Computing 256-bit elliptic curve discrete logarithms in 26 days on a fault-tolerant trapped-ion quantum computer with 20,000 qubits

Thomas Häner,* Felix Tripier,* Jacob Young,* Michael Naehrig, Andrii Maksymov, Safwan Alam, Dmitri Maslov, Matthew Parrott, Yvette de Sereville, Jordan Sullivan, Mark Webster, Nicolas Delfosse, John Gamble, and Martin Roetteler
IonQ Inc.

(Dated: September 3, 2026)

One of the strengths of our recently proposed *Walking Cat* Architecture [1] for a trapped-ion quantum computer is that it is straightforward to extend and optimize for a specific application. As a proof-of-concept, here we present such optimizations for solving the 256-bit elliptic curve discrete logarithm problem (ECDLP) on `secp256k1`, which is the elliptic curve used by blockchain technologies such as Bitcoin, using Shor's algorithm. We optimize the circuits from Schrottenloher's recent work [2] and arrive at a logical quantum circuit for solving the ECDLP using about 1450 qubits and $40 \cdot 10^6$ Toffoli gates, with a rigorous lower bound on the logical-level success probability that holds with confidence at least $1-2^{-128}$, as well as a heuristic estimate thereof. Using our compilation toolchain in combination with manual optimization of the logical layout and integrated routing, we produce estimates for the logical measurement depth and the required number of physical qubits by compiling all components to measurement schedules that obey the architectural constraints. A key ingredient is a fast CCZ magic-state factory and a depth-one CCZ state injection, reducing the execution time of CCZ gates by a factor of 31. We increase the logical-measurement parallelism using non-overlapping cat-based measurements in parallel, and we leverage the recently proposed logical CliNR protocol to speed up Clifford operations. To reduce the qubit overhead, we introduce a more efficient loss correction protocol, design a layout that allows us to recycle the CliNR ancilla qubits, and provision reusable cat-state resources according to the circuit's peak measurement parallelism. All results and optimizations combined, we conclude that a trapped-ion quantum computer based on our architecture would be able to solve the ECDLP on `secp256k1` in approximately 25.7 days using 19,397 physical qubits with an estimated success probability of 63%.

* These authors contributed equally to this work.

CONTENTS

Part I
Introduction and overview

I. INTRODUCTION

Quantum computers have the potential to deliver substantial speedups over their classical counterparts for certain computational problems in cryptography, chemistry, materials science, data science, differential equations, and optimization [3–14], and are thus expected to accelerate the process of scientific discovery [15].

Resource estimates for some of these applications have been used to measure progress of the field, because such estimates capture improvements in quantum algorithm design, circuit optimization, fault-tolerant quantum computing architecture and quantum error correction. For example, the number of physical qubits required for factoring 2048-bit numbers using Shor's algorithm [3] has dropped by three to four orders of magnitude over the last 14 years [16–19], from approximately 10^9 qubits in 2012 [20, 21] to 10^6 qubits with 2D grid connectivity [22] or 10^5 qubits when assuming long-range connectivity [23].

Similar progress has been made for solving the elliptic curve discrete logarithm problem (ECDLP). Since the initial estimates by Proos and Zalka from 2003 [24] with a gate count of $6 \cdot 10^9$, a series of works [25–27] has reduced the resource requirements (without amortizing over multiple instances [27]) to about $200 \cdot 10^6$ Toffolis [27] in 2023, and finally to 1462 logical qubits and $84 \cdot 10^6$ Toffoli gates [2] in 2026. In addition, recent work has also explored reducing the space requirements and managed to achieve logical qubit counts of about 1200—using more space-efficient adders [2, 28–30] or a compression of the output [31]—and even below 850 [32] using location-controlled arithmetic, but the resulting Toffoli counts are (in the latter two cases significantly) larger.

Moreover, the 256-bit curve `secp256k1` admits additional optimizations, because its prime is pseudo-Mersenne. As a result, gate counts for solving the ECDLP on `secp256k1` can be lower than those for other elliptic curves [2, 28]: about 1462 qubits and $60 \cdot 10^6$ Toffoli gates have been shown to be sufficient [2]. Recent high-level resource estimates suggest that solving this problem might require under 20,000 physical qubits on a neutral-atom quantum computer [33], or fewer than 500,000 physical qubits on a superconducting quantum computer [28].

In this work, we design a specialized architecture for a fault-tolerant trapped-ion quantum computer capable of solving the ECDLP on `secp256k1` in 26 days using approximately 20,000 qubits, whereas previous estimates for solving this problem with a trapped-ion architecture required between 1.2 million qubits [34] and 9.4 million qubits [27]. Our work goes beyond previous resource estimation studies [2, 26–28] and architecture proposals [33, 35] in several ways. In particular, we make the following contributions:

1. We develop and describe a detailed trapped-ion architecture for the ECDLP, including a compiler and complete descriptions of all logical operations, quantum error-correcting codes, and error-correction circuits. This design is compatible with the Walking Cat architecture [1]. To our knowledge, it is the first ECDLP architecture for trapped ions based on quantum LDPC codes.

2. We reduce the Toffoli count of Schrottenloher's circuits [2] from $2^{25.78} \approx 58$ million to 39 million Toffoli gates at 1457 logical qubits.

3. We derive rigorous bounds on the success probability of the entire quantum algorithm, taking into account approximation errors (phase as well as computational basis errors) in arithmetic circuits.

4. We reduce the execution time of Toffoli gates in the Walking Cat architecture by a factor of 31 by introducing a Toffoli state factory, a fast Toffoli injection circuit, and by increasing the cat-based logical-measurement parallelism.

5. We quantify and take into account the overheads associated with the layout of arithmetic components and the allocation and transport of ancilla qubits consumed by logical operations.

