quad \text{(by brute force random search for a satisfying } P) \tag{17}$$

Assumption 1 (random search for small modular deviations is efficient): A set P of small primes satisfying $(\prod P) \geq N^m$ and $\Delta_N(\prod P) < 2^{-f}$ can be found in $O(2^f \cdot \text{poly}(m \cdot \text{len } N))$ expected time by randomly varying the primes included in P.

As an example, here's a set of 25000 primes I found in ten seconds using a 128 core machine. Each prime is 22 bits long, and the set achieves a modular deviation below 2^{-32} versus the RSA2048 challenge number [Wik24]:

$$\begin{aligned} P_1 &= \text{PrimesBetween}(3814620, 2^{22}) \\ P_2 &= \{2097769, 3484783, 3814501, 3814543, 3814561, 3814583, 3814609\} \\ L &= \prod (P_1 \cup P_2) \\ N &= \text{RSA}_{2048} \\ \Delta_N(L) &< 2^{-32} \end{aligned} \tag{18}$$

Given the promise that the modular deviation of the wraparound error is at most 2^{-f}, we can generalize Equation 16 from approximating arithmetic modulo N to approximating arithmetic modulo L then modulo N:

$$\begin{aligned} S &= \left(\sum_{k=0}^{A-1} s_k\right) \bmod L \bmod N \\ \widetilde{S} &= \left(\sum_{k=0}^{A-1} (s_k \bmod L \bmod N) \gg t\right) \bmod (N \gg t) \\ \Delta_N(S - (\tilde{S} \ll t)) &\leq O(A \cdot 2^{-f} + A \cdot \Delta_N(L)) \\ &\leq O(A \cdot 2^{-f}) \end{aligned} \tag{19}$$

We can now apply Equation 19 to Equation 13, producing an expression for $\widetilde{V} \approx V \gg t$:

$$\begin{aligned}\widetilde{V} &= \left(\sum_{j=0}^{|P|-1}\sum_{k=0}^{\ell-1} r_{j,k}\left(\left[(u_j \ll k) \bmod L \bmod N\right] \gg t\right)\right) \bmod (N \gg t) \\ &= \left(\sum_{j=0}^{|P|-1}\sum_{k=0}^{\ell-1} r_{j,k} C_{j,k}\right) \bmod (N \gg t)\end{aligned} \tag{20}$$

So, by using truncated residue arithmetic, the modular exponentiation has been approximated into an f-bit dot product between the bits of the residues and a table of classically precomputable constants $C_{j,k}$:

$$C_{j,k} = \left(\left[(u_j \ll k) \bmod L \bmod N\right] \gg t\right) \bmod (N \gg t) \tag{21}$$

The computation of $\widetilde{V}$ incurs $O(2^{-f})$ modular deviation per truncated addition, and there are $|P| \cdot \ell$ additions, so the total modular deviation of $\widetilde{V}$ is at most

$$\Delta_N(V - (\widetilde{V} \ll t)) \leq O(|P| \cdot \ell \cdot 2^{-f}) \tag{22}$$

It will be clear from the next subsection that, in order for period finding to work, it's sufficient to achieve a constant total modular deviation (e.g. a modular deviation of 10%). Solving for f in $|P| \cdot \ell \cdot 2^{-f} = O(1)$ after expanding terms using Equation 3, Equation 4, Equation 6, and Equation 8 gives a bound on the size of f:

$$f = O(\log \log N) \tag{23}$$

See Figure 3 for working Python code that computes $\widetilde{V} \approx g^e \bmod N$ using this method.

2.2 Approximate Period Finding

Shor's algorithm, as usually described, performs period finding against a periodic function $f(x)$. It relies crucially on the fact that $f(x)$ is *exactly* periodic, with $f(x + P) = f(x)$ for all values of x. In this paper I'm computing an approximation $\widetilde{f} \approx f$, so I can only guarantee *approximate periodicity*:

$$\forall x, y \in \mathbb{Z} : \Delta_N\left(\widetilde{f}(x + yP) - \widetilde{f}(x)\right) \leq \epsilon. \tag{24}$$

Exact period finding will fail if used on a function that only guarantees approximate periodicity, because the error in the approximation can make the function aperiodic. To fix this, we must avoid measuring the error. For example, instead of measuring the entire output of $\widetilde{f}(x)$ you could measure a truncated output $\widetilde{f}(x)//(100N\epsilon)$. This could work, but causes two problems. The first problem is that truncating the output means more inputs will be consistent with it. This will cause the system to collapse to a large superposition of periodic signals, instead of a simple periodic signal. I'll address this by analyzing the behavior of period finding against superpositions of periodic signals, as was done in [MS22]. The second problem with truncating the output is that the parts of the output that aren't measured would need to be uncomputed, by running the modular exponentiation process backwards. The modular exponentiation is already the most expensive part of the algorithm, so this would double the execution costs. I'll address this by using superposition masking.

Superposition masking uses sacrificial superpositions to control the information that can enter into a register or be revealed by measurements [Zal06; Gid19a; JG20]. In the present context, we want to avoid learning the low bits of $\widetilde{f}(x)$. We can achieve this by measuring $\widetilde{f}(x) + s$ instead of $\widetilde{f}(x)$, where s is a uniform superposition over a contiguous range $[0, \lceil SN \rceil)$. I'll address the value of S later; for now note that larger values of S will suppress error from the approximation but reduce the amount of information remaining for period finding.

Because $\widetilde{f}(x)$ happens to be computed with a series of additions into an output register, it's possible to compute and measure $\widetilde{f}(x) + s$ without needing to then uncompute $\widetilde{f}(x)$ or s. This is

```python
import math


def approximate_modular_exponentiation(
        g: int,
        Q_e: int,
        N: int,
        P: list[int],
        f: int,
) -> int:
    """Computes an approximate modular exponentiation.

    Args:
        g: The base of the exponentiation.
        Q_e: The exponent.
        N: The modulus.
        P: Small primes for the residue arithmetic.
        f: Kept bits during approximate accumulation.

    Returns:
        An approximation of pow(g, Q_e, N).

        The modular deviation of the approximation is
        at most 3 * Q_e.bit_length() / 2**f .
    """
    L = math.prod(P)
    ell = max(p.bit_length() for p in P)
    m = Q_e.bit_length()
    t = N.bit_length() - f
    assert L == math.lcm(*P) and L >= N ** m and (L % N) < (N >> f)

    Q_total = 0
    for p in P:
        Q_residue = 1
        for k in range(Q_e.bit_length()):
            precomputed = pow(g, 1 << k, N) % p

            # controlled inplace modular multiplication:
            if Q_e & (1 << k):
                Q_residue *= precomputed
                Q_residue %= p

        u = (L // p) * pow(L // p, -1, p)
        for k in range(ell):
            precomputed = (((u << k) % L % N) >> t) % (N >> t)

            # controlled inplace modular addition:
            if Q_residue & (1 << k):
                Q_total += precomputed
                Q_total %= N >> t

    return Q_total << t


def test_approximate_modular_exponentiation():
    import random
    import sympy

    # RSA100 challenge number
    N = int(
        "152260502792253336053561837813263742971806811496 13"
        "80688657908494580122963258952897654000350692006139"
    )
    g = random.randrange(2, N)
    Q_e = random.randrange(2**100)  # small exponent for quicker test
    P = [
        *sympy.primerange(239382, 2**18),
        131101,
        131111,
        131113,
        131129,
        131143,
        131149,
        131947,
        182341,
        239333,
        239347,
    ]
    f = 24
    result = approximate_modular_exponentiation(g, Q_e, N, P, f)
    error = result - pow(g, Q_e, N)
    deviation = min(error % N, -error % N) / N
    ell = max(p.bit_length() for p in P)
    assert deviation <= 3 * len(P) * ell / 2**f
```

Figure 3: Example Python code that approximates $\widetilde{V} \approx g^e \bmod N$ given a choice of P and f. Registers that would store quantum values during the quantum factoring are prefixed with "Q_". All would-be-quantum registers are of size $O(\log \log N)$, except the input register Q_e. For simplicity, this code omits optimizations (like windowing) and crucial quantum details (like uncomputation). See Appendix A.1 for more detailed code.

done by initializing the output register to s (instead of 0), then adding the approximation $\widetilde{f}(x)$ into the output register, and then measuring the entire output register. To understand this process in detail, let's analyze the states that occur as the algorithm progresses.

At the beginning of the algorithm, the output register is storing a uniform superposition (the mask). There is also the input register: an m qubit register storing its own uniform superposition. Together these two registers form the initial state $|\psi_0\rangle$:

$$\begin{aligned}|\psi_0\rangle &= |0 : 2^m : 1\rangle \otimes |0 : \lceil SN \rceil : 1\rangle \\ &= \frac{1}{\sqrt{2^m \lceil SN \rceil}} \sum_{e=0}^{2^m-1} \sum_{s=0}^{\lceil SN \rceil - 1} |e\rangle \otimes |s\rangle\end{aligned} \tag{25}$$

The algorithm now adds the approximation $\widetilde{f}(e)$ into the output register, modulo N, forming the actual pre-measurement state $|\widetilde{\psi_1}\rangle$:

$$|\widetilde{\psi_1}\rangle = \frac{1}{\sqrt{\lceil SN \rceil 2^m}} \sum_{e=0}^{2^m-1} \sum_{s=0}^{\lceil SN \rceil - 1} |e\rangle \otimes |(s + \widetilde{f}(e)) \bmod N\rangle \tag{26}$$

If we had used f instead of $\widetilde{f}$, we would have produced the *ideal* pre-measurement state $|\psi_1\rangle$:

$$|\psi_1\rangle = \frac{1}{\sqrt{\lceil SN \rceil 2^m}} \sum_{e=0}^{2^m-1} \sum_{s=0}^{\lceil SN \rceil - 1} |e\rangle \otimes |(s + f(e)) \bmod N\rangle \tag{27}$$

Consider the infidelity $1 - |\langle \psi_1 | \widetilde{\psi_1} \rangle|^2$. For each possible computational basis value e of the input register, conditioning $|\widetilde{\psi_1}\rangle$ and $|\psi_1\rangle$ on the input register storing e produces a uniform superposition in the output register covering $\lceil SN \rceil$ contiguous values. If the maximum modular deviation of $\widetilde{f}$ is ϵ, then the conditioned ranges in $|\widetilde{\psi_1}\rangle$ are offset from the conditioned ranges in $|\psi_1\rangle$ by at most $N\epsilon$. The width of the ranges is at least S/ϵ times wider than the offset from the deviation. Therefore the infidelity between the two conditioned states is a most ϵ/S. But this is true for *every* condition, and so also bounds the total infidelity between the two states:

$$1 - |\langle \psi_1 | \widetilde{\psi_1} \rangle|^2 \leq \epsilon / S \tag{28}$$

We can treat this infidelity as contributing a chance of the algorithm failing, and thereby continue the analysis assuming we are working with the ideal state $|\psi_1\rangle$ instead of the actual state $|\widetilde{\psi_1}\rangle$. For example, if we set $S = 1000\epsilon$ then $|\widetilde{\psi_1}\rangle$ would have 99.9% overlap with $|\psi_1\rangle$ and so any success rate analysis we did with $|\psi_1\rangle$ would produce a result within 0.1% of the true success rate.

The next step in the algorithm, which we'll analyze with $|\psi_1\rangle$ instead of $|\widetilde{\psi_1}\rangle$, is to measure the output register. This produces some measurement result V, and collapses the system into the post-measurement state $|\psi_2\rangle$. This state is a superposition of all the exponents whose masked output ranges overlapped with V:

$$|\psi_2\rangle \propto \sum_{e=0}^{2^m-1} |e\rangle \cdot \text{int}\left(0 \leq (V - f(e)) \bmod N < \lceil NS \rceil\right) \tag{29}$$

Let P be the period of f, and let R be the set of exponents less than P that are consistent with the measured value V. We can rewrite $|\psi_2\rangle$ in these terms:

$$R = \left\{ e \in \mathbb{N} \;\middle|\; e < P \wedge 0 \leq (V - f(e)) \bmod N < \lceil NS \rceil \right\} \tag{30}$$

$$|\psi_2\rangle \approx \frac{1}{\sqrt{|R|}} \sum_{e \in R} |e : P : 2^m\rangle \tag{31}$$

The above equality is approximate because 2^m isn't guaranteed to be a multiple of P. Some of the states $|e : P : 2^m\rangle$ have $2^m // P$ non-zero amplitudes while others may have $2^m // P + 1$. To avoid annoyances like this, I'll continue the analysis while pretending that the signal and quantum

Fourier transform are being worked on modulo a multiple of the period (MP) rather than modulo a power of 2 (2^m). In reality we must operate modulo 2^m, since we don't know P, but it's well known that the modulo-2^m behavior approximates the modulo-MP behavior [Cop02]. The error in the approximation decreases exponentially as m increases, because the overlap between a candidate frequency $\widetilde{\theta}$ and a true frequency θ contains a 2^m scaling factor that forces the range of high-overlap candidate frequencies to tighten towards the true frequencies:

$$\begin{aligned} |\theta\rangle &= \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} |k\rangle e^{i\theta k} \\ \left|\langle\theta|\widetilde{\theta}\rangle\right|^2 &= \left|\frac{1}{2^m} \sum_{k=0}^{2^m-1} e^{i(\theta-\widetilde{\theta})k}\right|^2 \approx \operatorname{sinc}^2\left(2^m(\theta-\widetilde{\theta})/2\right) \end{aligned} \tag{32}$$

So let's pretend we have a post-measurement state modulo MP instead of modulo 2^m:

$$|\psi_2'\rangle = \frac{1}{\sqrt{|R|}} \sum_{e\in R} |e : P : MP\rangle \tag{33}$$

The state $|r : MP : P\rangle$ is a "simple periodic signal". Shor proved that period finding works on simple periodic signals. The state $|\psi_2'\rangle$ isn't a simple periodic signal; it's a superposition of such signals. In [MS22] it was shown that period finding works on these states, as long as the contents of R are randomized. For completeness I'll reproduce a similar argument here, and conjecture that this randomized analysis applies to the actual values of R that appear when factoring.

The modulo-MP quantum Fourier transform of a simple periodic signal $|r : MP : P\rangle$ is a superposition with non-zero amplitudes at multiples of $MP/P = M$. The non-zero amplitudes all have equal magnitude, but their phases depend on r:

$$\mathrm{QFT}_{MP} |r : MP : P\rangle = \mathrm{GRAD}^r_{MP} |0 : MP : M\rangle \tag{34}$$

Because the QFT of every simple periodic signal only has non-zero amplitudes at multiples of M, the QFT of any superposition of simple periodic signals can also only have non-zero amplitudes at multiples of M. No new frequency peaks can appear. However, there is still the possibility for interference amongst the existing frequency peaks. If each simple periodic signal $|r : MP : P\rangle$ is assigned an amplitude α_r then we find:

$$\begin{aligned} \mathrm{QFT}_{MP} \sum_{r=0}^{P-1} \alpha_r |r : MP : P\rangle &= \sum_{r=0}^{P-1} \alpha_r \mathrm{GRAD}^r_{MP} |0 : MP : M\rangle \\ &= \sum_{k=0}^{P-1} \left(\frac{1}{\sqrt{P}} \sum_{r=0}^{P-1} \alpha_r e^{i2\pi rk/P}\right) |kM\rangle \\ &= \sum_{k=0}^{P-1} \beta_k |kM\rangle \end{aligned} \tag{35}$$

where β_k is the amplitude of the frequency peak at $|kM\rangle$:

$$\beta_k = \frac{1}{\sqrt{P}} \sum_{r=0}^{P-1} \alpha_r e^{i2\pi rk/P} \tag{36}$$

In the actual algorithm, the amplitudes of the simple periodic signals are zero for remainders outside a set R, and equal for all remainders in R:

$$\alpha_r = \frac{\mathrm{int}(r \in R)}{\sqrt{|R|}} \tag{37}$$

Let $w = |R|$ and assume that R was chosen uniformly at random from the $\binom{P}{w}$ possible sets of w remainders modulo P. Our goal is to derive a simple expression for $E(|\beta_k|^2)$, the expected rate of

measuring the k'th peak, given this assumption. Start by expanding definitions from Equation 36 and Equation 37, averaged over possible values of R:

$$\begin{aligned}
E(|\beta_k|^2) &= \frac{1}{\binom{P}{w}} \sum_{\substack{R \in \mathcal{P}([0,P)) \\ |R|=w}} \left| \frac{1}{\sqrt{P}} \sum_{r=0}^{P-1} \frac{\text{int}(r \in R)}{\sqrt{|R|}} e^{ikr2\pi/P} \right|^2 \\
&= \frac{1}{\binom{P}{w} wP} \sum_{\substack{R \in \mathcal{P}([0,P)) \\ |R|=w}} \sum_{r_1=0}^{P-1} \sum_{r_2=0}^{P-1} \text{int}(r_1 \in R) \cdot \text{int}(r_2 \in R) \cdot e^{ikr_1 2\pi/P} e^{-ikr_2 2\pi/P}
\end{aligned} \tag{38}$$

When $k = 0$, all the $e^{ik\cdots}$ terms simplify to 1 and the overall expression simplifies to $E(|B_0|^2) = w/P$. So focus on the case where $k > 0$.