6. We implement a compiler to provide accurate depth estimates and physical qubit counts by compiling all high-level algorithmic components to executable programs for our architecture, replacing all naive Clifford decompositions with integrated logical routing and getting exact depth counts.

7. We consider the impact of qubit transport, leakage, loss, and qubit reloading on the logical operation time and noise, which is significant for atomic qubits and often ignored in resource estimations [23, 27, 33].

8. We derive conservative estimates for logical operation times based on a detailed structure of the syndrome extraction circuit, whereas some previous estimates assume a syndrome extraction time of 1ms for any code which may lead to over-optimistic time estimates [23, 27, 33].

9. We account for all logical operations including Toffoli gates, Clifford gates, and logical measurements which constitute a significant fraction of the total runtime but are often omitted from other resource estimates [23, 28, 33]. For example, half of the cost of a Gidney adder [36] lies in the measurement-based uncomputation of temporary ANDs, a cost not captured by its Toffoli count.

Throughout this work, we prioritize simplicity over exhaustive optimization. For example, increasing the amount of classical postprocessing or executing more instances of the quantum algorithm, potentially in parallel, could reduce the size of the quantum circuit required for each instance [31, 37]. In the compiler, we use a single adder family throughout the application rather than switching between width- and depth-optimized adders according to the available scratch space and runtime requirements. At the architecture level, we similarly keep the device configuration fixed throughout the computation rather than dynamically reallocating qubits between memory and gate resources. Finally, we report a conservative upper bound on runtime by treating every logical measurement as lasting as long as a Toffoli injection. We likewise find a lower bound on the success probability by assuming that all 69 memory blocks remain active throughout the computation, although the lifespan of memory is generally shorter. These choices leave room for further optimization, but we forgo changes that offer only marginal gains in favor of a simpler design that is easier to analyze and verify.

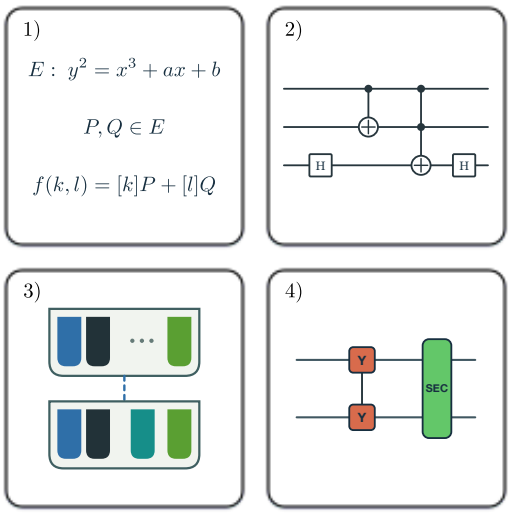


FIG. 1: Overview of the paper in terms of an end-to-end pipeline of increasingly lowered representations of Shor's algorithm for the ECDLP. Stage 1 gives the high-level mathematical description of the algorithm (Part II). Stage 2 describes it in terms of a high-level logical quantum circuit (Part III). Stage 3 compiles these circuits into executable, architecture-legal measurement schedules (Part IV). Panel 3 illustrates the assignment of logical qubits to memory blocks across successive measurement layers: each box represents a memory block, each colored rectangle represents a logical qubit, and the dashed line denotes a logical-qubit move between memory blocks performed by integrated routing. Stage 4 lowers the schedules to native operations on the QEC architecture using its logical instruction set architecture (ISA) (Part V).

II. OVERVIEW

A. Architecture

Our ECDLP-specialized architecture is derived from the Walking Cat Architecture [1], which is a general-purpose architecture for fault-tolerant quantum computing with trapped ions [38–42]. The Walking Cat Architecture relies on a 2D grid of trapped ions, each encoding a qubit, which can be transported across the grid, leveraging electronic qubit control [43], to establish long-range connections between qubits. Transport enables the use of quantum LDPC codes [44], making the architecture more efficient than surface-code-based architectures [21, 45]. We assume two-qubit gates with a noise rate of 10^{-4}, which has been demonstrated experimentally on small trapped-ion devices [46], and single-qubit operations with a noise rate of 10^{-5}, which is also within reach of current trapped-ion devices [47].

A naive implementation of Shor's algorithm for solving the ECDLP on the general-purpose Walking Cat architecture [1] is possible but faces the following challenges. (1) The magic factories of [1] produce T states (more precisely H states) with a logical error rate of $7 \cdot 10^{-8}$ at best. However, solving the ECDLP on `secp256k1` using our circuits would require 39 million Toffoli gates consuming a total of 273 million T states. Here we assume the implementation [48] for exact Toffoli (which uses seven T gates and seven CNOT gates) rather than the relative-phase-Toffoli-based four-T constructions of Jones [49] and Maslov [50], since the latter are more challenging to compare costs to as they require an additional logical ancilla. The T state logical error rate is not sufficient to achieve a high probability of success for the algorithm. (2) Each T state injection consumes a logical measurement which takes an average of about 40ms and a CNOT gate takes an average of about 90ms (assuming 3 logical measurements per CNOT gate, 5 SECs per logical measurement and 6.7ms per SEC). If they are implemented sequentially, this would result in a runtime of at least 910ms per Toffoli gate and 411 days in total. One could increase the number of magic factories, but this only gives a small improvement because the Toffoli-width of this algorithm is limited to one, which limits its T-width. Indeed, Babbush et al. assume sequential Toffoli gates in their resource estimation [28]. (3) Storing 1500 logical qubits requires a large number of physical qubits. For example, storing the logical qubits using a distance-9 surface code would consume 243,000 qubits, excluding the cost for logical operations and magic factories. One could use the Walking Cat architecture code Q70 encoding 6 logical qubits into 70 data qubits, but it would still require 55,000 qubits split into 250 memory blocks, ignoring the cost of cat factories and magic factories. A reasonable option in terms of qubit count is the code Q102, which encodes 22 logical qubits into 102

data qubits (using 102 ancilla qubits for syndrome extraction and 102 beacon qubits for loss correction), consuming about 20,000 qubits to store 1500 logical qubits. (4) Using a dense code like Q102, which encodes many logical qubits in each block, may reduce logical gate parallelism because it is challenging to perform multiple logical operations at once in the same block given an arbitrary Clifford frame. One can consider using the recently proposed logical CliNR scheme [51] to simplify Clifford frames, but making CliNR available for each Q102 memory block would require two extra blocks of Q102 for each memory block, adding a total of 40,000 qubits.