The sum over values of R can be pushed rightward, so that it just counts how many values of R are compatible with a choice of (r_1, r_2). When $r_1 = r_2$, there are $\binom{P-1}{w-1}$ values of R that satisfy $r_1 \in R \wedge r_2 \in R$. When $r_1 \neq r_2$, there are $\binom{P-2}{w-2}$ satisfying values. This replaces the sum over values of R with a choice of multiplier determined by $r_1 \stackrel{?}{=} r_2$:

$$\begin{aligned}
E(|B_{k>0}|^2) &= \frac{1}{\binom{P}{w} wP} \sum_{r_1=0}^{P-1} \sum_{r_2=0}^{P-1} e^{ikr_1 2\pi/P} e^{-ikr_2 2\pi/P} \left(\sum_{\substack{R \in \mathcal{P}([0,P)) \\ |R|=w}} \text{int}(r_1 \in R \wedge r_2 \in R) \right) \\
&= \frac{1}{\binom{P}{w} wP} \sum_{r_1=0}^{P-1} \sum_{r_2=0}^{P-1} e^{ir_1 k2\pi/P} e^{-ir_2 k2\pi/P} \begin{cases} r_1 = r_2 & \to \binom{P-1}{w-1} \\ r_1 \neq r_2 & \to \binom{P-2}{w-2} \end{cases}
\end{aligned} \tag{39}$$

Note that, for non-zero values of k, the nested sum in the above equation would evaluate to 0 if the right side multiplier was replaced by a constant:

$$0 < k < P \implies \sum_{r_1=0}^{P-1} \sum_{r_2=0}^{P-1} e^{ir_1 k2\pi/P} e^{-ir_2 k2\pi/P} = 0 \tag{40}$$

This allows the right hand multiplier to be offset by any fixed amount, without changing the total. I choose to offset by $-\binom{P-2}{w-2}$, zeroing the multiplier for the $r_1 \neq r_2$ case, leaving behind only the $r_1 = r_2$ cases. The expression then simplifies to a fraction independent of k:

$$\begin{aligned}
E(|B_{k>0}|^2) &= \binom{P}{w}^{-1} \frac{1}{wP} \sum_{r=0}^{P-1} e^{irk2\pi/P} e^{-irk2\pi/P} \left(\binom{P-1}{w-1} - \binom{P-2}{w-2} \right) \\
&= \binom{P}{w}^{-1} \frac{1}{wP} P \left(\binom{P-1}{w-1} - \binom{P-2}{w-2} \right) \\
&= \frac{P-w}{P(P-1)}
\end{aligned} \tag{41}$$

In other words, when w randomly chosen simple periodic signals are superposed, the system behaves as if β_0 takes a w/P cut and then $\beta_1, \ldots, \beta_{P-1}$ equally split whats left:

$$E(|\beta_k|^2) = \begin{cases} k = 0 & \to \frac{w}{P} \\ 0 < k < P & \to \left(1 - \frac{w}{P}\right) \cdot \frac{1}{P-1} \end{cases} \tag{42}$$

Therefore, in the randomized case, period finding against a superposition of w simple periodic signals has a w/P probability of failing due to sampling 0. In the usual period finding algorithm, the probability of sampling 0 would be $1/P$ instead of w/P. The randomized case otherwise behaves identically to the usual period finding used in Shor's algorithm.

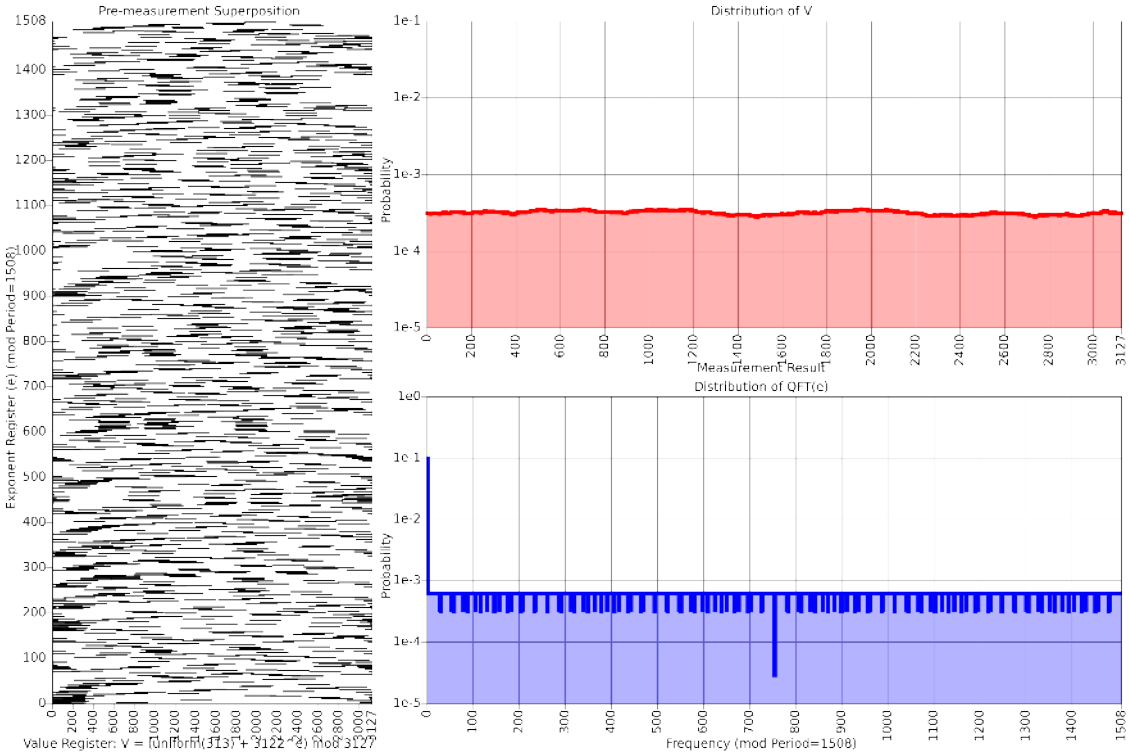

Figure 4: A small-scale example of how masking affects period finding. Left: a pre-measurement state, with white meaning zero amplitude and black meaning equal non-zero amplitude. Top right: probability of measuring different values of the output register. Masking has smoothed the distribution to be nearly uniform. Bottom right: probability of measuring different frequency peaks of the input register. Appears uniform (instead of spiky) because the frequencies are computed modulo the period P. Without masking, the bottom right distribution would be exactly uniform. With masking, the zero'th frequency is substantially more likely due to constructive interference from the masking. Certain frequencies are also less likely.

Of course, in an actual factoring, the set R isn't random. It's a deterministic function of N, g, and the sampled measurement. The success rate won't necessarily be suppressed by the factor $1 - w/P \approx 1 - S$ that it would be in the random case. In Figure 4, I show an example case. You can see in that figure that the frequency spectrum has some dips. These wouldn't be present in the randomized case, and will skew the success rate. In Figure 5, I show success rate numerics for different values of S and N. The first thing I notice when looking at the data is that the success rate appears to skew slightly higher than $1 - S$. Some instances are slightly worse than $1 - S$ but most are slightly better. The average success rate is higher than I expected. It also seems that the variation in success rate is small (there's no instance where a masking proportion of 0.1 produced a success suppression factor below $1 - 0.2$ or where a masking proportion of 0.01 produced a success suppression factor below $1 - 0.03$). When collecting data for the figure, I was hoping the variations wouldn't just be small but would noticeably decrease as N was increased. No such effect is apparent in the data (keeping mind that the plotted values of N are tiny compared to 2^{2048}).

Although the numerics show that the real case isn't exactly identical to the randomized case, they appear close enough that, for my purposes in this paper, the real costs can be estimated by using the randomized analysis. I will estimate costs as if a superposition mask proportion of S incurs a probability S of the shot failing and a probability $1 - S$ of the shot behaving as if no masking was present (see Assumption 2).

> **Assumption 2 (behavior of masked period finding)**: When using a superposition mask that covers a proportion S of the output space, the cost of using masking can be roughly estimated by multiplying the expected number of shots by $1/(1 - S)$.

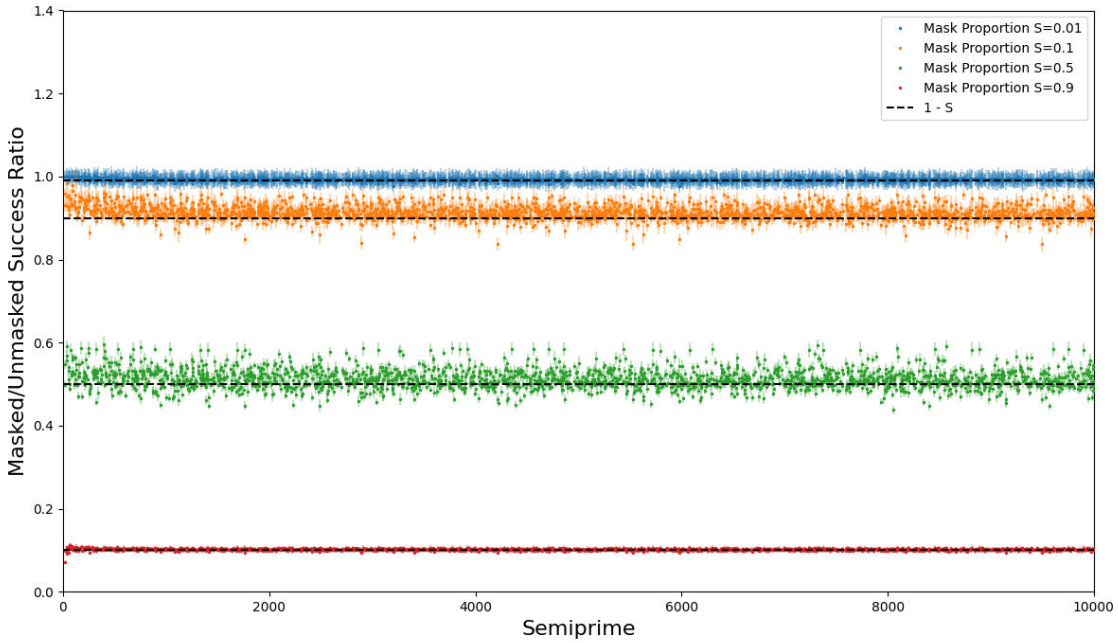

Figure 5: Suppression of success rate due to the use of superposition masking, estimated by Monte Carlo sampling. Error bars are computed by separately computing likelihoods for the masked samples and the unmasked samples, finding the respective high/low hypothesis probabilities that have a likelihood within 100x of the max likelihood hypothesis given the sampled data, and then making the smallest/largest ratios possible with those high and low values.

Let's now return to choosing the proportional width S of the superposition mask. Recall from Equation 28 that the maximum infidelity of the analysis, caused by using an approximation $\widetilde{f}$ with a modular deviation of at most ϵ, is ϵ/S. Also recall from Assumption 2 that a mask covering a proportion S of the output space behaves like an S chance of failure. The sum $S+\epsilon/S$ of these two error mechanisms is an upper bound on the chance P_{deviant} of a shot failing due to using masking and approximations:

$$P_{\text{deviant}} \leq S + \epsilon/S \tag{43}$$

Minimizing this sum produces a choice for S, the proportional size of the mask:

$$S = \sqrt{\epsilon} \tag{44}$$

This implies that an approximation with modular deviation ϵ causes a failure rate of at most

$$P_{\text{deviant}} \leq 2\sqrt{\epsilon} \tag{45}$$

This could likely be improved, for example by using a Gaussian mask instead of a uniform mask or by analyzing the distribution of deviations rather than focusing on the worst case deviation. But $2\sqrt{\epsilon}$ suffices for my purposes in this paper, so I leave such optimizations to future work.

2.3 Ekerå-Håstad Period Finding

Elsewhere in the paper, I describe computations in terms of period finding against the function $f(e) = g^e \pmod N$. This would require an input register with $2n$ qubits [Sho94], where $n = \text{len } N$. As in Chevignard et al's paper, I actually use Ekerå-Håstad-style period finding [EH17] instead of Shor-style period finding. Ekerå-Håstad-style period finding is specialized to the RSA integer case (it requires that N factors into two similarly-sized primes), but uses $n/2 + n/s$ input qubits instead of $2n$.

Ekerå-Håstad-style period finding works as follows. The algorithm is given a positive integer parameter (s). A classical value g is chosen, uniformly at random, from $\mathbb{Z}_N^*$ (the multiplicative group modulo N). A second classical value $h = g^{N-1} \bmod N$ is derived from g and N. An $n/2 + \lceil n/(2s) \rceil$ qubit register (x) is prepared into a uniform superposition. An $\lceil n/(2s) \rceil$ qubit register (y) is similarly prepared into a uniform superposition. The value $g^x h^y \pmod N$ is computed under superposition, and discarded. Then x and y are measured in the frequency basis, producing a data point (x', y'). This is repeated at least $s + 1$ times. Post-processing then recovers, with high probability, $d = \text{dlog}_{g \pmod N}(h)$ from the list of recorded points. With high probability it will be the case that $d = p + q - 2$, where p, q are the prime factors, because

$$\begin{aligned} g^d \bmod N &= h \bmod N \\ &= g^{N-1} \bmod N \\ &= g^{pq-1} \bmod N \\ &= g^{(p-1)(q-1)+p+q-2} \bmod N \\ &= g^{p+q-2} \bmod N \end{aligned} \tag{46}$$

The factors are then recovered by solving for p in the quadratic equation $p \cdot (d - p + 2) = N$.

A detailed analysis of the post-processing and the expected number of repetitions is available in [Eke20] (see table 1 of that paper in particular). In this paper, I'll stay in the regime where the expected number of repetitions is $s + 1$. And of course $g^x h^y \pmod N$ will be computed approximately, instead of exactly, since it compiles into a series of controlled multiplications the same way $g^e \bmod N$ would have.

Because Ekerå-Håstad-style period finding involves combining multiple shots, I should discuss how to handle bad shots. Bad shots caused by masking are usually easy to detect, because the most common failure should be the quantum computer returning the integer 0 rather than a useful value. On the other hand, bad shots caused by a logical gate error during the computation are essentially silent. They manifest as the postprocessing failing despite having collected $s + 1$ shots. When this occurs, as long as errors are bounded to reasonable rates, it's sufficient to take a few extra shots while running the postprocessing on all $\binom{t}{s+1}$ possible combinations of shots (where t is the total number of shots). If t grows too large to reasonably check all the combinations, restart with a different choice of g.

Recall from Equation 8 that L, the size of the residue system, must be at least N^m (where m is the number of multiplications or equivalently the number of input qubits). Because increasing s reduces the number of input qubits, it reduces the number of multiplications which then reduces the number of primes in the residue system which then reduces the total amount of work that needs to be done. This compounded benefit notably improves the spacetime tradeoff of increasing s. For example, using $s = 3$ results in a computation that is both smaller *and* faster than using $s = 1$ despite the overhead of taking additional shots.

2.4 Arithmetic Optimizations

The example code I showed in Figure 3 is simple, but inefficient. In this subsection, I describe optimizations that reduce its cost by orders of magnitude.

The first optimization is to replace modular multiplications with modular additions, by classically precomputing short discrete logarithms. This optimization was introduced in [CFS24], but I describe it here for completeness.

Recall that, for each prime p in the residue system, we need to quantum compute a controlled product to get p's residue $V_p = V \bmod p$:

$$V_p = \prod_{k=0}^{m-1} M_k^{e_k} \pmod p \tag{47}$$

Because p is small (e.g. 22 bits long), it's feasible to find a multiplicative generator g_p modulo p and to solve for $D_{p,k}$ in

$$g_p^{D_{p,k}} \bmod p = M_k \bmod p \tag{48}$$

Note that this equation doesn't have a solution when $M_k \bmod p = 0$. In [CFS24], they track the values of p and M_k for which this occurs, and handle them as a special case on the quantum computer. I instead avoid this corner case by making it a selection criteria of the residue number system that it doesn't occur. That is to say, I require that

$$\forall p \in P, \forall x \in M : p \bmod x \neq 0 \tag{49}$$

where P is the desired set of primes making up the residue system and M is a given set of all possible multipliers. This criterion implies the residue system can only be chosen *after* picking the random generator g used by Shor's algorithm, since M depends on g. (M also depends on windowing parameters I'll describe later.) Consequently, I use different residue systems even when targeting the same value of N. Choosing the residue system is a per-shot cost, not a per-factoring cost.

To ensure it's possible to efficiently find a satisfying residue system, primes failing Equation 49 need to be excluded *before* making the random variations mentioned in Assumption 1. In the rare event that too many primes are excluded, resulting in a search space so small that a value of P satisfying Equation 17 is unlikely to exist, just increment the target prime bit length ℓ. This will roughly double the number of candidate primes, and roughly halve the chance of each candidate prime failing Equation 49, and so should quickly guarantee a solution exists if repeated.

Every value $D_{p,k} = \operatorname{dlog}(g_p, M_k, p)$ can be precomputed before starting the quantum computation. As a result, the controlled product computation is replaced by a controlled sum computation:

$$S_p = \sum_{k=0}^{m-1} D_{p,k} \cdot e_k \tag{50}$$

You can then compute V_p from S_p using a modular exponentiation. To further reduce costs, use the fact that the order of g_p is known to be $p-1$ and compress S_p into $S_p \bmod (p-1)$ before exponentiating:

$$V_p = g_p^{S_p \bmod (p-1)} \bmod p \tag{51}$$

Note that this exponentiation performs multiplications, which is the operation we were trying to avoid. However, crucially, the number of multiplications is now len p (i.e. dozens) instead of $O(\log N)$ (i.e. thousands).