To address challenges (1) and (2), we design a two-level magic factory producing magic states with logical error rate below 10^{-9}. Our magic factory directly produces CCZ states in a quantum LDPC code as opposed to manufacturing Toffoli gates from T gates. This results in a dramatic reduction in runtime of the quantum algorithm, which is dominated by Toffoli gates once layout and routing have been optimized. To make both CCZ state production and injection faster, we include multiple cat factories per block and we perform parallel logical operations using disjoint logical operators. This allows us to build a fast CCZ factory and a depth-1 CCZ injection circuit, which we use to perform Toffoli gates as a result of the local Clifford equivalence of CCZ and Toffoli gates. Using fast Toffoli injections and Clifford frame tracking results in a Toffoli gate time of 29.5ms, which is faster than the naive implementation by a factor of 31. Four Toffoli factories consuming a total of $1276 = 4 \cdot 319$ qubits are enough to produce Toffoli states with an average production time under 28ms, providing a continuous flow of Toffoli gates. The challenge (3) is addressed by optimizing the Walking Cat architecture to make use of the dense code Q102. We use the logical CliNR scheme, as suggested in (4), but to avoid the extra cost of 40,000 qubits, we pipeline the implementation of the logical CliNR scheme, leveraging the structure of the quantum algorithm and its limited Toffoli width. We find that 4 extra Q102 blocks are sufficient, a reduction of 39,200 qubits compared to the naive implementation of CliNR. Moreover, we further reduce the qubit count by exploiting the limited Toffoli width of the quantum algorithm. Because only a relatively small number of Toffoli gates can be executed in parallel, many cat states remain idle for prolonged periods of time. Instead of filling all the cat state factories with qubits, we use a restricted number of cat qubits which are transported to different cat factories as needed. We minimize the number of cat qubits required through careful pipelining and routing. Finally, we improve the underlying qubit loss model of the Walking Cat architecture and introduce a new leakage and loss correction unit. Together, these improvements remove the need for beacon qubits, eliminating about one third of the qubits in each memory block.

B. Algorithm and implementation

At the level of logical circuits, we present two optimizations for the quantum circuits by Schrottenloher [2] that result in a reduction of the Toffoli count from $58 \cdot 10^6$ to $39 \cdot 10^6$ at a logical qubit count of 1457. Namely, we employ an alternative representation for the in-place multiplication [2, 52] in which controlled additions become conditionally-inverted additions—which are about half as costly in terms of non-Clifford gates [36]—and we build a modular squarer for pseudo-Mersenne primes based on a single Karatsuba split [53] and a squarer-specific optimization of Litinski's non-modular multiplier [54].

Because the resulting point addition circuits are approximate in the sense that they produce correct outputs and no (-1)-phase errors for most, but not all, inputs from $\mathbb{F}_p$, we derive bounds on the probability of success of individual components and the entire double-scalar multiplication. To enable a tight translation of these component-level success probability bounds to the failure probability of the entire algorithm, we use random offset masks. Because we sample fresh masks for every run of the algorithm, we are able to prove an upper bound p_f on the failure probability for each fixed input scalar pair. In turn, this per-input bound can be turned into a lower bound on the probability of success of Shor's algorithm that is at most $4p_f$ lower than the success probability given an exact implementation of the double-scalar multiplication. Previous work [2, 22, 24, 28] used heuristics and informal fidelity or shot-failure arguments, while bounds via state distance [19, 55] lead to an $\mathcal{O}(\sqrt{p_f})$-bound on the reduction in the probability of success.

We then implement and map these logical circuits to our architecture. We describe our internal compilation toolchain with its verification pipeline, which we use to obtain executable programs for each high-level component in the algorithm, producing rigorous space and depth estimates. To reduce the space-time footprint of the components (adders, multipliers, etc.), we optimize the assignment of logical qubits to the different memory blocks of our architecture. In our final depth estimates, we take into account the overhead due to different such assignments. Unlike prior gate-count resource estimates that abstract away Clifford and routing costs, we compile every charged component to an executable, architecture-legal measurement schedule with routing included. Thanks to our *integrated routing*, i.e., the process of merging cross-block routing operations between two components with those same components, we are able to reduce the routing overhead to 5% of the total runtime.

C. Results and Outline

We combine our accurate depth estimates with (1) the logical measurement times provided by our proposed architecture, (2) the rigorous lower bound on the success probability of the algorithm, and (3) the estimated prob-

ability of a logical error. This allows us to conclude that our proposed architecture, with around 20,000 physical qubits, can solve the ECDLP on `secp256k1` in approximately 25.7 days per attempt. Using Mosca's rigorous lower bound [56] as a starting point, the single-run success probability estimate is 40.7%; using Ekerå's heuristic success probability estimate [37] as the starting point gives 63.3%.

Table I lists the depth for each component, its contribution to the total of all 28 windowed point additions, including the semi-classical inverse quantum Fourier transforms (iQFT), as well as the total runtime in hours and the device's physical-qubit footprint.