The second optimization that I apply is windowing [Kut06; VI05; Gid19b]. When computing S_{p_j} from e, iterate over e in chunks of w_1 qubits at a time instead of 1 qubit at a time. The w_1 qubits are used as the address of a lookup into a classical table of 2^{w_1} values, each being the discrete logarithm of the combined product that would have been multiplied into the total if the qubits took on a specific value. Performing the lookup has a cost of 2^{w_1}, but reduces the number of iterations of the inner loop computing S_{p_j} from m to $\lceil m/w_1 \rceil$.

Similar to increasing the Ekerå-Håstad parameter s, increasing w_1 gives compounded benefits. A larger value of w_1 batches more multiplications together, which reduces the effective number of multiplications, which by Equation 8 reduces the required size of the residue system.

I also use windowing when computing V_{p_j} from S_{p_j}. The method is essentially identical to what's shown in [Gid19b], except that two multiplications are saved by initializing the accumulator using a lookup. This was independently suggested in [LNS25].

I also use windowing when adding approximate offsets into the final total, but only because lookups were needed in that loop anyways. It has a negligible impact on the overall cost of the algorithm. Lookups could also be used when reducing S_{p_j} modulo p_j, with similarly negligible impact.

The third optimization I apply is to merge the computation of S_{p_j} with the uncomputation of $S_{p_{j-1}}$. In the transition from iteration $j-1$ to iteration j, I could uncompute $S_{p_{j-1}}$ by a series of controlled subtractions $\sum_{k=0}^{m-1} -D_{p_{j-1},k-1} \cdot e_k$ and then compute S_j by a series of controlled additions $\sum_{k=0}^{m-1} D_{p_j,k} \cdot e_k$. But, because these use the same controls e_k, these two things can be merged. $S_{p_{j-1}}$ can be mutated into S_{p_j} by performing controlled additions of precomputed per-exponent-qubit differences $D_{p_j,k} - D_{p_{j-1},k}$:

$$D'_{j,k} = D_{p_j,k} - D_{p_{j-1},k} \tag{52}$$

$$S_{p_j} = S_{p_{j-1}} + \sum_{k=0}^{m-1} D'_{p_j,k} \cdot e_k \tag{53}$$

This is a simple optimization, significant only because computing each value of S_{p_j} is a substantial portion of the cost of the algorithm.

The fourth important optimization that I apply is deferring and merging phase corrections. In many places in the algorithm, measurement based uncomputations produce phasing tasks. These phasing tasks don't need to be performed immediately, and often the phasing tasks produced in one place can be merged with phasing tasks produced in another place. I do this so many times throughout the implementation that it would be tedious to list them all. But, for example, consider that a lookup performed by unary iteration [Bab+18] has AND gate uncomputations that correspond to CZ gates. The later uncomputation of the output of the lookup will produce substantially more complex phase corrections [Ber+19]. The CZ corrections produced by the computation are a subset of the phase corrections that can occur during the uncomputation, and the CZ corrections can be deferred until the uncomputation. So the CZ corrections can be merged into the uncomputation. From a cost perspective, this deletes the CZ corrections.

A fifth optimization I apply is to flip each modular addition `X += T[b] (mod p)` into a subtraction `X -= T2[b] (mod p)` (where `T2[b] = p - T[b]` is a trivially modified lookup table). The benefit of doing this is that an addition exceeding `p` requires an additional comparison to detect, but a subtraction underflowing 0 is easy to detect by concatenating an extra qubit `Q` to the top of the `X` register. `Q` will flip to 1 iff the subtraction underflows. So, after the subtraction, the modular addition can be completed by performing `X += [0, p][Q]` and then uncomputing `Q` using measurement based uncomputation. The uncomputation requires a phase correction 50% of the time, where the phase correction is based on a comparison. This modular adder costs $2.5n \pm O(1)$ Toffolis gates in expectation (compared to $3.5n \pm O(1)$ in [Ber+24]). If the modular addition is later uncomputed, its cost can be further reduced to $2n \pm O(1)$ by deferring and merging the phase correction.

See Appendix A.1 for reference Python code implementing the optimized arithmetic.

3 Results

3.1 Logical Costs

Essentially all work performed by the algorithm is addition, lookup, and "phaseup" operations:

- An n qubit addition is an operation that acts on an n qubit offset register (a) and an n qubit target register (b). It performs the classical transition $(a, b) \to (a, (b + a) \bmod 2^n)$ and has a Toffoli cost of $n - 1$ as well as an ancilla cost of $n - 1$ [Gid18]. Note that, for the purposes of cost estimation, subtractions and comparisons are counted as additions because their circuits are nearly identical.

- A lookup is an operation, parameterized by a classical table of values T, that acts on an n qubit address register (a) and a quantum output register. It performs the classical transition $(a, 0) \to (a, T_a)$ and has a Toffoli cost of $2^n - n - 1$ as well as an ancilla cost of $n - 1$ [Bab+18].

- A "phaseup" is an operation, parameterized by a classical table of values T, that acts on an n qubit target register (a). It negates the amplitudes of states that have a non-zero entry in the table, and has a Toffoli cost of $\sqrt{2^n} \pm O(n)$ [Ber+19].

To estimate total cost, I first estimate the numbers and sizes of these three operations by counting occurrences in the reference code in Appendix A.1. Table 3 shows symbolic tallies of these quantities. I also count qubit allocations and deallocations, to list symbolic qubit tallies in Table 4. The symbolic tallies can be turned into numeric estimates by choosing parameter values (see Table 2) and substituting.

To pick parameters, I ran a grid scan. The Ekerå-Håstad parameter s was ranged from 2 to 14. The prime bit length ℓ used by the residue arithmetic system was ranged from 18 to 25. The window size w_1 used by loop1 was ranged from 2 to 8. The window size $w_3 = w_{3a} = w_{3b}$ used by loop3 and unloop3 was ranged from 2 to 6. The window size w_4 used by loop4 was ranged from 2 to 8. The length f of the truncated accumulator was ranged from 24 to 59. Infeasible combinations were discarded (e.g. if the prime bit length is too small, it won't be possible to find a set of primes $\prod P \geq N^{m/w_1}$).

The best performing parameter combinations are shown as Pareto frontiers in Figure 2. I also chose to highlight, in Table 5, parameters that minimized q^3t where q is the qubit count and t is the Toffoli count. Cubing q is just an arbitrary way of favoring space savings more than time savings.

Symbol	Equivalent to	Asymptotic	Description
N	-	$\theta(2^n)$	Number to factor.
n	len N	$\theta(n)$	Bit size of number to factor.
ℓ	-	$\theta(\log n)$	Bit size of primes in residue system.
s	-	$\theta(1)$	Ekerå-Håstad parameter.
m	$\lceil n/2 + n/s \rceil$	$\theta(n)$	Number of input qubits.
f	-	$\theta(\log n)$	Size of output accumulator.
w_1	$\log_2(\ell + \text{len } m) \pm O(1)$	$\theta(\log \log n)$	Window length used by loop 1.
w_3	$\log_2(\ell)/2 \pm O(1)$	$\theta(\log \log n)$	Window length used by loop 3 and unloop 3.
w_4	$\log_2(\ell) \pm O(1)$	$\theta(\log \log n)$	Window length used by loop 4.
W_1	$\lceil m/w_1 \rceil$	$\theta(n/\log \log n)$	Number of windows iterated by loop 1.
W_3	$\lceil \ell/w_3 \rceil$	$\theta(\log n/\log \log n)$	Number of windows iterated by loop 3.
W_4	$\lceil \ell/w_4 \rceil$	$\theta(\log n/\log \log n)$	Number of windows iterated by loop 4.
$\|P\|$	$\approx nm/(\ell \cdot w_1)$	$\theta(n^2/(\log n \log \log n))$	Number of primes in residue system.

Table 2: Descriptions of variables used in cost estimates.

Subroutine	Iterations	Register Size	Address Size	Additions	Lookups	Phaseups	Toffolis
loop1	$(\|P\| + 1) \cdot W_1$	$\ell + \text{len } m$	w_1	1	1	0	$\widetilde{\Theta}(n^3)$
loop2	$\|P\| \cdot \text{len } m$	$\ell + \text{len } m$	-	2	0	0	$\widetilde{\Theta}(n^2)$
loop3 (startup)	$\|P\|$	ℓ	$2w_3$	0	1	0	$\widetilde{\Theta}(n^2)$
loop3 (body)	$\|P\| \cdot (W_3 - 2) \cdot W_3$	ℓ	w_3	2	1	0	$\widetilde{\Theta}(n^2)$
loop4	$\|P\| \cdot W_4$	f	w_4	1.5	2.5	1	$\widetilde{\Theta}(n^2)$
unloop3 (body)	$\|P\| \cdot (W_3 - 2) \cdot 2 \cdot W_3$	ℓ	w_3	2.5	1.5	1	$\widetilde{\Theta}(n^2)$
unloop3 (cleanup)	$\|P\|$	ℓ	$2w_3$	0	0	1	$\widetilde{\Theta}(n^2)$
unloop2	$\|P\| \cdot \text{len } m$	$\ell + \text{len } m$	-	2	0	0	$\widetilde{\Theta}(n^2)$

Table 3: Symbolic tallies of additions, lookups, and phaseups used by the subroutines of the algorithm. The variables being used are defined in Table 2.

Subroutine	Added Qubits	Temporary Qubits	Total Qubits
Algorithm Startup	$m + f$	0	$m + f$
Enter Outer Loop	$\ell + \text{len } m$	0	$m + f + \ell + \text{len } m$
loop1	0	$2(\ell + \text{len } m)$	$m + f + 3\ell + 3 \text{ len } m$
loop2	0	$\ell + \text{len } m$	$m + f + 2\ell + 2 \text{ len } m$
loop3	ℓ	2ℓ	$m + f + 4\ell + \text{len } m$
loop4	0	$2f$	$m + 3f + 2\ell + \text{len } m$
unloop3	$-\ell$	2ℓ	$m + f + 4\ell + \text{len } m$
unloop2	-0	$\ell + \text{len } m$	$m + f + 2\ell + 2 \text{ len } m$
Exit Outer Loop	$-\ell - \text{len } m$	$2(\ell + \text{len } m)$	$m + f + 3\ell + 3 \text{ len } m$
Measure Approximate Result	$-f$	0	$m + f$
Frequency Measurement	$-m$	$O(\log(1/\epsilon))$	$m + O(\log(1/\epsilon))$

Table 4: Symbolic tallies of qubits used by parts of the algorithm. The variables being used are defined in Table 2.

n	s	ℓ	w_1	w_3	w_4	f	m	P_{deviant}	E(shots)	Toffolis	Qubits
1024	8	18	6	3	6	28	640	2.87%	9.4	1.1e+09	742
1536	8	21	6	3	5	31	960	1.83%	9.3	3.1e+09	1074
2048	8	21	6	3	5	33	1280	1.25%	9.2	6.5e+09	1399
3072	8	21	6	3	5	35	1920	0.91%	9.2	1.9e+10	2043
4096	8	24	6	3	5	36	2560	0.80%	9.2	4.0e+10	2692
6144	8	24	6	3	5	39	3840	0.42%	9.1	1.2e+11	3978
8192	8	24	6	3	5	40	5120	0.40%	9.1	2.7e+11	5261

Table 5: Highlighted parameter choices, and resulting logical cost estimates, for factoring RSA integers of various sizes. P_{deviant} is the shot failure rate due to using an approximate modular exponentiation with masking. E(shots) is the expected number of shots, equal to $(s+1)/(1-P_{\text{deviant}})/0.99$ (the $/0.99$ accounts for the chance of postprocessing failure [Eke20]). The Toffolis column is expected Toffolis per factoring (not per shot).

3.2 Physical Costs

The imagined physical layout of the algorithm is as follows. There will be three regions: compute, hot storage, and cold storage. The hot storage region will store qubits "normally", as distance d surface code patches using $2(d+1)^2$ physical qubits per logical qubit. The cold storage region will store qubits more densely, by using yoked surface codes [Gid+25]. The compute region will have room for performing lattice surgery and use magic state cultivation [GSJ24] followed by 8T-to-CCZ distillation [Jon13] to power Toffoli gates.

I will show later in this subsection that, when factoring a 2048 bit integer, each shot will take roughly 12 hours and involve fewer than 1600 logical qubits (including idle hot patches). Given the assumed surface code cycle time of 1 microsecond, this implies $1600 \cdot 12 \cdot 60 \cdot 60 \cdot 10^6 \approx 6.9 \cdot 10^{13}$ logical qubit rounds of runtime to protect. Choosing a target logical error rate of 10^{-15} per logical qubit round will thus result in a no-logical-error shot rate of $(1 - 10^{-15})^{6.9 \cdot 10^{13}} \approx 93.3\%$.

Referring to Figure 6, note that a distance of 25 is sufficient for normal surface code patches to reach a per-patch per-round logical error rate of 10^{-15}. So my hot patches will use $2 \cdot (25+1)^2 = 1352$ physical qubits per logical qubit. For the cold storage, again referring to Figure 6, yoking with a 2D parity check code reaches a logical error rate of 10^{-15} when using 430 physical qubits per logical qubit. So cold logical qubits will be roughly triple the density of hot logical qubits.

The $n = 2048$ row of Table 5 specifies $s = 8$, $\ell = 21$, $w_1 = 6$, $w_3 = 3$, $w_4 = 5$, $f = 33$, and $m = 1280$. Plugging these values into Table 4, the maximum logical qubit count occurs during loop4 when there are $m + 3f + 2\ell + \text{len } m = 1409$ logical qubits active. The $m = 1280$ input logical qubits are in cold storage, and so cover $1280 \cdot 430 = 550400$ physical qubits. The remaining $3f + 2\ell + \text{len } m = 131$ logical qubits are in hot storage, and so cover $131 \cdot 1352 = 177112$ physical qubits. Finally, the compute region will use a 7x18 region of hot patches (170352 physical qubits). This is enough for six magic state factories, each covering a 3×4 area of hot patches, as well as three

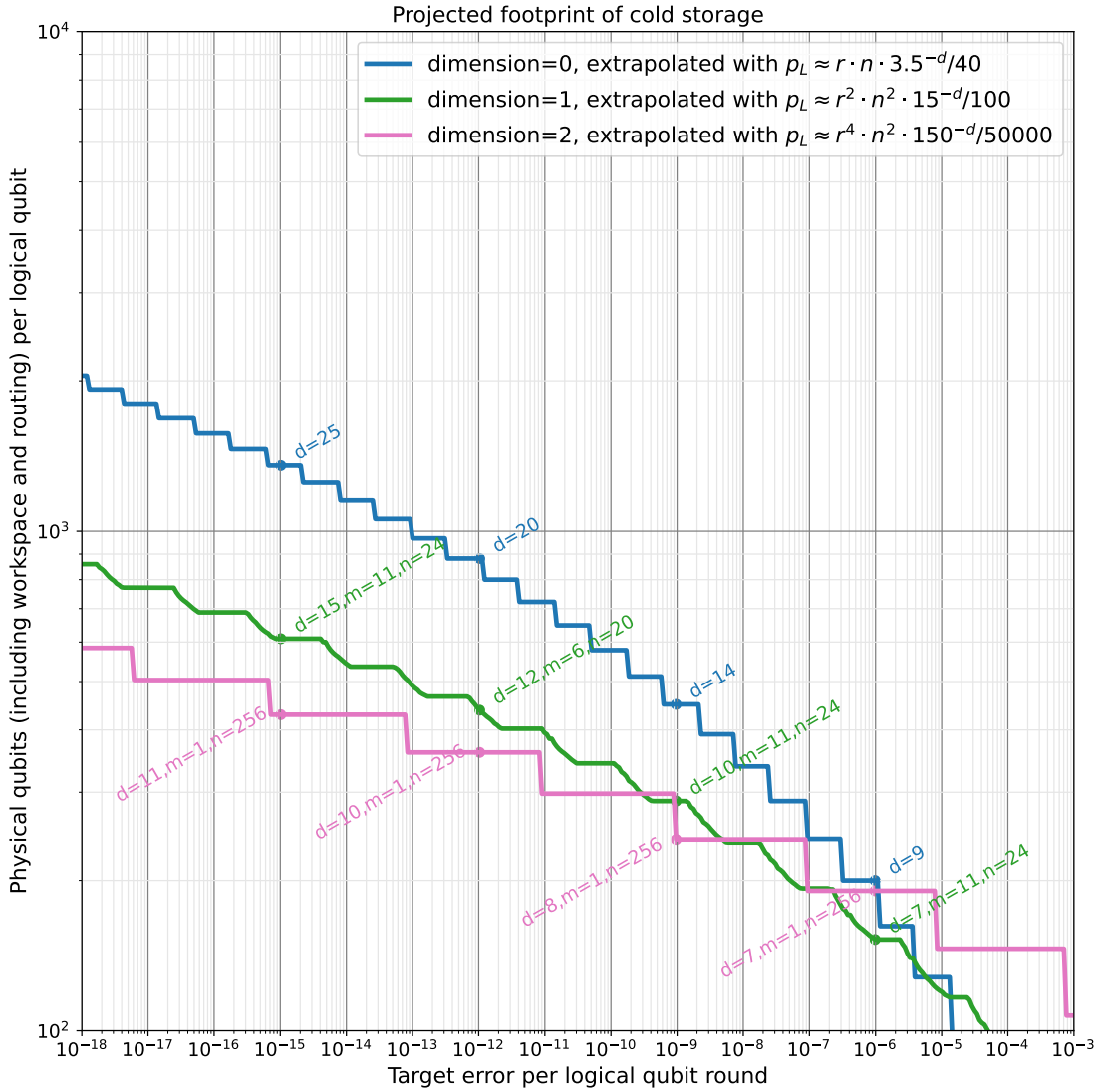

Figure 6: Projections of the cost of storage of normal surface codes and surface codes "yoked by" (concatenated below) 1D and 2D parity check codes, under uniform depolarizing noise. The blue line is storage using normal surface codes ("hot storage" in this paper). The pink line is storage using surface codes yoked by a two dimensional parity check code ("cold storage" in this paper). The d, m, n labels indicate the surface code patch diameter (d), the number of used surface code patches per yoked code instance (n), and the number yoked code instances sharing lattice surgery workspace (m). This is a variant of supplementary figure 2 of [Gid+25], with fit parameters rounded to one-and-a-half sig figs instead of one sig fig. I'm using this variation because I was worried that the projected costs with one sig fig were too optimistic, due to rounding up the fit constants. Specifically: in this figure the dimension=0 suppression gain parameter is 3.5 instead of 4, and the dimension=2 suppression gain parameter is 150 instead of 200, making the curves more pessimistic.

columns of workspace. The factories (from figure 24 of [GSJ24]) are notably smaller than the 15×8 factories used in [GE21], due to replacing the first stage of distillation with magic state cultivation. The total number of logical qubits, including idle hot patches, is $1280 + 131 + 7 \cdot 18 = 1537 < 1600$, as promised when picking the code distances. The total number of physical qubits is 897864.