The paper is organized along the lines of the pipeline described in Fig. 1. In Part II, we introduce Shor's algorithm for the ECDLP and discuss the success probability of the algorithm when using approximate arithmetic. In Part III, we present the arithmetic circuits and optimizations that we use to implement the double-scalar multiplication. We describe our compiler, the logical layout of components, the integrated routing and our verification pipeline in Part IV. In Part V, we present a quantum charge-coupled device (QCCD) model for trapped ions based on an extension of the Walking Cat architecture [1]. Relative to the baseline architecture, it achieves significant spacetime savings through a logical ISA tailored to Shor's algorithm for the ECDLP and optimized handling of resource states for logic. Finally, we conclude the paper with a discussion of our results in Part VI.

Part II
Elliptic curve discrete logarithms

In this part, we introduce Shor's algorithm for the elliptic curve discrete logarithm problem (ECDLP) and give a bound on its success probability when an approximate version of the oracle is used instead of an ideal implementation.

III. SHOR'S ALGORITHM FOR THE ECDLP

Along with the factoring algorithm, Shor [3] presented an algorithm to compute discrete logarithms. The variant for solving the ECDLP was worked out by Proos and Zalka [24]. Subsequent works explored various optimizations and space-time tradeoffs [2, 25–28, 35].

Our discussion of Shor's algorithm for the ECDLP is restricted to the case of elliptic curves defined over large characteristic finite fields, and we work in a prime order subgroup. Let $E : y^2 = x^3 + ax + b$ for $a, b \in \mathbb{F}_p$ be an elliptic curve defined over the prime field $\mathbb{F}_p$. The set $E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p \mid y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$ of $\mathbb{F}_p$-rational points on E together with the point at infinity $\mathcal{O}$ is an abelian group with neutral element $\mathcal{O}$. For an integer $k \in \mathbb{Z}$ and a point P, we use the notation $[k]P$ to denote the k-fold sum $P + P + \cdots + P$ (for negative k, $[k]P = -[-k]P$). Let $P \in E(\mathbb{F}_p)$ be a point of prime order r, i.e., $[r]P = \mathcal{O}$ and the subgroup $\langle P\rangle = \{[k]P \mid k \in \mathbb{Z}\}$ generated by P has order r. The *elliptic curve discrete logarithm problem (ECDLP)* in $\langle P\rangle$ is: given $Q \in \langle P\rangle$, compute $d \in \mathbb{Z}_r$ such that $Q = [d]P$.

The ECDLP is a special case of period finding and the hidden subgroup problem and uses the quantum Fourier transform (QFT). Naturally, an r-dimensional QFT would be used here, but in practice a 2^n-dimensional one is preferred, where n is the bitlength of r, $2^{n-1} \leq r < 2^n$. In that setting, the algorithm computes the following quantum state $|\Psi\rangle$, a superposition over all scalars $k, l \in [0, 2^n)$,

$$|\Psi\rangle = \frac{1}{2^n} \sum_{k,l \in [0,2^n)} |k\rangle\,|l\rangle\,|[k]P + [l]Q\rangle\,.$$

After applying the inverse QFT to the first two registers, measuring these registers provides a pair of classical values from which the discrete logarithm can be recovered with high probability by classical postprocessing. Using the semi-classical Fourier transform [57], it is possible to interleave the QFT operations on the scalar qubit registers $|k\rangle\,|l\rangle$ with the point addition operations and thus reduce the number of qubits needed to a single one that can be reused. In this paper, we use a windowed version of the semi-classical QFT with window size $w = 16$ [26, 27]

TABLE I: Algorithm component-level layer and runtime accounting for solving the ECDLP on `secp256k1` using Shor's algorithm at 0.02950 s per measurement layer. Each ordinary lookup is charged as ≈236,599 base layers plus an average of 24,483 stochastic-penalty layers (261,082 in total), and each phase-fix lookup is charged as 36,857 base layers plus an average of 11,984 stochastic-penalty layers (48,841 in total). The initial 2^{18} lookup is charged as 940,028 base layers plus an average of 150,513 stochastic-penalty layers (1,090,541 in total). CliNR is described in Part V. The total expected runtime is 616.555 h, or **25.690 days**. The physical-qubit footprint is 19,397; see Part V for the architecture discussion and appendix E 1 for the footprint derivation.

Component	Layers/instance	Instances/window	Layers/window	Layers/28 windows	Runtime (h)
Unary lookup (including penalty)	261,082	3	783,246	21,930,888	179.686
Approximate mod. subtract	912	4	3,648	102,144	0.837
In-place multiplication	820,994	2	1,641,988	45,975,664	376.691
Approximate mod. add	674	3	2,022	56,616	0.464
Square-subtract	123,530	1	123,530	3,458,840	28.339
Approximate mod. negation	179	1	179	5,012	0.041
Unary phase-fix (worst case)	48,841	1	48,841	1,367,548	11.205
Routing between components	20	1	20	560	0.005
CliNR at turnarounds	7.73	5,783	44,703	1,251,684	10.255
iQFT (per-window average)	408	1	408	11,424	0.094
28-window subtotal			2,648,585	74,160,380	607.616
Initial lookup (of size 2^{18})	1,090,541	—	—	1,090,541	8.935
Initial iQFT outside the window loop	408	—	—	408	0.003
Total expected				75,251,329	616.555
Physical-qubit footprint					19,397

and we drop the final three windows in favor of additional classical postprocessing [27, 37]. In addition, we replace the first windowed point addition by a direct table lookup [28], resulting in a total of 28 windowed point additions.