For slack, I report the physical qubit count as one million instead of 900 thousand. Specifically, I'm a bit worried that the cold storage might get slightly worse during loop1 of the algorithm. During this loop, the yoke measurements would have to contend with Z-splitting copies [BH17] of cold logical qubits to stream to hot storage to be used as the addresses of lookups. I'm confident 100K physical qubits is sufficient slack to ensure the streaming can be interleaved with the required yoke checks. I leave detailed analysis and simulations of cold storage, under various workloads, as future work.

A mock-up of the spatial layout (after some rounding) is shown in Figure 7.

Let's now turn to runtime. According to figure 1 of [GSJ24], cultivating a T state with a logical error rate of 10^{-7} uses 30000 physical qubit $\cdot$ rounds of spacetime volume. Each 8T-to-CCZ factory covers $3 \cdot 4 \cdot 26^2 \cdot 2$ qubits, and needs 8 T states, suggesting an average cultivation time of 14.7 rounds. The factory itself (shown in figure 24 of [GSJ24]) has 6 layers of lattice surgery. It uses temporally encoded lattice surgery [CC22] so each layer should be able to execute in $2/3d$ rounds rather than d rounds. In total this amounts to 114.7 rounds per CCZ state, which I round up to 150 rounds for slack. Since the magic state cultivation targets a logical error rate of 10^{-7} and 8T-to-CCZ distillation has an error suppression of $28p^2$ [Jon13], the distilled CCZ states will have a logical error rate of $28 \cdot (10^{-7})^2 < 10^{-12}$. This is low enough (compared to the Toffoli count of 6.5 billion and other error rates in the paper) that we can ignore it. Note that, because there are six CCZ factories, the lattice surgery period (25 microseconds because $d = 25$) happens to be equal to the CCZ state period (25 microseconds because $150/6 = 25$). These two numbers (combined with Table 3) allow me to bound how long additions, lookups, and phaseups take.

The largest additions that occur are the $f = 33$ qubit additions into the output register during loop4. These additions require 32 CCZ states. It will take at least $32 \cdot 25 = 800$ microseconds to generate and consume these states. After these states are consumed, the adder has finished computing its output but it still needs to uncompute one of its inputs. This is strictly simpler, not requiring any CCZ states, so the whole adder will take at most $800 \cdot 2 = 1600$ microseconds. I round this up to 2 milliseconds to leave slack for things like rearranging qubits at the beginning and end of the operation. See Appendix A.2 for a more detailed mock-up of the addition operation.

The largest lookups that occur are the $w_1 = 6$ address qubit lookups during loop1. These lookups need at least $2^6 - 6 - 1 = 57$ CCZ states, and at least 63 layers of lattice surgery [Bab+18]. This will take at least $63 \cdot 25 = 1575$ microseconds. I again round this up to 2 milliseconds for slack. See Appendix A.4 for a more detailed mock-up of the lookup operation.

The largest phaseup operations also use 6 qubit addresses. This requires 8 CCZ states (instead of the 64 used by a 6 qubit lookup), and the amount of lattice surgery required is similarly more than twice as small as what's needed for a lookup. So we can safely conclude phaseup operations will take less than half as long as lookups; at most 1 millisecond. See Appendix A.3 for a more detailed mock-up of the phaseup operation.

With operation durations in hand, we can plug parameter values into Table 3 and then add up the iterations times the operations times the durations. Using 2 milliseconds per addition, 2 milliseconds per lookup, and 1 millisecond per phaseup produces an estimate of 12.07 hours per shot (as promised when picking the code distance). The $n = 2048$ row of Table 5 indicates the expected number of shots is 9.2, giving an expected total time of $12.07 \cdot 9.1/24 = 4.63$ days. However, this doesn't yet account for shots lost due to logical errors. Recall from earlier that shots have a 93.3% chance of not having a logical error. Dividing 4.63 days by 93.3% gives the actual time estimate per factoring: 4.96 days. Which I round up to a week for slack.

So, summarizing, I estimate that factoring a 2048 bit RSA integer requires less than a million noisy qubits and will finish in less than a week. This estimate assumes a quantum computer with a surface code cycle time of 1 microsecond, a control system reaction time of 10 microseconds, a square grid of qubits with nearest neighbor connectivity, and a uniform depolarizing noise model with a noise strength of 1 error per 1000 gates. See Figure 1 for a comparison of this estimate to previous cost estimates.

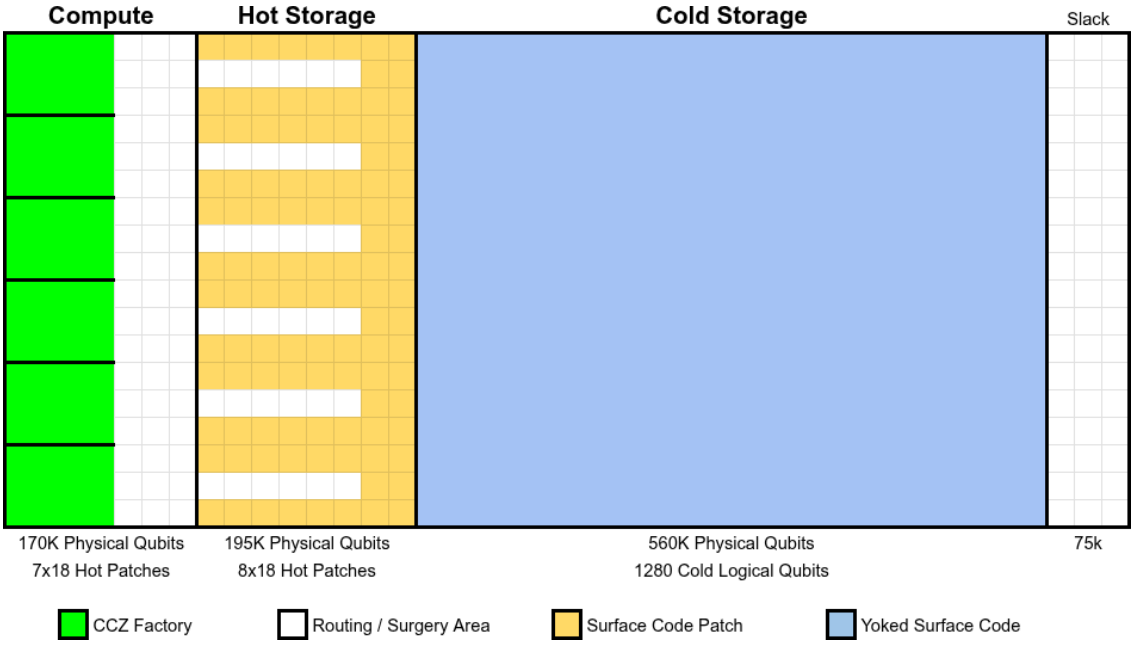

Figure 7: Mock-up of a physical layout for $n = 2048, s = 8$. On the left: a narrow "compute" region containing six magic state factories and some workspace for lattice surgery. On the right: a large "cold storage" region for high-density storage of idle qubits using yoked surface codes [Gid+25] In the middle: "hot storage" for active qubits that need to rapidly interact with the compute region. Some slack space is included in case of unforeseen inefficiencies.

4 Conclusion

In this paper, I reduced the expected number of qubits needed to break RSA2048 from 20 million to 1 million. I did this by combining and streamlining results from [CFS24], [Gid+25], and [GSJ24]. My hope is that this provides a sign post for the current state of the art in quantum factoring, and informs how quickly quantum-safe cryptosystems should be deployed.

Without changing the physical assumptions made by this paper, I see no way reduce the qubit count by another order of magnitude. I cannot plausibly claim that a 2048 bit RSA integer could be factored with a hundred thousand noisy qubits. But there's a saying in cryptography: "attacks always get better" [Sch09]. Over the past decade, that has held true for quantum factoring. Looking forward, I agree with the initial public draft of the NIST internal report on the transition to post-quantum cryptography standards [Moo+24]: vulnerable systems should be deprecated after 2030 and disallowed after 2035. Not because I expect sufficiently large quantum computers to exist by 2030, but because I prefer security to not be contingent on progress being slow.

5 Acknowledgments

I thank Greg Kahanamoku-Meyer, Thiago Bergamaschi, Dave Bacon, Cody Jones, James Manyika, and Matt McEwen for helpful discussions. I thank Michael Newman for discussions and for revisiting [Gid+25] to produce the variant of one its figures shown in Figure 6. I thank Adam Zalcman for discussions and for contributing code to speed up finding prime sets with low modular deviation. I thank Martin Ekerå for helpful discussions, as well as for helpful corrections, and I look forward to Martin extending the results to other cryptographic cases and becoming a co-author in a future version of this paper. I thank Hartmut Neven, and the Google Quantum AI team as a whole, for creating an environment where this work was possible.

References

[Bab+18] Ryan Babbush, Craig Gidney, Dominic W. Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. "Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity". In: *Physical Review X* 8.4 (Oct. 2018). ISSN: 2160-3308. DOI: 10.1103/physrevx.8.041015. URL: http://dx.doi.org/10.1103/PhysRevX.8.041015 (Cited on pages 16, 20).

[Bea03] S. Beauregard. "Circuit for Shor's algorithm using 2n+3 qubits". In: *Quantum Information and Computation* 3.2 (Mar. 2003), pp. 175–185. ISSN: 1533-7146. DOI: 10.26421/qic3.2-8. URL: http://dx.doi.org/10.26421/QIC3.2-8 (Cited on page 3).

[Bec+96] David Beckman, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. "Efficient networks for quantum factoring". In: *Physical Review A* 54.2 (Aug. 1996), pp. 1034–1063. ISSN: 1094-1622. DOI: 10.1103/physreva.54.1034. URL: http://dx.doi.org/10.1103/PhysRevA.54.1034 (Cited on page 3).

[Ber+19] Dominic W Berry, Craig Gidney, Mario Motta, Jarrod R McClean, and Ryan Babbush. "Qubitization of Arbitrary Basis Quantum Chemistry by Low Rank Factorization". In: *arXiv preprint arXiv:1902.02134* (2019) (Cited on pages 16, 34).

[Ber+24] Dominic W. Berry, Yu Tong, Tanuj Khattar, Alec White, Tae In Kim, Sergio Boixo, Lin Lin, Seunghoon Lee, Garnet Kin-Lic Chan, Ryan Babbush, and Nicholas C. Rubin. *Rapid initial state preparation for the quantum simulation of strongly correlated molecules*. 2024. DOI: 10.48550/ARXIV.2409.11748. URL: https://arxiv.org/abs/2409.11748 (Cited on page 16).

[BH17] Niel de Beaudrap and Dominic Horsman. "The ZX calculus is a language for surface code lattice surgery". In: *arXiv preprint arXiv:1704.08670* (2017) (Cited on pages 20, 31).

[CC22] Christopher Chamberland and Earl T. Campbell. "Universal Quantum Computing with Twist-Free and Temporally Encoded Lattice Surgery". In: *PRX Quantum* 3.1 (Feb. 2022). ISSN: 2691-3399. DOI: 10.1103/prxquantum.3.010331. URL: http://dx.doi.org/10.1103/PRXQuantum.3.010331 (Cited on page 20).

[CFS24] Clémence Chevignard, Pierre-Alain Fouque, and André Schrottenloher. *Reducing the Number of Qubits in Quantum Factoring*. Cryptology ePrint Archive, Paper 2024/222. 2024. URL: https://eprint.iacr.org/2024/222 (Cited on pages 2, 3, 14, 15, 21).

[Cop02] D. Coppersmith. *An approximate Fourier transform useful in quantum factoring*. 2002. DOI: 10.48550/ARXIV.QUANT-PH/0201067. URL: https://arxiv.org/abs/quant-ph/0201067 (Cited on page 10).

[CW] R. Cleve and J. Watrous. "Fast parallel circuits for the quantum Fourier transform". In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. SFCS-00. IEEE Comput. Soc, pp. 526–536. DOI: 10.1109/sfcs.2000.892140. URL: http://dx.doi.org/10.1109/SFCS.2000.892140 (Cited on page 3).

[EH17] Martin Ekerå and Johan Håstad. "Quantum algorithms for computing short discrete logarithms and factoring RSA integers". In: *Post-Quantum Cryptography: 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings 8*. Springer. 2017, pp. 347–363 (Cited on pages 3, 13).

[Eke16] Martin Ekerå. *Modifying Shor's algorithm to compute short discrete logarithms*. Cryptology ePrint Archive, Report 2016/1128. https://eprint.iacr.org/2016/1128. 2016 (Cited on page 3).

[Eke19] Martin Ekerå. *Revisiting Shor's quantum algorithm for computing general discrete logarithms*. 2019. DOI: 10.48550/ARXIV.1905.09084. URL: https://arxiv.org/abs/1905.09084 (Cited on page 3).

[Eke20] Martin Ekerå. "On post-processing in the quantum algorithm for computing short discrete logarithms". In: *Designs, Codes and Cryptography* 88.11 (Aug. 2020), pp. 2313–2335. ISSN: 1573-7586. DOI: 10.1007/s10623-020-00783-2. URL: http://dx.doi.org/10.1007/s10623-020-00783-2 (Cited on pages 14, 18).

[Eke21] Martin Ekerå. “Quantum algorithms for computing general discrete logarithms and orders with tradeoffs”. In: *Journal of Mathematical Cryptology* 15.1 (Jan. 2021), pp. 359–407. ISSN: 1862-2984. DOI: 10.1515/jmc-2020-0006. URL: http://dx.doi.org/10.1515/jmc-2020-0006 (Cited on page 3).

[FG18] Austin G Fowler and Craig Gidney. “Low overhead quantum computation using lattice surgery”. In: *arXiv preprint arXiv:1808.06709* (2018). DOI: 10.48550/arXiv.1808.06709 (Cited on pages 30, 34).

[Fow+12] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland. “Surface codes: Towards practical large-scale quantum computation”. In: *Phys. Rev. A* 86 (2012). arXiv:1208.0928, p. 032324. DOI: 10.1103/PhysRevA.86.032324 (Cited on pages 2, 3).

[GE21] Craig Gidney and Martin Ekerå. “How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits”. In: *Quantum* 5 (2021), p. 433. DOI: 10.22331/q-2021-04-15-433 (Cited on pages 2, 3, 20).

[GF19] Craig Gidney and Austin G. Fowler. *Flexible layout of surface code computations using AutoCCZ states*. 2019. DOI: 10.48550/ARXIV.1905.08916. URL: https://arxiv.org/abs/1905.08916 (Cited on pages 30, 31, 39).

[Gid+25] Craig Gidney, Michael Newman, Peter Brooks, and Cody Jones. “Yoked surface codes”. In: *Nature Communications* (May 2025). ISSN: 2041-1723. DOI: 10.1038/s41467-025-59714-1. URL: https://doi.org/10.1038/s41467-025-59714-1 (Cited on pages 18, 19, 21).

[Gid16] Craig Gidney. *Turning Gradients into Additions into QFTs*. https://algassert.com/post/1620. Accessed: 2025-01-01. July 2016 (Cited on page 38).

[Gid17] Craig Gidney. “Factoring with n+ 2 clean qubits and n-1 dirty qubits”. In: *arXiv preprint arXiv:1706.07884* (2017) (Cited on page 3).

[Gid18] Craig Gidney. “Halving the cost of quantum addition”. In: *Quantum* 2 (2018), p. 74 (Cited on pages 16, 38, 39).

[Gid19a] Craig Gidney. *Approximate encoded permutations and piecewise quantum adders*. 2019. DOI: 10.48550/ARXIV.1905.08488. URL: https://arxiv.org/abs/1905.08488 (Cited on page 7).

[Gid19b] Craig Gidney. *Windowed quantum arithmetic*. 2019. DOI: 10.48550/ARXIV.1905.07682. URL: https://arxiv.org/abs/1905.07682 (Cited on page 15).

[Gid25] Craig Gidney. “Data for "How to factor 2048 bit RSA integers with a million noisy qubits"”. In: *Zenodo* (May 2025). DOI: 10.5281/zenodo.15347487 (Cited on pages 1, 26).

[GM19] Vlad Gheorghiu and Michele Mosca. *Benchmarking the quantum cryptanalysis of symmetric, public-key and hash-based cryptographic schemes*. 2019. DOI: 10.48550/ARXIV.1902.02332. URL: https://arxiv.org/abs/1902.02332 (Cited on pages 2, 3).