The efficiency and success probability of the overall algorithm is mainly determined by the efficiency and accuracy of the most costly operation, namely computing the quantum operation U_f that represents the double scalar multiplication $f(k,l) = [k]P + [l]Q$ given the fixed points P and Q that constitute the ECDLP:

$$U_f : |k\rangle\,|l\rangle\,|\mathcal{O}\rangle \mapsto |k\rangle\,|l\rangle\,|[k]P + [l]Q\rangle = |k\rangle\,|l\rangle\,|f(k,l)\rangle\,.$$

This paper focuses on quantum circuits for U_f specifically for the elliptic curve `secp256k1` used in Bitcoin. The curve is given in short Weierstrass form as $E : y^2 = x^3 + 7$, defined over the prime field $\mathbb{F}_p$, where p is the pseudo-Mersenne prime

$$p = 2^{256} - c, \quad c = 2^{32} + 2^{10} - 2^6 + 2^4 + 1 = 2^{32} + 977.$$

The curve has prime order, which means that the number $r = \#E(\mathbb{F}_p)$ of points on the curve is given by $r = p + 1 - t$, where t is the trace of Frobenius (here: $t = 432420386565659656852420866390673177327$). We pick this curve for various reasons. First of all, it is used in the Bitcoin system and its security is relied upon to secure vast amounts of cryptocurrency. Concrete resource estimates for this specific curve are of high value. Second, recent resource estimates used to discuss the security of cryptocurrencies against quantum attacks by Babbush et al. [28] and the state-of-the-art circuits by Schrottenloher [2] of lowest cost are based on `secp256k1` as well (due to the special pseudo-Mersenne shape of p).

IV. SINGLE-SHOT SUCCESS PROBABILITY

This section considers the success probability of a single run of Shor's algorithm for the ECDLP. All state-of-the-art implementations of Shor's algorithm have to deal with superpositions that contain a number of failed computations due to the use of optimizations such as approximate arithmetic [2]. It is thus important to analyze how such failures affect the overall success probability.

Here, we bound the reduction in success probability due to the use of an approximate implementation of the double-scalar multiplication that produces wrong outputs or residual phases on a small fraction of inputs.

A. Approximate oracle and success probability

Previous work [2, 24] has found that the cost of the double scalar multiplication can be further reduced by introducing approximations beyond relying on the generic point addition formula, ignoring the few exceptional cases (such as point doubling and adding $\mathcal{O}$). Our circuits also make use of such approximations as well as approxima-

tions that introduce phase errors, meaning that we do not prepare the ideal state

$$|\Psi\rangle = \frac{1}{2^n}\sum_{k,l}|k\rangle\,|l\rangle\,|[k]P+[l]Q\rangle = \frac{1}{2^n}\sum_{k,l}|k\rangle\,|l\rangle\,|f(k,l)\rangle\,,$$

but instead only an approximate state before we measure the input registers in the Fourier basis. We note that as in previous work [26, 27], our actual implementation uses a windowed implementation of the double scalar multiplication with the semi-classical quantum Fourier transform [57]. Since both variants are equivalent, we will pretend for the theoretical analysis that we keep both input registers for the scalars k and l, and that the inverse quantum Fourier transform is applied at the very end of the algorithm before we measure the input registers.

When preparing the approximate state instead of the ideal state above, we also introduce several random masks that are captured by the parameter ω. We prepare a state where the double scalar multiplication output is shifted by a (random) elliptic curve point that only depends on ω, P, Q. For each run of our Shor implementation, we sample a non-zero $a_0 \xleftarrow{\$} \mathbb{F}_r^*$ and initialize the accumulator register with the point $[a_0]P \neq \mathcal{O}$; we also shift our lookup tables by fixed curve points such that none of the table points is the point at infinity $\mathcal{O}$, leading to an additional constant shift by a point $[\mu_P]P + [\mu_Q]Q \in E(\mathbb{F}_p)$ independent of k and l. Therefore, we compute

$$|k\rangle\,|l\rangle\,|\mathcal{O}\rangle \mapsto |k\rangle\,|l\rangle\,|[a_0+\mu_P]P+[\mu_Q]Q+[k]P+[l]Q\rangle\,.$$

This avoids special treatment for the first point addition and the addition of table lookup points, where having $\mathcal{O}$ as one of the summands constitutes an exceptional case for the addition formulas. It also randomizes the occurrence of exceptional cases and arithmetic failures for different input scalars (k, l). While such a shift does not affect the success probability of Shor's algorithm [24], we still assume that it is removed by subtracting the point $[a_0 + \mu_P]P + [\mu_Q]Q$ before we apply the quantum Fourier transform, so that we can ignore it for the analysis. However, since this hypothetical (and exact) removal of $[a_0 + \mu_P]P + [\mu_Q]Q$ operation acts as the identity on the input register, we do not implement it in practice.

We write $f_\omega(k, l)$ for the ideal function that computes $[a_0 + \mu_P]P + [\mu_Q]Q + [k]P + [l]Q$ including the random masks, and $\widetilde{f}_\omega(k, l)$ for the function computed by our circuits using the approximate arithmetic (that we will explain in detail below), which computes an approximation of $f_\omega(k, l)$. Integer-valued versions of the circuits to compute $\widetilde{f}_\omega(k, l)$ are represented in Algorithms 4, 5, and 6.

The state we prepare before applying the inverse quantum Fourier transform can therefore be written as

$$|\tilde{\Psi}_\omega\rangle := \frac{1}{2^n}\sum_{k,l} s_\omega(k,l)\,|k\rangle\,|l\rangle\,|\widetilde{f}_\omega(k,l)\rangle\,,$$

where $s_\omega(k, l) \in \{-1, 1\}$ absorbs (-1)-phase errors. We define the failure set

$$B_\omega = \{(k,l) \in [0,2^n)\times[0,2^n) \mid f_\omega(k,l) \neq \widetilde{f}_\omega(k,l) \vee s_\omega(k,l) = -1\},$$

capturing failures due to elliptic curve exceptional cases, arithmetic approximations, and phase errors. Failure means there is a residual phase (e.g., due to approximate phase-fixes) or the output is wrong in the computational basis (e.g., the gcd computation produced a wrong output). We assume for our analysis that no other errors are introduced such that $f_\omega(k, l) = \widetilde{f}_\omega(k, l)$ and $s_\omega(k, l) = 1$ for $(k, l) \notin B_\omega$.