[GSJ24] Craig Gidney, Noah Shutty, and Cody Jones. *Magic state cultivation: growing T states as cheap as CNOT gates*. 2024. DOI: 10.48550/ARXIV.2409.17595. URL: https://arxiv.org/abs/2409.17595 (Cited on pages 18, 20, 21).

[Hor+12] Dominic Horsman, Austin G Fowler, Simon Devitt, and Rodney Van Meter. “Surface code quantum computing by lattice surgery”. In: *New Journal of Physics* 14.12 (2012), p. 123011. DOI: 10.1088/1367-2630/14/12/123011 (Cited on page 26).

[HRS16] Thomas Häner, Martin Roetteler, and Krysta M Svore. “Factoring using 2n+ 2 qubits with Toffoli based modular multiplication”. In: *arXiv preprint arXiv:1611.07995* (2016) (Cited on page 3).

[JG20] Samuel Jaques and Craig Gidney. *Offloading Quantum Computation by Superposition Masking*. 2020. DOI: 10.48550/ARXIV.2008.04577. URL: https://arxiv.org/abs/2008.04577 (Cited on page 7).

[Jon+12] N. Cody Jones, Rodney Van Meter, Austin G. Fowler, Peter L. McMahon, Jungsang Kim, Thaddeus D. Ladd, and Yoshihisa Yamamoto. "Layered Architecture for Quantum Computing". In: *Physical Review X* 2.3 (July 2012). ISSN: 2160-3308. DOI: 10.1103/physrevx.2.031007. URL: http://dx.doi.org/10.1103/PhysRevX.2.031007 (Cited on pages 2, 3).

[Jon13] Cody Jones. "Low-overhead constructions for the fault-tolerant Toffoli gate". In: *Physical Review A* 87.2 (2013), p. 022328 (Cited on pages 18, 20, 30).

[Kni95] E Knill. *On Shor's quantum factor finding algorithm: Increasing the probability of success and tradeoffs involving the Fourier Transform modulus*. Tech. rep. Technical Report LAUR-95-3350, Los Alamos Laboratory, 1995 (Cited on page 3).

[KSV02] A. Kitaev, A. Shen, and M. Vyalyi. *Classical and Quantum Computation*. American Mathematical Society, May 2002. ISBN: 9781470418007. DOI: 10.1090/gsm/047. URL: http://dx.doi.org/10.1090/gsm/047 (Cited on pages 38, 39).

[Kut06] Samuel A. Kutin. *Shor's algorithm on a nearest-neighbor machine*. 2006. DOI: 10.48550/ARXIV.QUANT-PH/0609001. URL: https://arxiv.org/abs/quant-ph/0609001 (Cited on page 15).

[Lit18] Daniel Litinski. "A game of surface codes: Large-scale quantum computing with lattice surgery". In: *arXiv preprint arXiv:1808.02892* (2018) (Cited on pages 30, 34, 38, 39).

[LKS24] Guang Hao Low, Vadym Kliuchnikov, and Luke Schaeffer. "Trading T gates for dirty qubits in state preparation and unitary synthesis". In: *Quantum* 8 (June 2024), p. 1375. ISSN: 2521-327X. DOI: 10.22331/q-2024-06-17-1375. URL: http://dx.doi.org/10.22331/q-2024-06-17-1375 (Cited on page 35).

[LN22] Daniel Litinski and Naomi Nickerson. *Active volume: An architecture for efficient fault-tolerant quantum computers with limited non-local connections*. 2022. DOI: 10.48550/ARXIV.2211.15465. URL: https://arxiv.org/abs/2211.15465 (Cited on pages 2, 3).

[LNS25] Alessandro Luongo, Varun Narasimhachar, and Adithya Sireesh. *Optimized circuits for windowed modular arithmetic with applications to quantum attacks against RSA*. 2025. DOI: 10.48550/ARXIV.2502.17325. URL: https://arxiv.org/abs/2502.17325 (Cited on page 15).

[ME99] Michele Mosca and Artur Ekert. "The Hidden Subgroup Problem and Eigenvalue Estimation on a Quantum Computer". In: *Quantum Computing and Quantum Communications*. Springer Berlin Heidelberg, 1999, pp. 174–188. ISBN: 9783540492085. DOI: 10.1007/3-540-49208-9_15. URL: http://dx.doi.org/10.1007/3-540-49208-9_15 (Cited on pages 3, 38, 39).

[Moo+24] Dustin Moody, Ray Perlner, Andrew Regenscheid, Angela Robinson, and David Cooper. *NIST IR 8547 ipd: Transition to Post-Quantum Cryptography Standards*. NIST, 2024. DOI: 10.6028/nist.ir.8547.ipd. URL: http://dx.doi.org/10.6028/NIST.IR.8547.ipd (Cited on page 21).

[MP01] Alan Mishchenko and Marek Perkowski. "Fast heuristic minimization of exclusive-sums-of-products". In: (2001). URL: http://archives.pdx.edu/ds/psu/12886 (Cited on page 34).

[MS22] Alexander May and Lars Schlieper. "Quantum Period Finding is Compression Robust". In: *IACR Transactions on Symmetric Cryptology* (Mar. 2022), pp. 183–211. ISSN: 2519-173X. DOI: 10.46586/tosc.v2022.i1.183-211. URL: http://dx.doi.org/10.46586/tosc.v2022.i1.183-211 (Cited on pages 3, 7, 10).

[NSM20] Yunseong Nam, Yuan Su, and Dmitri Maslov. "Approximate quantum Fourier transform with O(n log(n)) T gates". In: *npj Quantum Information* 6.1 (Mar. 2020). ISSN: 2056-6387. DOI: 10.1038/s41534-020-0257-5. URL: http://dx.doi.org/10.1038/s41534-020-0257-5 (Cited on pages 38, 39).

[OC17] Joe O'Gorman and Earl T. Campbell. "Quantum computation with realistic magic-state factories". In: *Physical Review A* 95.3 (Mar. 2017). ISSN: 2469-9934. DOI: 10.1103/physreva.95.032338. URL: http://dx.doi.org/10.1103/PhysRevA.95.032338 (Cited on pages 2, 3).

[PG14] Archimedes Pavlidis and Dimitris Gizopoulos. “Fast quantum modular exponentiation architecture for Shor’s factoring alogrithm”. In: *Quantum Information and Computation* 14.7 & 8 (May 2014), pp. 649–682. ISSN: 1533-7146. DOI: 10.26421/qic14.7-8-8. URL: http://dx.doi.org/10.26421/QIC14.7-8-8 (Cited on page 3).

[PP00] S. Parker and M. B. Plenio. “Efficient Factorization with a Single Pure Qubit and N Mixed Qubits”. In: *Physical Review Letters* 85.14 (Oct. 2000), pp. 3049–3052. ISSN: 1079-7114. DOI: 10.1103/physrevlett.85.3049. URL: http://dx.doi.org/10.1103/PhysRevLett.85.3049 (Cited on pages 3, 38, 39).

[Reg24] Oded Regev. “An Efficient Quantum Factoring Algorithm”. In: *Journal of the ACM* (Dec. 2024). ISSN: 1557-735X. DOI: 10.1145/3708471. URL: http://dx.doi.org/10.1145/3708471 (Cited on page 3).

[RS14] Neil J. Ross and Peter Selinger. “Optimal ancilla-free Clifford+T approximation of z-rotations”. In: (2014). DOI: 10.48550/ARXIV.1403.2975. URL: https://arxiv.org/abs/1403.2975 (Cited on page 38).

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 1557-7317. DOI: 10.1145/359340.359342. URL: http://dx.doi.org/10.1145/359340.359342 (Cited on page 3).

[Sch09] Bruce Schneier. *New Attack on AES*. https://www.schneier.com/blog/archives/2009/07/new_attack_on_a.html. Accessed: 2025-02-14. 2009 (Cited on page 21).

[Sho94] Peter W Shor. “Algorithms for quantum computation: Discrete logarithms and factoring”. In: *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*. Ieee. 1994, pp. 124–134 (Cited on pages 3, 13).

[Van+10] Rodney Van Meter, Thaddeus D Ladd, Austin G Fowler, and Yoshihisa Yamamoto. “Distributed quantum computation architecture using semiconductor nanophotonics”. In: *International Journal of Quantum Information* 8.01n02 (2010), pp. 295–323. DOI: 10.1142/s0219749910006435 (Cited on pages 2, 3).

[VBE96] Vlatko Vedral, Adriano Barenco, and Artur Ekert. “Quantum networks for elementary arithmetic operations”. In: *Physical Review A* 54.1 (July 1996), pp. 147–153. ISSN: 1094-1622. DOI: 10.1103/physreva.54.147. URL: http://dx.doi.org/10.1103/PhysRevA.54.147 (Cited on page 3).

[VI05] Rodney Van Meter and Kohei M. Itoh. “Fast quantum modular exponentiation”. In: *Physical Review A* 71.5 (May 2005). ISSN: 1094-1622. DOI: 10.1103/physreva.71.052320. URL: http://dx.doi.org/10.1103/PhysRevA.71.052320 (Cited on page 15).

[Whi+09] Mark G. Whitney, Nemanja Isailovic, Yatish Patel, and John Kubiatowicz. “A fault tolerant, area efficient architecture for Shor’s factoring algorithm”. In: *Proceedings of the 36th annual international symposium on Computer architecture*. ISCA ’09. ACM, June 2009, pp. 383–394. DOI: 10.1145/1555754.1555802. URL: http://dx.doi.org/10.1145/1555754.1555802 (Cited on page 3).

[Wik24] Wikipedia. *RSA numbers - RSA-2048*. Accessed: 2024-12-19. 2024. URL: https://en.wikipedia.org/wiki/RSA_numbers#RSA-2048 (Cited on page 6).

[Zal06] Christof Zalka. “Shor’s algorithm with fewer (pure) qubits”. In: *arXiv preprint quant-ph/0601097* (2006) (Cited on pages 3, 7).

[Zal98] Christof Zalka. “Fast versions of Shor’s quantum factoring algorithm”. In: *arXiv preprint quant-ph/9806084* (1998) (Cited on page 3).

A Detailed Mock-ups

A.1 Reference Python Implementation of Algorithm

In this section, I provide tested reference Python code for the optimized algorithm. Understanding the code requires knowing its conventions. In the example code, all quantum values are stored as variables of type `quint`. These variables are prefixed with `Q_` and highlighted red. A quint is a superposed unsigned integer with a known length in qubits (accessed via python's `len` function). Quints can be sliced as if they were Python lists, which returns a view of a subsection of the quint. For example, if `Q_example` is a quint then `Q_example[1:5]` is a view of its second, third, fourth, and fifth qubits (the view's integer value is equal to `Q_example // 2 % 16`). Expressions like `(Q_a << len(Q_b)) | Q_b` are also just constructing views, by concatenating qubits, not performing actual work on the quantum computer.

Quints are created by allocation calls like `qpu.alloc_quint` or `qpu.alloc_phase_gradient`, and cleared by "del" calls like `qpu.del_by_equal_to`. Measurement based uncomputations begin with a call to `qpu.mx_rz` or `qpu.del_measure_x`. For each qubit in the quint, these methods measure the qubit in the X basis then either clear the qubit to $|0\rangle$ or deallocate it. The measurements determine if parts of the superposition where the corresponding qubit was ON had their amplitude negated by phase kickback, and are bit packed into an `int` returned by the method.

The simulation code is expected to uncompute all registers and kickback phases before finishing, which is verified by calling `qpu.verify_clean_finish()` (not shown in the example code). Under the hood, the simulator works by tracking the value and phase of a few randomly sampled classical trajectories. This isn't sufficient to verify interference effects, or to verify that information wasn't incorrectly revealed (e.g. it can't verify that masking was done correctly), but it fuzzes that the classical output is correct and that phase kickback from measurement based uncomputation is being fixed.

The key quantum operations that appear in the example code are additions, subtractions, lookups, and phase flip comparisons. An addition looks like `Q_a += b`, which offsets `Q_a` by `b` (working modulo `2**len(Q_a)`). A subtraction looks like `Q_a -= b`. A phase flip comparison looks like `qpu.z(Q_a < b)`, where the `qpu.z` is shorthand for "phase flip the parts of the superposition where". Additions, subtractions, and comparisons all have circuits that are minor variations on an addition circuit and so, in cost estimates, they are all treated as additions.

Lookups appear as parts of other expressions. For example, `Q_a += T[Q_b]` implies that a lookup of the table `T` addressed by `Q_b` will be performed in order to produce the offset value that will be added into `Q_a`. The lookup is uncomputed by the end of the addition using measurement based uncomputation, producing phase flip corrections to do. These phase flip corrections go into the "vent" of the lookup, which will be specified earlier in the code with a line like `T = T.venting_into(V)`. Later in the code, a call like `qpu.z(V[Q_b])` will appear, which is the phaseup used to resolve the accumulated phase corrections from the lookups. Alternatively, a call like `qpu.push_uncompute_info(V)` may appear. This call will be matched by a `qpu.pop_uncompute_info()` call in a later subroutine, which will handle the phase corrections. This is done because often an address is used for multiple lookups, for example during a computation subroutine and again during an uncomputation subroutine, and the phase corrections of such lookups can be merged.

A special kind of lookup that appears is GHZ lookups like `Q_a += Q_b.ghz_lookup(k)`. This is a lookup with a single address qubit and a lookup table of the form `[0, k]` for some integer k. These lookups can be implemented very efficiently in lattice surgery, by Z-splitting [Hor+12] the single address qubit into each non-zero value of k. At the end of the lookup the qubits can be Z-merged by measuring all but one of them in the X basis, and applying a corrective Z gate to the remaining qubit if an odd number of the measurements returned `True`. So a GHZ lookup is more akin to a layer of lattice surgery than to a full blown table lookup, and correspondingly I don't count GHZ lookups as lookups in cost estimates.

Much of the code refers to an instance of `ExecutionConfig`, which includes information about the exponentiation that is being performed as well as resources like the precomputed tables of numbers used by the various subroutines. Readers who want to see the details of this class or run the code should refer to the Zenodo upload [Gid25].

```
import numpy as np

from facto.algorithm.prep import ExecutionConfig
from scatter_script import QPU, quint, Lookup


def approx_modexp(
        Q_exponent: quint,
        conf: ExecutionConfig,
        qpu: QPU,
) -> quint:
    """Quantum computes an approximate modular exponentation.

    Args:
        Q_exponent: The superposed exponent to exponentiate by.
        conf: Problem configuration data, precomputed tables, etc.
        qpu: Simulator instance being acted upon.

    Returns:
        The result of the approximate modular exponentation.
    """

    Q_dlog: quint = qpu.alloc_quint(length=conf.len_dlog_accumulator)
    Q_result: quint = qpu.alloc_quint(
        length=conf.len_accumulator + 1,
        scatter=True,  # Initialized to a superposition mask.
        scatter_range=1 << conf.mask_bits,
    )
    loop1_vent = Lookup(np.zeros((conf.num_windows1, 1 << conf.window1), dtype=np.bool_))

    for i in range(len(conf.periods)):
        p = int(conf.periods[i])

        # Offset Q_dlog to equal dlog(generators[i], residue, p)
        loop1(
            Q_dlog=Q_dlog,
            Q_exponent=Q_exponent,
            vent=loop1_vent,
            i=i,
            conf=conf,
        )

        # Compress Q_dlog while preserving Q_dlog % (p - 1)
        Q_compressed = loop2(
            Q_target=Q_dlog,
            modulus=p - 1,
            compressed_len=p.bit_length(),
        )

        # Perform: let Q_residue := pow(generators[i], Q_dlog, p)
        Q_residue = loop3(
            Q_dlog=Q_compressed,
            i=i,
            conf=conf,
            qpu=qpu,
        )

        # Approximate: Q_result += Q_residue * (L//p) * pow(L//p, -1, p)
        loop4(
            Q_residue=Q_residue,
            Q_acc=Q_result,
            i=i,
            conf=conf,
            qpu=qpu,
        )

        # Perform: del Q_residue := pow(generators[i], Q_dlog, p)
        unloop3(
            Q_unresult=Q_residue,
            Q_dlog=Q_compressed,
            i=i,
            conf=conf,
            qpu=qpu,
        )

        # Uncompress Q_dlog
        unloop2(
            Q_target=Q_dlog,
            modulus=p - 1,
            compressed_len=p.bit_length(),
        )

    # Uncompute Q_dlog.
    loop1(
        Q_dlog=Q_dlog,
        Q_exponent=Q_exponent,
        vent=loop1_vent,
        i=len(conf.periods),
        conf=conf,
    )
    Q_dlog.del_by_equal_to(0)

    # Clear accumulated phase corrections from Q_dlog-related lookups.
    for j in range(conf.num_windows1):
        Q_k = Q_exponent[j * conf.window1 :][: conf.window1]
```

```
            qpu.z(loop1_vent[j, Q_k])

    return Q_result


def loop1(
        Q_dlog: quint,
        conf: ExecutionConfig,
        Q_exponent: quint,
        i: int,
        vent: Lookup,
) -> None:
    """Offsets Q_dlog to the discrete log of the next residue.

    Args:
        Q_dlog: The register to offset.
        conf: Specifies details like the base of the exponent for the current
            residue, window sizes, and precomputed tables.
        Q_exponent: In the problem spec this is the value to exponentiate by,
            but in this method it's the value to use as addresses for table
            lookups that get added together.
        i: The iteration variable of the loop iterating over the primes in
            the residue number system. Indexes the modulus and generator,
            as well as related table data, from within `conf`.
        vent: Where to dump deferred phase data for uncomputing lookups.
    """
    for j in range(conf.num_windows1):
        Q_k = Q_exponent[j * conf.window1 :][: conf.window1]
        table = conf.lookup1[i, j].venting_into(vent[j])
        Q_dlog += table[Q_k]


def loop2(
        Q_target: quint,
        modulus: int,
        compressed_len: int,
) -> quint:
    """Compresses `Q_target % modulus` into a smaller part of `Q_target`.

    Args:
        Q_target: The register containing the remainder to compress.
        compressed_len: How large the remainder should be
            after compression. At least modulus.bit_length().
        modulus: The modulus used in the remainder computation.

    Returns:
        The slice of `Q_target` that the remainder is in.
    """
    n = len(Q_target)
    while n > compressed_len:
        n -= 1
        threshold = modulus << (n - compressed_len)

        # Perform a step of binary long division.
        Q_target[: n + 1] -= threshold
        Q_target[:n] += Q_target[n].ghz_lookup(threshold)

    return Q_target[:n]


def loop3(
        conf: ExecutionConfig,
        Q_dlog: quint,
        i: int,
        qpu: QPU,
) -> quint:
    """Computes `pow(conf.generators[i], Q_exponent, conf.periods[i])`.

    Args:
        conf: Specifies details like the base of the exponent for the current
            residue, window sizes, and precomputed tables.
        Q_dlog: Superposed value to exponentiate by.
        i: The iteration variable of the loop iterating over the primes in
            the residue number system. Indexes the modulus and generator,
            as well as related table data, from within `conf`.
        qpu: Simulator instance being acted upon.

    Returns:
        An allocated register containing the result of the exponentation.
    """
    modulus = int(conf.periods[i])
    Q_result = qpu.alloc_quint(length=modulus.bit_length() + 1)
    Q_helper = qpu.alloc_quint(length=modulus.bit_length() + 1)

    # Skip first 2 iterations by direct lookup.
    table = conf.lookup3c[i].venting_into_new_table()
    Q_l = Q_dlog[: conf.window3a * 2]
    Q_result ^= table[Q_l]
    qpu.push_uncompute_info(table.vent)

    # Decompose exponentiation into windowed multiplications.
    for j in range(2, conf.num_windows3a):
        Q_l1 = Q_dlog[j * conf.window3a :][: conf.window3a]

        # Perform `let Q_helper := Q_residue * X % N`
```

```
        for k in range(conf.num_windows3b):
            Q_l0 = Q_result[k * conf.window3b :][: conf.window3b]
            Q_l = (Q_l1 << conf.window3b) | Q_l0
            table = conf.lookup3a[i, j, k].venting_into_new_table()

            # Subtraction mod the modulus.
            Q_helper -= table[Q_l]
            Q_helper[:-1] += Q_helper[-1].ghz_lookup(modulus)

            # Defer phase corrections to `unloop3` method.
            phase_wrap = Q_helper[-1].mx_rz()
            qpu.push_uncompute_info((phase_wrap, table.vent))

        Q_result, Q_helper = Q_helper, Q_result

        # Defer `del Q_helper := Q_residue * X^-1 % N` to `unloop3`
        qpu.push_uncompute_info(Q_helper.mx_rz())

    Q_helper.del_by_equal_to(0)
    return Q_result


def loop4(
        conf: ExecutionConfig,
        Q_residue: quint,
        Q_acc: quint,
        i: int,
        qpu: QPU,
) -> None:
    """Adds Q_residue's approximate contribution into Q_result_accumulator.

    Args:
        conf: Specifies details like the base of the exponent for the current
            residue, window sizes, and precomputed tables.
        Q_residue: The residue of the exact total for the current modulus.
        Q_acc: Where to add the residue's approximate contributions.
        i: The iteration variable of the loop iterating over the primes in
            the residue number system. Indexes the modulus and generator,
            as well as related table data, from within `conf`.
        qpu: Simulator instance being acted upon.
    """
    trunc = conf.truncated_modulus
    for j in range(conf.num_windows4):
        Q_k = Q_residue[j * conf.window4 :][: conf.window4]
        table = conf.lookup4[i, j].venting_into_new_table()
        table2 = trunc - table

        # Subtraction mod the truncated modulus.
        Q_acc -= table[Q_k]
        Q_acc[:-1] += Q_acc[-1].ghz_lookup(trunc)

        if Q_acc[-1].mx_rz():
            # Fix wraparound phase with a comparison.
            qpu.z(Q_acc[:-1] >= table2[Q_k])

        # Fix deferred corrections with a phase lookup.
        qpu.z(table.vent[Q_k])


def unloop3(
        Q_unresult: quint,
        conf: ExecutionConfig,
        Q_dlog: quint,
        i: int,
        qpu: QPU,
) -> None:
    """Uncomputes `pow(conf.generators[i], Q_exponent, conf.periods[i])`.

    Args:
        Q_unresult: The register to uncompute and delete.
        conf: Specifies details like the base of the exponent for the current
            residue, window sizes, and precomputed tables.
        Q_dlog: Superposed value to exponentiate by.
        i: The iteration variable of the loop iterating over the primes in
            the residue number system. Indexes the modulus and generator,
            as well as related table data, from within `conf`.
        qpu: Simulator instance being acted upon.
    """
    modulus = int(conf.periods[i])
    Q_helper = qpu.alloc_quint(length=modulus.bit_length() + 1)

    # Decompose un-exponentiation into windowed un-multiplications.
    for j in range(2, conf.num_windows3a)[::-1]:
        Q_l0 = Q_dlog[j * conf.window3a :][: conf.window3a]

        # Perform `let Q_helper := Q_unresult * X^-1 % N`
        for k in range(conf.num_windows3b)[::-1]:
            Q_l1 = Q_unresult[k * conf.window3b :][: conf.window3b]
            Q_l = (Q_l0 << conf.window3b) | Q_l1
            table1 = conf.lookup3b[i, j, k].venting_into_new_table()
            table2 = modulus - table1

            # Subtraction mod the modulus.
            Q_helper -= table2[Q_l]
            Q_helper[:-1] += Q_helper[-1].ghz_lookup(modulus)
```

```
            # Uncompute wraparound qubit.
            not_phase_wrap = Q_helper[-1].mx_rz()
            qpu.z(not_phase_wrap)  # Only for phase cleared verification.
            if not_phase_wrap:
                # Fix wraparound phase with a comparison.
                qpu.z(Q_helper[:-1] < table1[Q_l])

            # Fix deferred corrections with a phase lookup.
            qpu.z(table1.vent[Q_l])

        phase_mask_from_helper_during_compute = qpu.pop_uncompute_info()
        qpu.cz(Q_helper, phase_mask_from_helper_during_compute)
        Q_unresult, Q_helper = Q_helper, Q_unresult

        # Perform 'del Q_helper := Q_unresult * X % N'
        for k in range(conf.num_windows3b)[::-1]:
            Q_l1 = Q_unresult[k * conf.window3b :][: conf.window3b]
            Q_l = (Q_l0 << conf.window3b) | Q_l1
            phase_wrap, phase_table = qpu.pop_uncompute_info()
            table1 = conf.lookup3a[i, j, k].venting_into(phase_table)
            table2 = modulus - table1

            # Subtraction mod the modulus.
            Q_helper -= table2[Q_l]
            Q_helper[:-1] += Q_helper[-1].ghz_lookup(modulus)

            # Uncompute wraparound qubit.
            not_phase_wrap = Q_helper[-1].mx_rz()
            qpu.z(not_phase_wrap)  # Only for phase cleared verification.
            if phase_wrap ^ not_phase_wrap:
                # Fix wraparound phase with a comparison.
                qpu.z(Q_helper[:-1] < table1[Q_l])

            # Fix deferred corrections with a phase lookup.
            qpu.z(phase_table[Q_l])

    # Skip last 2 iterations by measurement based uncomputation of lookup.
    Q_helper.del_by_equal_to(0)
    mx = Q_unresult.del_measure_x()
    vent = qpu.pop_uncompute_info()
    vent ^= conf.lookup3c[i].phase_corrections_for_mx(mx)
    Q_l = Q_dlog[: conf.window3a * 2]
    qpu.z(vent[Q_l])


def unloop2(
        Q_target: quint,
        modulus: int,
        compressed_len: int,
) -> None:
    """Uncompresses 'Q_target % modulus' from a smaller part of 'Q_target'.

    Args:
        Q_target: The register containing the remainder to uncompress.
        compressed_len: How large the remainder was
            after compression. At least modulus.bit_length().
        modulus: The modulus used in the remainder computation.
    """

    n = compressed_len
    while n < len(Q_target):
        threshold = modulus << (n - compressed_len)

        # Un-perform a step of binary long division.
        Q_target[:n] -= Q_target[n].ghz_lookup(threshold)
        Q_target[: n + 1] += threshold

        n += 1
```

A.2 Addition Operation

Given two inputs a and b, their sum $s = a + b$ satisfies this equation at each bit position k:

$$s_{k+1} = ((a_k \oplus s_k) \cdot (b_k \oplus s_k)) \oplus (a_k \oplus b_k \oplus s_k \oplus a_{k+1} \oplus b_{k+1}) \tag{54}$$

Note that the identity is built out of three Z parity products: $P_1 = Z_{a_k} Z_{s_k}$, $P_2 = Z_{b_k} Z_{s_k}$, and $P_3 = Z_{a_k} Z_{b_k} Z_{s_k} Z_{a_{k+1}} Z_{b_{k+1}}$. This makes the identity interesting, when using lattice surgery, because Z parity products are relatively cheap to access [FG18; Lit18].

A circuit diagram based on Equation 54 is shown in Figure 8. The first half of the diagram is an out-of-place adder. It computes another qubit of s, using qubits from a and b. The second half uncomputes a qubit of b, using the fact that $b = s - a$. Note that the diagram inlines details of a teleportation used to perform a Toffoli gate. Normally, teleported Toffolis are corrected using CX gates and CZ gates [Jon13; GF19]. The circuit instead applies corrections based on teleporting S

gates into multi-qubit Pauli products. Also, the circuit defers corrections that only affect phases until uncomputing $b = a - s$. This allows those corrections to be merged into similar corrections during the uncomputation, balancing out the circuit's usage of delayed choice routing qubits. It uses 3 routing qubits during the computation and also 3 during the uncomputation, instead of 6 during the computation and 2 during the uncomputation [GF19].

Figure 9 shows how Equation 54 can be implemented as a ZX graph and then optimized into an efficient lattice surgery layout. (The diagram doesn't include the delayed choice correction operations shown in Figure 8. They're attached to the CCZ state before it enters the picture.) An important detail here is that the optimized graph has a different "calling convention" from the initial graph. Instead of the qubits of a and b being passed into the adder via input ports, and then returned via output ports, they are attached to the adder via a "Z port". To pass a qubit into a Z port, Z-split the qubit into two qubits [BH17] then Z-merge one of them into the Z port. The remaining qubit stays behind, becoming the output qubit without moving. This calling convention is beneficial because it halves the number of ports.

The physical arrangement of the addition is as follows. The two registers to add are arranged on opposite sides of short hallways leading to the compute region. Conveniently, these hallways have a pitch of 3 surface code patches which matches the pitch of the adder building block and also the pitch of the magic state factories. The qubits within the hallways are ordered so that the least significant qubits are further away from the compute region. As the adder block executes, the hallway is used to access the next two input qubits and then the adder block produces a qubit of the sum, which is stored in the hallway between the two inputs used to produce it. The hallway fills in gradually as the sum is computed.

After the sum is computed, either of the inputs can be uncomputed using measurement based uncomputation as shown on the right hand side of Figure 8. In context of the overall algorithm, usually one of the inputs is the previous value of an accumulator and the other input is the result of a looking up an offset. So the previous accumulator value would be uncomputed as in Figure 8, and then the lookup value would be uncomputed using measurement based uncomputation creating phase corrections to be handled by a later phaseup operation, leaving behind only the new accumulator value. Lattice surgery mock-ups of this process are shown in Figure 10 and Figure 11.

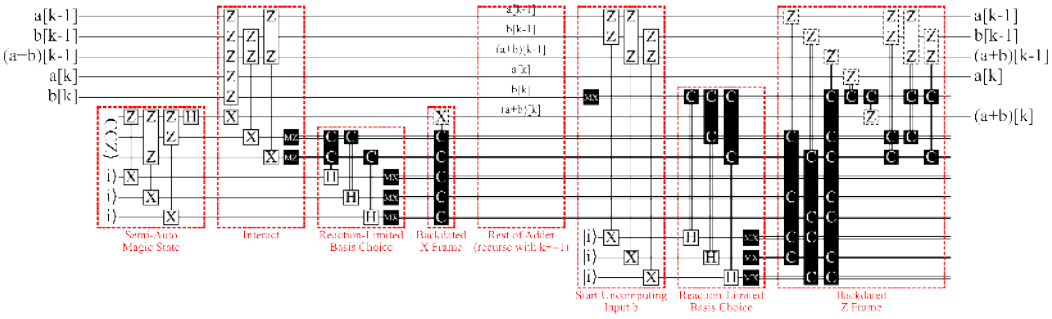

Figure 8: A quantum circuit diagram of an adder including Clifford corrections. Boxes containing X/Z characters are multi-qubit Pauli product operators. Classical parities are shown using "C" operators. Boxes linked by vertical lines form controlled operations, meaning they only act on their common -1 eigenspaces. Click here to open this circuit in Quirk.

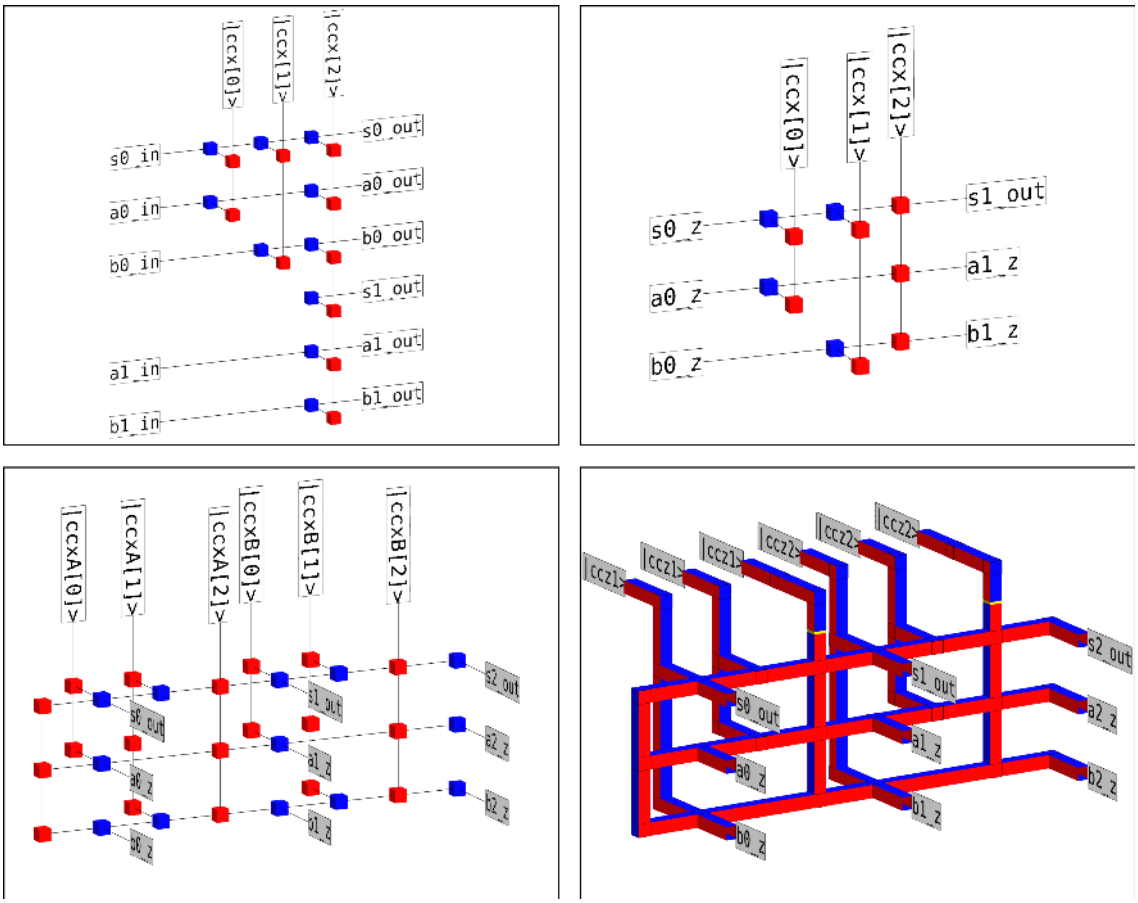

Figure 9: Adder building block diagrams. Top left is a ZX graph implementing Equation 54. Top right is the graph after merging input/output ports into Z ports. Bottom left is a complete 3-qubit adder built by attaching together multiple instances of the graph. Bottom right is an equivalent lattice surgery diagram.