The random offset masks ω used for our point addition circuits randomize the inputs (k, l) for which a given point addition fails. This allows us to prove that there exists a constant $p_f > 0$ such that for every fixed (k, l), the circuit fails with probability at most p_f over the random masks ω, i.e., we will compute p_f such that for all $(k, l) \in [0, 2^n) \times [0, 2^n)$,

$$\Pr_\omega[(k,l) \in B_\omega] \leq p_f.$$

By introducing a projector Π_S onto the set of measurement outcomes for which postprocessing succeeds pulled back through the inverse QFT, the success probabilities in the ideal and approximate cases are, respectively,

$$p_S = \|\Pi_S\,|\Psi\rangle\,\|^2 \text{ and } \tilde{p}_S = \mathbb{E}_\omega[\|\Pi_S\,|\tilde{\Psi}_\omega\rangle\,\|^2].$$

Given a lower bound $p_{S,lb}$ such that $p_{S,lb} \leq p_S$, we prove (see appendix B 4) that the expected success probability over our random offset masks is at least

$$\widetilde{p}_S \geq \left(\max\{0, \sqrt{p_{S,lb}} - 2p_f\}\right)^2.$$

B. Concrete estimates of the success probability of our implementation

In our implementation, we choose accuracy parameters such as the number of bits after which we truncate carry propagation in the approximate modular reduction [2] in a way that guarantees that with confidence level at least $1 - 2^{-128}$, we have that the probability of failure of the entire sequence of 28 point additions (instead of 32, as in [27]) is at most

$$p_f \leq 0.0335.$$

We arrive at this bound by combining a Monte Carlo-based bound (for the in-place multiplications), which gives rise to the $1 - 2^{-128}$ confidence level, with exactly computed bounds for the remaining components, as detailed in appendix B. The bounds below all inherit this confidence level, but since the noncoverage probability of at most 2^{-128} is very small compared to the uncertainty in other aspects that affect the final resource estimate, we omit it.

To arrive at a bound for the success probability of a single run of Shor's algorithm using an approximate oracle, we may use the success probability lower bound by Mosca [56, p. 58] for an ideal oracle implementation and power-of-two QFTs of

$$p_S \geq p_S^{(\text{Mosca})} = \frac{r-1}{r}\left(\frac{8}{\pi^2}\right)^2 \approx 0.657$$

as a starting point to find that

$$\tilde{p}_S \geq 0.553$$

is a lower bound on the probability of success assuming no logical errors.

While Mosca uses 2 padding qubits for each of the two scalar registers, one extra qubit per register is sufficient, since the order $r < 2^n$ and thus the required phase accuracy of $\frac{1}{2r}$ [56] is achieved using $M = 2^{n+1}$. To match this setting, we increase the size of the table lookup replacing the first windowed point addition to 2^{18} and change the subsequent window boundaries by one scalar bit. Enlarging the first lookup has a small effect on the total runtime (see Table I) that is included in our runtime estimate, and re-indexing does not lead to any change in gate or qubit counts. Classical postprocessing [27, 37] is then used to recover the 48 bits corresponding to the final 3 phase estimation windows that we have removed from the algorithm. It would be possible to remove more than 3 windowed point additions by increasing the amount of classical postprocessing or the number of (parallel) runs of the quantum algorithm [31, 37], but we do not make use of these optimizations here to facilitate comparison to prior work.

As an alternative to the lower bound by Mosca, we may use Ekerå's heuristic success probability estimate [37] as a starting point. Assuming an exact oracle and the postprocessing described in Ref. [37],

$$p_S^{(\text{Ekerå})} = 0.99,$$

which results in an estimate of the algorithmic success probability when using our approximate oracle of

$$\hat{p}_S \approx 0.861.$$

Finally, we take into account the probability of failure due to a logical error, 26.45%, from appendix E 3.

End-to-end success probability estimates: Taking into account the probability of a logical error, our implementation thus succeeds in a single run with probability 40.7% when using Mosca's lower bound and with probability 63.3% using Ekerå's postprocessing and heuristic success probability estimate.

The runtime and success probabilities quoted above are per attempt. In general, one may repeat the algorithm multiple times to increase the overall success probability, or simply run the algorithm on multiple devices in parallel. Assuming independent failures and the 63.3% single-shot success probability estimate, running the application on five identical devices in parallel increases the probability that at least one succeeds from 63.3% to $1-(1-0.633)^5 = 99.3\%$. In other words, one may run the algorithm reliably without increasing the wall-clock runtime by using multiple devices.

Part III
Point addition circuits

The most expensive component of Shor's algorithm for computing elliptic curve discrete logarithms on a curve E is the quantum operation that implements double-scalar multiplication

$$U_f : |k\rangle |l\rangle |\mathcal{O}\rangle \mapsto |k\rangle |l\rangle |[k]P + [l]Q\rangle ,$$

where $k, l \in [0, 2^n)$ are n-bit scalars, $\mathcal{O}$ is the point at infinity, and P and Q are the two elliptic curve points given by the discrete logarithm instance, $Q = [d]P$.