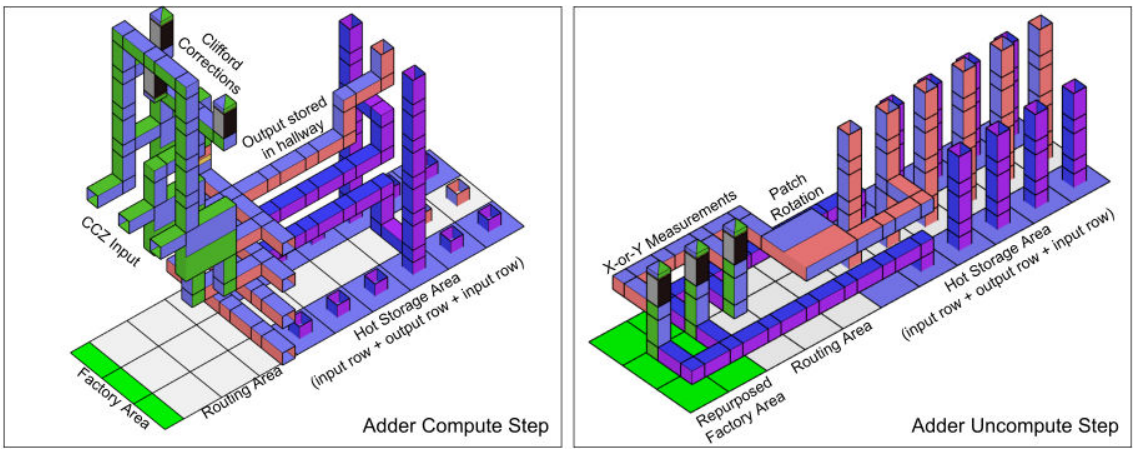

Figure 10: Manual lattice surgery mock-up of adder compute and uncompute steps. Note that the steps tile horizontally, and Clifford corrections are being done after the operation (rather than in a reaction limited fashion). The uncomputation step assumes one of the inputs comes from a lookup, meaning its qubits can be removed by measurement based uncomputation with deferred phase corrections. Green highlights show CCZ-state-associated lattice surgery. Purple highlights show input-qubit-associated lattice surgery.

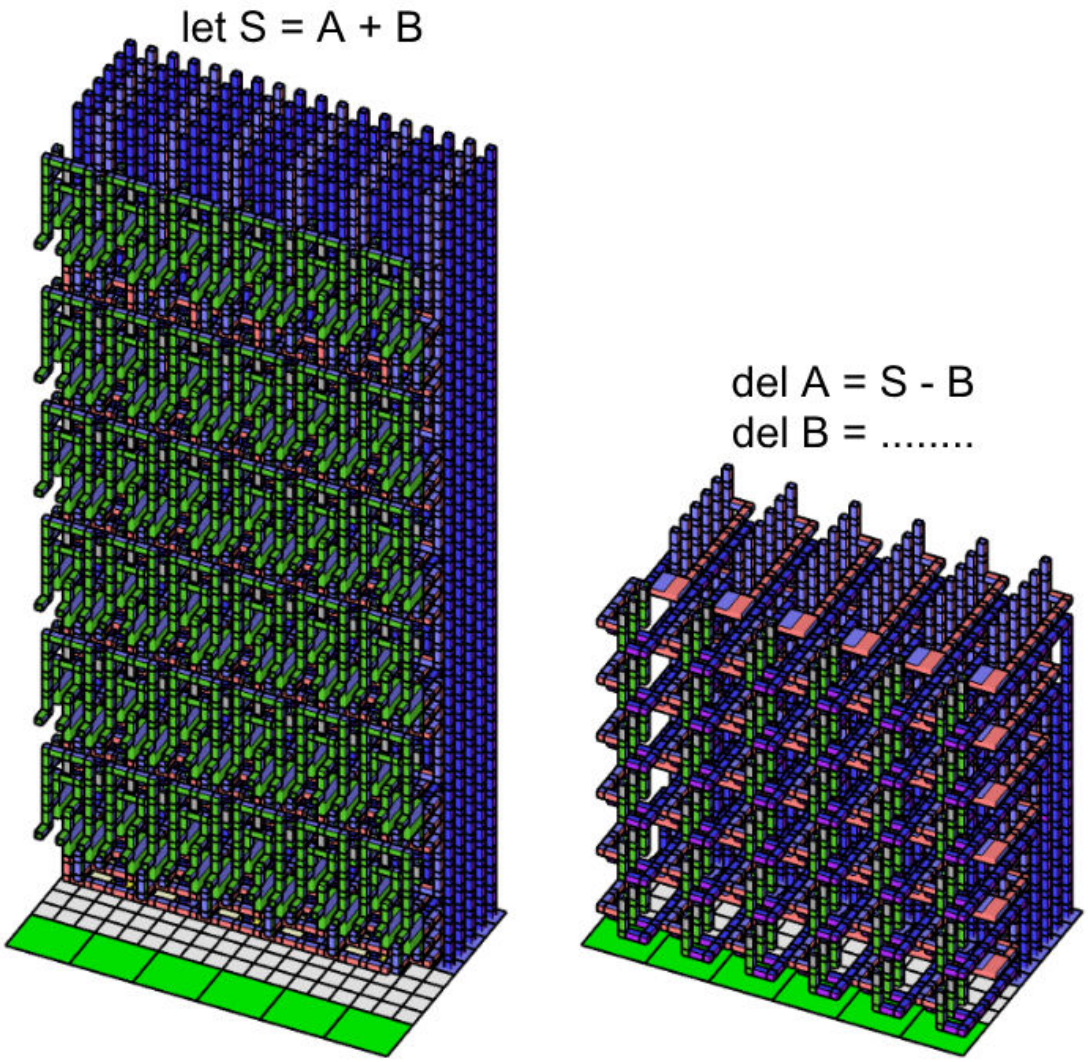

Figure 11: Manual lattice surgery mock-up of a 35 qubit inplace addition. Only intended to give a qualitative sense of the high level layout; some details are omitted and the 3d model likely contains some minor mistakes. Uses many copies of Figure 10. The offset B can be deleted in tandem because in context it comes from a lookup addressed by qubits stored elsewhere.

A.3 Phaseup Operation

A "phaseup" is a quantum operation that negates the amplitude of computational basis states determined by a classical table of bits T. A phaseup acts on an n qubit address register and is driven by a 2^n bit table T. Typically n will be small, for example $n = 6$, due to the exponential size of the table. Phaseups appear when uncomputing table lookups with measurement based uncomputation [Ber+19].

$$\text{Phaseup}_T = \sum_{k=0}^{2^n-1} \text{negif}(T_k)|k\rangle\langle k| \tag{55}$$

For this paper, I imagined phaseups to be implemented by splitting the register into a low half L (with the $n//2$ least significant qubits) and a high half H (with the $n - n//2$ most significant qubits). This effectively reshapes T into a $2^{\text{len } L} \times 2^{\text{len } H}$ matrix, with rows/columns indexed by H, L. Each half-register would then be expanded into its "power product": a register storing every possible product that can be formed out of the original register's qubits. For example, if L was a three-qubit register storing the computational basis state $|c, b, a\rangle$ then its power product L^* is the computational basis state $|cba, cb, ca, c, ab, b, a, 1\rangle$. Power products can be computed with a series of AND gates (for example, see the left side of Figure 12). Once the power products are computed, they can implement the phaseup operation using a series of "masked phase flips". A masked phase flip is a multi-controlled Z gate, with the involved qubits determined by the positions of 1s within a binary mask m:

$$\text{MaskedPhaseFlip}_n(m) = \sum_{k=0}^{2^n-1} \text{negif}(k \,\&\, m = m)|k\rangle\langle k| \tag{56}$$

To implement the desired phaseup in terms of masked phase flips, it's necessary to determine the values of the masks. The data in T lists whether or not to phase flip each possible value of the register pair (H, L). We instead want an alternate form $\overline{T}$ that lists whether or not to include/exclude each possible masked phase flip (i.e. the coefficients of the phaseup's EXOR polynomial [MP01]). Conveniently, this conversion corresponds to a matrix multiplication:

$$\overline{T} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}^{\otimes \text{len } H} \cdot T \cdot \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}^{\otimes \text{len } L} \pmod 2 \tag{57}$$

$$\text{Phaseup}_T = \prod_{l=0}^{(\text{len } L^*)-1} \prod_{h=0}^{(\text{len } H^*)-1} \text{CZ}(H^*_h, L^*_l)^{\overline{T}_{h,l}} \tag{58}$$

In Figure 12 I show a circuit diagram of a phaseup implemented this way, including the data-driven CZ gates as well as the computation and uncomputation of the power products. Note that this circuit has optimized out the two trivial qubits $L^*_0 = H^*_0 = |1\rangle$, resulting in some CZ gates becoming Z gates and one CZ gate becoming an irrelevant global phase (not shown). Also note that Figure 12 has grouped the many individual CZ gates into a few large multi-target CZ gates. This is beneficial because in lattice surgery the marginal cost of targeting a larger Pauli product is notably lower than the marginal cost of doing another gate [FG18; Lit18].

I apply one key additional optimization beyond what's shown Figure 12. I allow the AND computations that appear during the computation of the power product to "wander". That is to say, I perform AND gates by gate teleportation but don't correct the teleportation. An AND gate computes $c = a \cdot b$, whereas a wandering AND gate computes $c = (a \oplus x) \cdot (b \oplus y)$ for measured random values x and y. Normally these x and y values are removed with corrective CNOT gates, performed on the quantum computer. Instead, I account for the CNOT gates by computing a correction matrix C to be multiplied into $\overline{T}$. (C also affects the phase corrections performed during the uncomputation of the power products.) In other words, the CNOT corrections created by the wandering AND gates are handled by rewrites in the classical control system instead of by extra quantum gates. This is analogous to how [Lit18] skips performing Clifford gates by folding their effects into the Pauli product operators targeted by T gates and measurement operations.

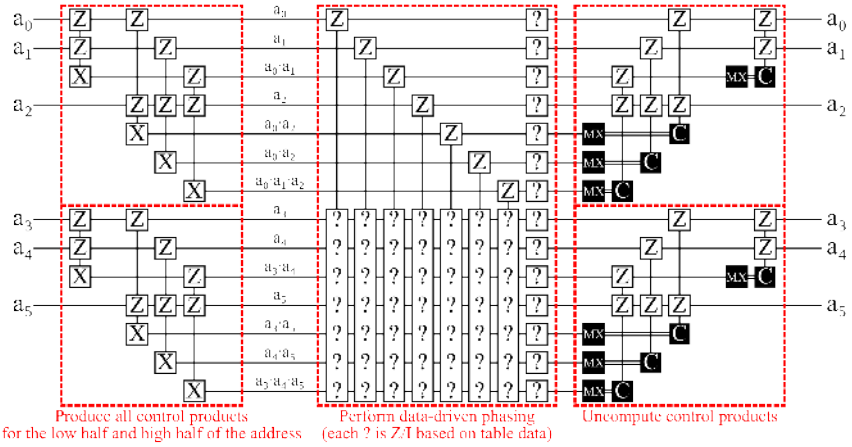

Figure 12: Circuit diagram of a phaseup. On the left, the input register is split into two halves and each half is expanded into its power product using AND gates. In the middle, the desired phase flips are encoded into the presence/absence of Z and CZ gates. On the right, the power products are uncomputed using measurement based uncomputation. Click here to open this circuit in Quirk.

Because wandering AND gates are used when computing the power products, computing the power products has constant reaction depth. Instead of needing to separately correct each layer of AND gates, the corrections merge into a single massive change to the set of performed Z and CZ gates. Only the uncomputation needs to be corrected layer by layer. So, interestingly, the overall reaction depth of the phaseup is $n/2 \pm O(1)$ instead of $n \pm O(1)$.

Overall, a phaseup can be done with $2\sqrt{2^n}$ AND gates, $2\sqrt{2^n}$ workspace qubits, and $\sqrt{2^n}$ multi-target CZ gates. This is reminiscent of the $O(\sqrt{n})$ costs of select-swap lookups [LKS24], which similarly starts by dividing the input register into two halves. See Figure 13 for a lattice surgery mock-up of a phaseup operation implemented in this way.

A.4 Lookup Operation

A lookup is an operation that initializes w output qubits using values from a classical table storing 2^n w-bit integers, indexed by an n qubit quantum address:

$$\text{Lookup}_T = \sum_{k=0}^{2^n-1} \Big(|T_k\rangle \otimes |k\rangle\Big)\langle k| \tag{59}$$

For this paper, I imagined implementing lookups similar to phaseups. First, split the address register into two halves and compute the power product of each half. Second, perform multi-target Toffoli gates (one for each pair of control qubits from the two half registers) to initialize the output qubits. As in the phaseup, the AND gates in the power product computation are allowed to wander. But now the multi-target Toffoli gates are also allowed to wander. This creates dependencies between the different Toffoli gates, where the teleportation outcome of one can change the qubits that must be targeted by another. These changes can be accounted for by the classical control system, but the Toffolis must be carefully ordered to avoid having to redo work and to avoid incurring reaction delays.

After the Toffolis finish, the lookup is completed by measuring all ancillary qubits in the X basis. This creates an enormous number of phase corrections, but every one of these corrections corresponds to phase flipping some subset of the address values. Therefore all these corrections can be merged into a phaseup, which can be deferred and merged into the phaseup that appears in the uncomputation of the lookup later on.

I show a mock-up for the lattice surgery of a lookup in Figure 14.

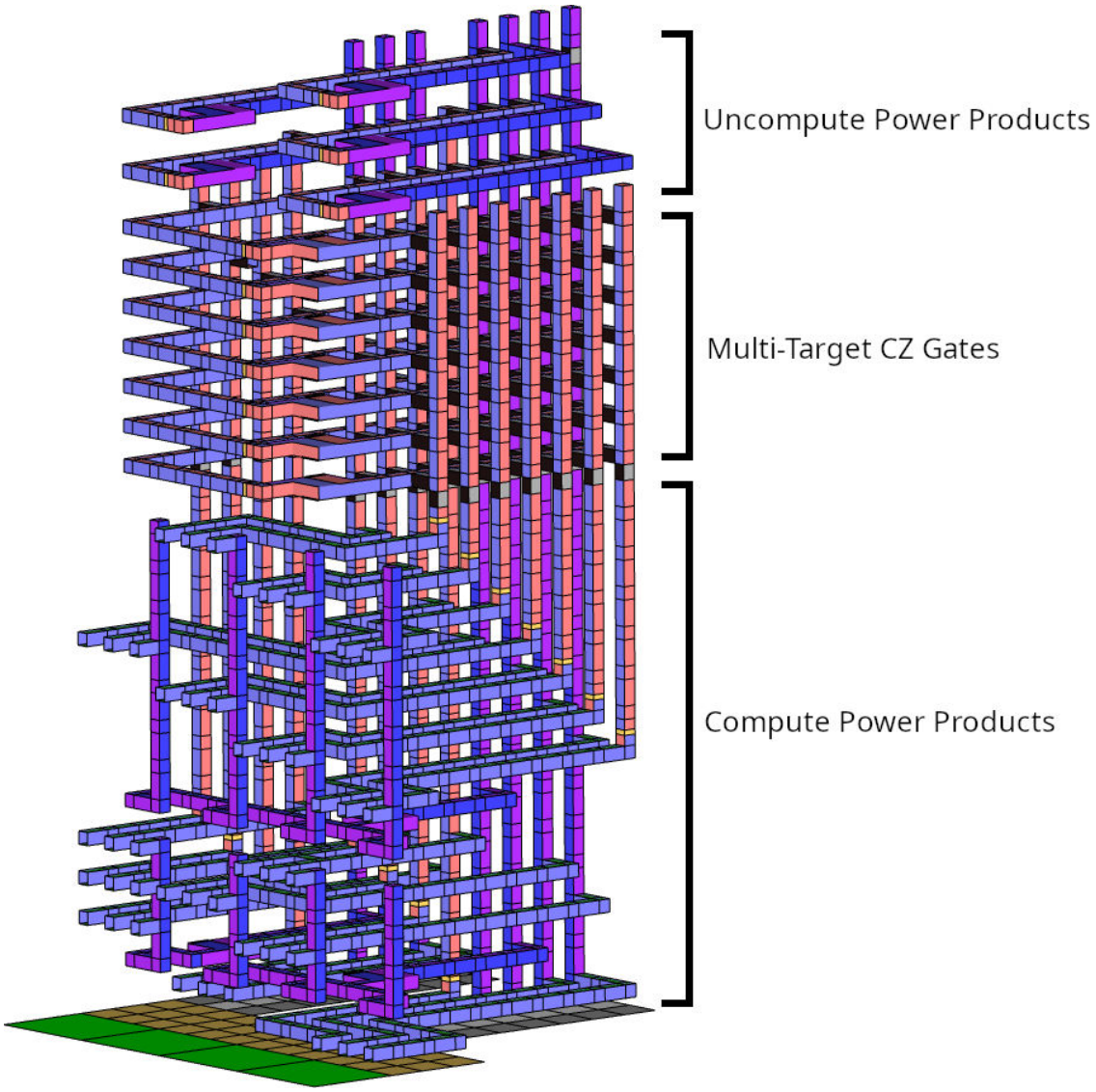

Figure 13: Manual mock-up of lattice surgery performing a phaseup operation. Only intended to give a qualitative sense of the high level layout; some details are omitted and the 3d model likely contains some minor mistakes. Address qubits are accented with purple, CCZ qubits are accented with green, output qubits are accented with red, and delays for control system reaction time are accented with black and white.