As in previous works [2, 26–28, 35], we implement a scalar multiplication $[k]P$ by repeated point addition, decomposing the scalar k into a windowed binary representation $k = \sum_{i=0}^{L-1} k_i 2^{i \cdot w}$ with window size w, where $L = \lceil n/w \rceil$ and $k_i \in \{0, 1, \ldots, 2^w - 1\}$, i.e.,

$$[k]P = \sum_{i=0}^{L-1} [k_i 2^{i \cdot w}]P.$$

For each windowing index $i \in \{0, 1, \ldots, L-1\}$, we classically precompute a table of 2^w points $[k_i 2^{i \cdot w}]P$ for $k_i \in \{0, 1, \ldots, 2^w - 1\}$. In the quantum scalar multiplication at the i-th window iteration, we load the corresponding table into a quantum register, and then add it to the accumulator register using a point addition circuit. After the scalar multiple $[k]P$ is computed into the accumulator, we proceed in an analogous way with the addition of $[l]Q$, using the corresponding precomputed tables for a windowed decomposition of l.

While the high-level structure of our circuits is identical, we introduce several random offset masks as already described above: we initialize the accumulator point with a classically sampled random non-zero point and we add a classically sampled random offset point to each precomputed table, which changes the entries of our tables to avoid the point at infinity (see appendix B). Both provide sources of randomness that allow us to prove a tighter bound on the success probability.

In this part, we describe the quantum circuits that we use for adding two elliptic curve points. We start with the recent construction by Schrottenloher [2], which combines the register sharing idea from Proos and Zalka [24] with the Dialog representation from Khattar et al. [52], and optimized approximate arithmetic. We make two changes to the construction:

First, we use a different representation of the computation of the binary gcd, in which controlled (modular) additions become conditionally-inverted (modular) additions, which are about half as expensive in our setting [2, 36]. This is similar in spirit to the optimized schoolbook multiplication circuits by Litinski [54].

Second, we implement the modular squaring step using a specialized version of the Litinski multiplier [54] for non-modular squaring and explicit modular reduction by the pseudo-Mersenne prime leveraging a single Karatsuba split [53].

In addition, we make minor optimizations such as using measurement-based uncomputation of temporary predicates. This introduces phase errors in addition to arithmetic errors, and we take this into account when deriving the success probability of the algorithm.

All optimizations combined, we arrive at a windowed elliptic curve point addition circuit that uses approximately $1.196 \cdot 10^6$ Toffoli gates (without the 3 table lookups of size 2^{16}), when targeting a final width of at most that in [2]: 1462 qubits including the 16 windowing qubits. To arrive at these gate counts, we follow the same convention as in previous work [28] to allow for a direct comparison, i.e., we only count actually executed gates and we estimate the Toffoli count of the entire algorithm using the same formula (see below). The Toffoli count above corresponds to the largest observed gate count from 100 runs. With a total of 28 point additions [27, 28, 37], this yields a gate count estimate for an entire run of Shor's algorithm of approximately $28(1.196 \cdot 10^6 + 3 \cdot 2^{16}) \approx 39.0 \cdot 10^6$ Toffoli gates.

V. HIGH-LEVEL IMPLEMENTATION OF POINT ADDITION

We start with an abstract description of the point addition circuit. We follow the same high-level construction as Schrottenloher [2] and implement windowed elliptic curve point addition using three forward table lookups [35], immediately freeing the output registers after consumption using measurement-based uncomputation. We ignore exceptional cases for point addition (showing below that their contribution is negligible), except the case where the classical point to be added is the point at infinity. This case must be handled, especially for small windowing parameters w, since a 2^{-w} fraction of the input table is $\mathcal{O}$. However, instead of adding extra controls to the circuit as done in [2], we choose a single offset point for each classically precomputed table that is added to each point in the table such that none of these points is the point at infinity $\mathcal{O}$. This results in a constant shift by the sum of all table offset points, which can be subtracted at the end. Since a constant shift does not change the outcome of the algorithm [24], in practice, we opt to not remove it.

We consider a single point addition of the accumulator point $A = (x_1, y_1)$ (which we initialize to a random nonzero $A = [a_0]P$ at the beginning) with a table point $T = (x_2, y_2)$. After the table lookup, we compute the coordinate offsets $(\Delta x, \Delta y) = (x_1 - x_2, y_1 - y_2)$ in-place and clear the lookup registers. We compute $\Delta y \mapsto \lambda$, where $\lambda = (\Delta x)^{-1} \Delta y$ using in-place multiplication [2, 52], but using conditionally-inverted additions instead of the controlled additions (or subtractions) used in Schrottenloher's implementation.

We then look up $3x_2$, update $\Delta x \mapsto \Delta x + 3x_2 = x_1 + 2x_2$, and clear the lookup register. We map $x_1 + 2x_2 \mapsto (x_1 + 2x_2 - \lambda^2) = (x_2 - x_3)$ and $\lambda \mapsto \lambda(x_2 - x_3) = (y_3 + y_2)$ using the same implementation of in-place multiplication as above.

We complete the point addition using a third and final table lookup, fixing the coordinate offsets, and uncomputing the table lookup with measurement-based uncomputation, using as input the XOR of all measurement masks from previous X-basis measurements of looked-up coordinates. For a detailed integer-valued representation of our circuits, see Algorithms 4, 5, and 6.

VI. OPTIMIZED BINARY GCD ITERATION

In this section, we describe an optimization that we applied to the implementation of the binary gcd circuit recently published by Schrottenloher [2], which internally uses the Dialog representation from Khattar et al. [52] for the two in-place multiplication steps in the elliptic curve point addition circuit.

For an approximate implementation, one may replace the comparison in the binary gcd by a shorter comparison that only considers $n_{\text{cmp}} < n$ most-significant bits [2] for a failure probability decreasing exponentially with n_{cmp}. As a result, the most expensive components of each iteration (at application scale) are the controlled adders and the controlled register swaps.