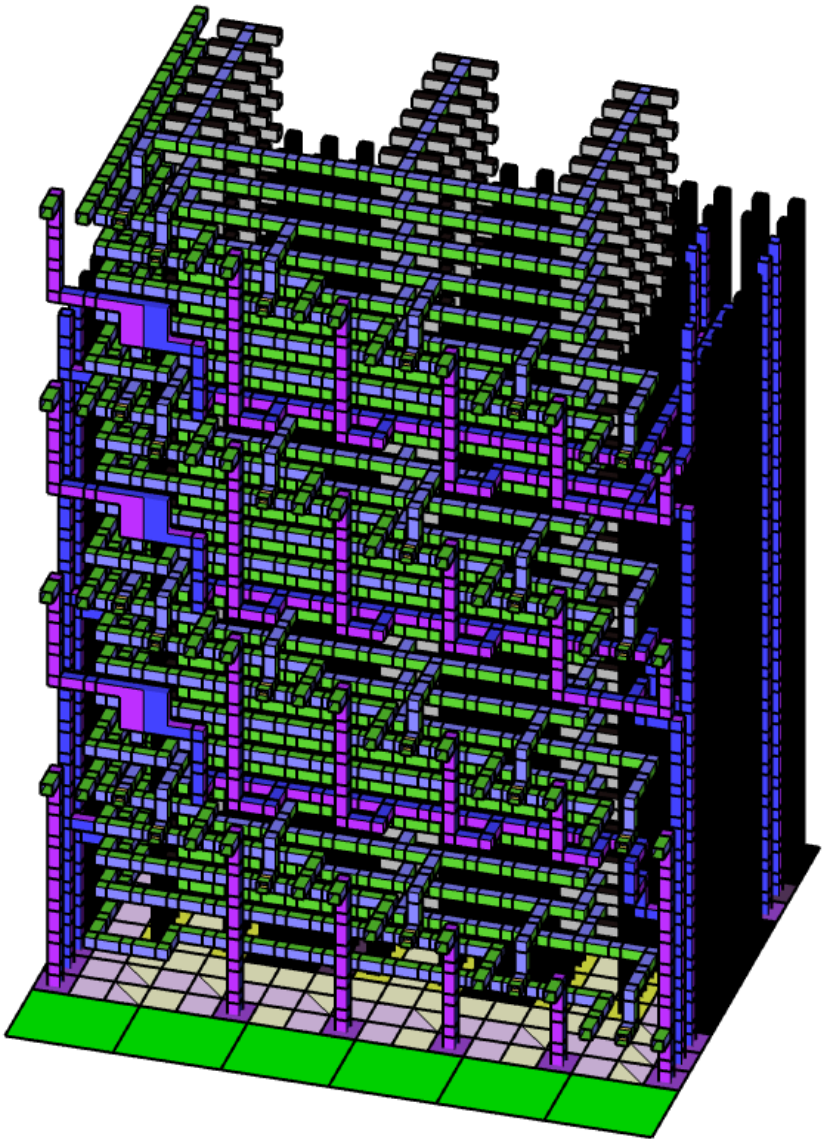

Figure 14: Manual mock-up of lattice surgery for a lookup operation. Only intended to give a qualitative sense of the high level layout; some details are omitted and the 3d model likely contains some minor mistakes. 21 multi-target CX gates are included in the diagram, out of the 63 that would be performed by a lookup with 6 address qubits. The computation of the two power products isn't shown. Green-accented pipes are CCZ qubits, purple-accented pipes are power product qubits, and black pipes are immobile qubits. One of the CCZ qubits pauses momentarily when leaving the factory, before rotating to become the target of a Toffoli state, to give time for the control system to update its targets in response to previous gate teleportations. One of the power products is keeping its qubits still, to be reached by one of the qubits of the CCZ state during each multi-target Toffoli. The other power product is proactively moving its qubits in front of the magic state factories, to intercept one qubit from each CCZ state.

A.5 Frequency Basis Measurement

A frequency basis measurement is implemented by performing an inverse Quantum Fourier transform (QFT) and then measuring in the computational basis. Because of the presence of the measurements, the deferred measurement principle can be used to merge the quadratically many controlled phase gates that normally appear in a QFT circuit into $n-1$ phase gates with adaptively determined phase angles [PP00; ME99] (see Figure 15).

To implement phase gates, I use kickback from additions into a phase gradient state [KSV02; Gid16; Gid18; NSM20]. A phase gradient state is a g qubit state where qubit k is in the state $Z^{2^{-k}}|+\rangle$. Equivalently, it's a uniform superposition where the amplitude for the computational basis state $|k\rangle$ has been phased by $k/2^n$ turns:

$$|\text{GRAD}_g\rangle = 2^{-g/2} \sum_{k=0}^{2^g-1} |k\rangle e^{i2\pi k/2^g} = \bigotimes_{k=0}^{g-1} Z^{2^{-k}}|+\rangle \tag{60}$$

If a classical constant C is added into a g qubit phase gradient state, controlled by a qubit q, then phase kickback will phase q by $-C/2^g$ turns while leaving the phase gradient state unchanged. C is chosen by rounding the desired angle θ to the nearest multiple of $2\pi/2^g$.

Rounding θ introduces a per-rotation error. The maximum over/under rotation is $\pi/2^g$ radians. The chance of a shot failure due to this rounding, across the $n-1$ rotations done by the frequency basis measurement, is at most $n\frac{\pi}{2^g}$.

Another source of error is the infidelity of the phase gradient state, due to approximate preparation. This error differs from the rounding error in that it only applies once per state, instead of once per use. The phase gradient state is an eigenstate of addition and, once prepared, is only used as the target of additions. Therefore the prepared state could be measured in the eigenbasis of addition without changing the behavior of the algorithm, projecting the prepared state into a perfect state with probability equal to its fidelity. So the phase gradient preparation contributes a shot failure probability no larger than the infidelity of the state, regardless of how many times the state is used.

To prepare phase gradient qubits, I use single qubit Clifford+T sequences [RS14]. I convert the sequences into a form that begins with a Pauli basis initialization, ends with a Clifford rotation, and otherwise alternates between performing $T_X^{\pm 1}$ and $T_Z^{\pm 1}$ where $T_Z = Z^{1/4}$ and $T_X = X^{1/4}$. In terms of lattice surgery, I imagine executing the sequence with the help of four neighboring patches. Each neighbor will repeatedly cultivate a T state, undergo a parity measurement against the target patch, and then undergo an adaptively chosen measurement. When a $T_X^{\pm 1}$ gate is needed, an M_{XX} measurement is performed between the target patch and one of its X-boundary neighbors that has a T state ready. This measurement reveals whether or not the target was rotated by $+45°$ or $-45°$ around the X axis. If it rotated in the correct direction, the neighboring patch is measured in the Z basis. Otherwise it's measured in the Y basis, correcting the direction of the rotation [Lit18] (up to classically tracked Paulis). The same story occurs for $T_Z^{\pm 1}$ gates, but with X and Z swapped. This strategy results in lattice surgery operations spiraling around the target patch, with the neighbors taking turns contributing T states (see Figure 16).

Concretely, I would use the gate sequences shown in Table 6 to prepare the phase gradient state's qubits. The first 11 qubits of the state are prepared using a total of 159 T gates. All further qubits, up to the desired length g, are approximated to high fidelity by $|+\rangle$ states. This preparation has an infidelity of $5 \cdot 10^{-6}$, assuming perfect gates. If the T states powering the gates are cultivated to an infidelity of $10^{-7}/4$, then inaccurate T gates will introduce an additional infidelity of $4 \cdot 10^{-6}$. So, in total, preparing the phase gradient state in this way contributes a less than 10^{-5} chance of algorithm failure. This chance could be reduced, by distilling better T states and using the more accurate gate sequences from Table 7. But ultimately the error costs and operation costs of the frequency basis measurement are completely negligible compared to other costs in the paper, so for the purposes of cost estimates I simply ignore it.

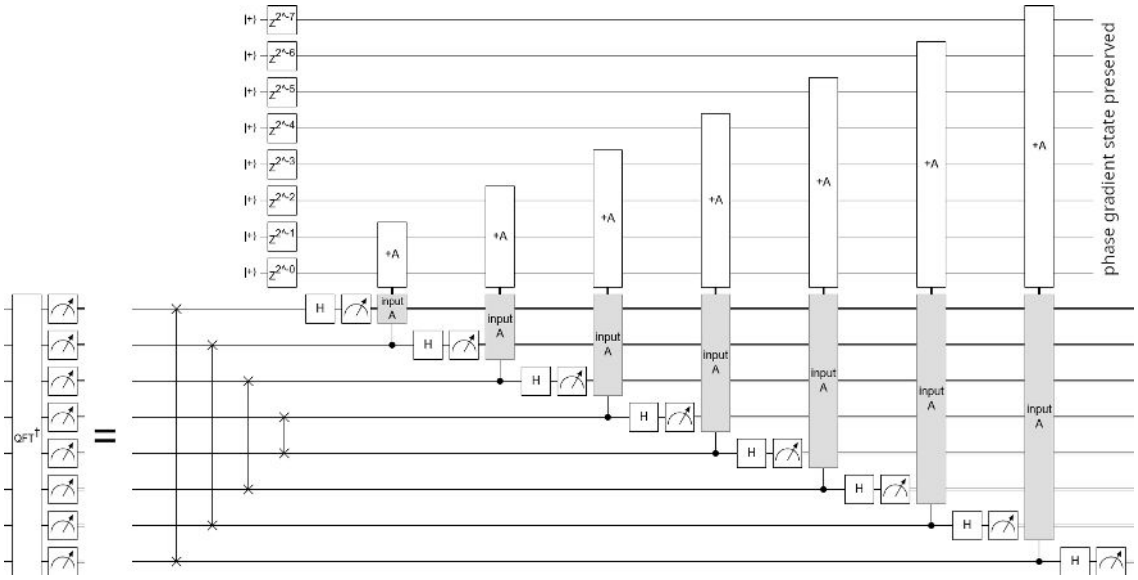

Figure 15: A frequency basis measurement implemented with adaptive phase gates [PP00; ME99] achieved by kickback from controlled addition of a classical constant into a phase gradient state [KSV02; Gid18]. The phase gradient state can be truncated, resulting in an approximate QFT [NSM20]. Click here to open this circuit in Quirk.

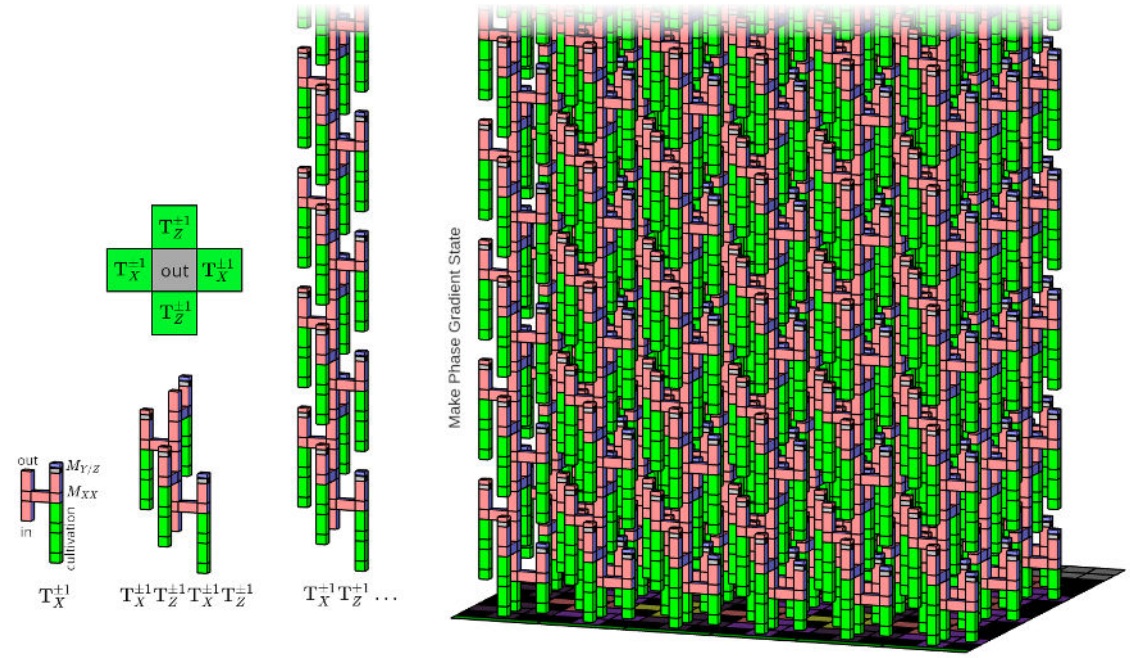

Figure 16: Manual mock-up of lattice surgery preparing a phase gradient state. Left: applying a $T_X^{\pm 1}$ gate by magic state cultivation and teleportation. The sign of the exponent is controlled by picking the basis of the $M_{Y/Z}$ measurement, accounting for the result of the M_{XX} pair measurement [Lit18; GF19]. Left middle: a central qubit can be surrounded with $T_X^{\pm 1}$ and $T_Z^{\pm 1}$ gates taking turns operating on it. Middle: this process can be continued as long as needed. Right: many phase gradient qubits being prepared in parallel.

Qubit Index (k in $Z^{2^{-k}}\|+\rangle$)	Init Basis	T Gate Directions (the signs in $T_X^{\pm1}, T_Z^{\pm1}, T_X^{\pm1}, T_Z^{\pm1}, \ldots$)	Finish Clifford	T Count	Infidelity
0	R_X		Z	0	0
1	R_Y			0	0
2	R_Z	+	H	1	0
3	R_Z	+--++-++--+-+++-+----+	H,Y	22	5.8e-7
4	R_Z	++++-----+--+--+-++-++	H	22	5.7e-7
5	R_Z	++----++-++-+++++---	X	21	2.8e-8
6	R_Z	---++-++++--++-----		21	3.2e-8
7	R_Z	+-++--++----	H	12	5.0e-7
8	R_Z	++--++---++-++-+--+	X	19	5.3e-8
9	R_Z	+--+--++-+-++---+----		21	7.2e-7
10	R_Z	++----++++++----++-	H	20	1.2e-7
11	R_X			0	5.9e-7
12	R_X			0	1.5e-7
Totals				159	3.4e-6

Table 6: Example decompositions of phase gradient qubit initializations, with a per-sequence infidelity target of 10^{-6}. Found using a meet-in-the-middle search. Each qubit is prepared with a Pauli basis initialization, then alternating $T_X^{\pm1}$ and $T_Z^{\pm1}$ gates, then a Clifford rotation. The string of signs specifies the sign of each $T_X^{\pm1}$ and $T_Z^{\pm1}$ gate in sequence, with the leftmost sign corresponding to the first $T_X^{\pm1}$ gate.

Qubit Index (k in $Z^{2^{-k}}\|+\rangle$)	Init Basis	T Gate Directions (the signs in $T_X^{\pm1}, T_Z^{\pm1}, T_X^{\pm1}, T_Z^{\pm1}, \ldots$)	Finish Clifford	T Count	Infidelity
0	R_X		Z	0	0
1	R_Y			0	0
2	R_Z	+	H	1	0
3	R_Z	+-+----++------++-+----+-----++-+++--+++----+++-++	H,Y	50	0
4	R_Z	----+++-+-+---+++-++--+-+-+-++++--+++-+-++--+-	H,Z	46	4.1e-16
5	R_Z	+-+--+----+---++-+++++--+++-++----++-++----+-++-+	S,H	48	6.8e-16
6	R_Z	---+++-------++--+++-+--++-++--+-+-+++++++----+-+	X	49	4.1e-17
7	R_Z	++---++++--+---++++-+---+-++-+-+++-+++--++-----+	H	50	6.9e-16
8	R_Z	++-++-+++--++-++-++++-++--++++++----------+--	Y	45	3.0e-16
9	R_Z	+---+--++++++++--+--++--++-+-++---+-++----+-+-++	H	48	2.3e-16
10	R_Z	+++--+--+-+++++++-++++-+-++-+-+++-++---++++-+---		49	5.0e-17
11	R_Z	++-+--++-++--++-----+-+-+-+---+++-------+-+--++	Z	47	9.7e-16
12	R_Z	++------++++++--+--+-++-+----+-++++--+++---+	Y	45	6.8e-16
13	R_Z	+++---+--++++-++--+++--++--+-++-++++--+-++--+	H,Z	46	6.9e-16
14	R_Z	+++--+-----+++---++++-+-++++-+++----++----+-+--+		49	3.9e-16
15	R_Z	+---++-+--+-++---+--+++-++-++++-++++-+++-+--+	H,Y	46	6.2e-16
16	R_Z	+--+-+-++--++++-+-++---++++-+-++++--+---+-++-+	H,Y	48	4.9e-16
17	R_Z	+-++----+---++++-+-+++-+-++++-+-+----+-++--		43	5.3e-16
18	R_Z	++---+--++--+---+--+-+---+---------++++++++--++--	X	49	1.8e-16
19	R_Z	+--+--+-+-+------+---+--+---++-+--+-+++-+-+++-++	Z	49	1.3e-16
20	R_Z	----+-++++--+-++--+++--+++--++------+-+-+---++--		49	9.7e-16
21	R_Z	++---++-++-+--++----+-+-+--+-++++--++-++---+++++	H	48	6.1e-16
22	R_Z	+-++-+-+----+++--+-++--+-+-+-++-++---++-+--++++-+--++	H,Z	52	3.4e-16
23	R_Z	+++---+---+-+--++--+------+--++--+-+---+---+++-	X	47	5.9e-16
24	R_Z	++++-----+-++---+-+++---+-+-++++-+++-++++-++++--+	X	49	2.5e-16
25	R_Z	+-+-++---+++-++++-+-+-++++-+-+-++++-+++---++-+-++		49	2.9e-16
26	R_X			0	5.5e-16
27	R_X			0	1.4e-16
Totals				1102	1.1e-14

Table 7: Same as Table 6 but with a per-qubit infidelity target of 10^{-15}.