We start by recalling the observation that controlled operations are sometimes more expensive than conditional adjoint operations, i.e., $U^c = |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes U$ can be more expensive than $U^{(\dagger)^c} = |0\rangle\langle 0| \otimes U + |1\rangle\langle 1| \otimes U^\dagger$ [54, 58, 59]. This is also the case for adders [54] and we would thus like to replace the controlled adders by conditionally-inverted adders.

To achieve this, we use an alternative representation of the register values: Instead of storing (u, v) during dialog construction and (r, s) during dialog replay, we store $(u, \tilde{v})$ and $(r, \tilde{s})$ with

$$\tilde{v} := v + (1 - a)u \text{ and } \tilde{s} := s - (1 - a)r,$$

where $a := v[0]$ is the parity of v. We also require that u is the odd register and we keep a persistent orientation qubit to store the orientation.

In each iteration, we may compute the next parity bit a_{new} from the current registers using that $\tilde{v}$ and u are always odd by

$$a_{\text{new}} = v_{\text{new}}[0] = \frac{(\tilde{v} - u)}{2}[0] = \tilde{v}[1] \oplus u[1],$$

where the third equality holds because $\tilde{v} - u = v + (1 - a)u - u = v - au = 2v_{\text{new}}$ in the usual representation. A correct iteration must result in $u_{\text{new}} = u$ and

$$\tilde{v}_{\text{new}} = v_{\text{new}} + (1 - a_{\text{new}})u_{\text{new}}.$$

Since $v_{\text{new}} = \frac{(\tilde{v} - u)}{2}$, we may subtract u (and then halve) if $a_{\text{new}} = 1$ (meaning that $\tilde{v}_{\text{new}} = v_{\text{new}}$). If $a_{\text{new}} = 0$, we need $\tilde{v}_{\text{new}} = v_{\text{new}} + u$, so we may add u (and then halve). In summary,

$$\underbrace{v + (1 - a)u}_{\tilde{v}} + (-1)^{a_{\text{new}}} u = \underbrace{v - au}_{2v_{\text{new}}} + \delta_{a_{\text{new}},0} 2u,$$

which can be rewritten as

$$\frac{\tilde{v} + (-1)^{a_{\text{new}}} u}{2} = v_{\text{new}} + (1 - a_{\text{new}})u = \tilde{v}_{\text{new}}.$$

A similar calculation shows that the update for the r, s registers during dialog replay will be

$$\tilde{s}_{\text{new}} = \tilde{s} - (-1)^{a_{\text{new}}} r \pmod p.$$

Therefore, by switching to this representation, we may replace the conditional (modular) adders by conditionally-inverted (modular) adders. For pseudo-Mersenne primes, conditionally-inverted modular adders may be implemented efficiently using conditional one's complement before and after the approximate modular adder described in [2]. One slight modification is necessary for the approximate modular adder that handles the $x + y = p$ case from [2]: In the addition branch, we conditionally swap the values $0 \leftrightarrow p$ using approximate comparisons before invoking the conditionally-inverted adder.

To see why bitwise complement is sufficient for pseudo-Mersenne primes $p = 2^n - c$, note that it acts as a correct modular negation followed by a non-modular shift by $c - 1$, which makes a fraction of $\frac{c-1}{p}$ inputs non-canonical. For canonical inputs, the approximate modular adder works correctly with high probability in our setting (see appendix B), and the final bitwise complement removes the shift again.

After the conditionally-inverted adder of each iteration, we have to also update the orientation of the $u, \tilde{v}$ registers for the next iteration. If v_{new} is odd, i.e., $a_{\text{new}} = 1$, we need to swap if $u > v$. Since $v = \tilde{v}$ in this case, this is equivalent to swapping the registers conditionally on $m_i := a_{\text{new}} \wedge (u_{\text{new}} > \tilde{v}_{\text{new}})$.

VII. MODULAR SQUARING

For the modular squaring step in the elliptic curve point addition, i.e., implementing

$$|x\rangle\,|y\rangle \mapsto |x\rangle\,|(y - x^2) \bmod p\rangle\,,$$

we start with the non-modular, integer schoolbook multiplier from [54], which uses a sequence of conditionally-inverted adders (instead of controlled adders), and derive a squaring-specific version with roughly half the non-Clifford gate cost.

We then use a single Karatsuba split [53] together with explicit approximate modular reductions, making use of the special form of the pseudo-Mersenne prime p. For computing the squares of smaller 128/129-bit integers, we use the optimized (non-modular) squarer.

A. Optimized squaring

Given an n-qubit input register $|x\rangle$ and a $2n$-qubit output register $|z=0\rangle$, the goal is to map $|x\rangle|z=0\rangle \mapsto |x\rangle|x^2\rangle$.

We start by writing the square of $x = \sum_{i=0}^{n-1} x_i 2^i$ as

$$\begin{aligned} x^2 &= \sum_{i=0}^{n-1} x_i 2^i \sum_{j=0}^{n-1} x_j 2^j \\ &= \sum_i 2^{2i} x_i + \sum_{i=0}^{n-1} \sum_{j>i} 2^{i+j+1} x_i x_j \\ &= \sum_i 2^{2i} x_i + \sum_{i=0}^{n-1} 2x_i 2^{2i+1} \underbrace{\sum_{j>i} 2^{j-i-1} x_j}_{=:h_i}. \end{aligned}$$

Since we would like to use conditionally-inverted adders instead of controlled adders, we need to bring this into a form prefactored by $(2x_i - 1)$ instead of $2x_i$, so

$$x^2 = \sum_i 2^{2i} x_i + \sum_{i=0}^{n-1} (2x_i - 1) 2^{2i+1} h_i + \sum_{i=0}^{n-1} 2^{2i+1} h_i.$$

The final sum in the expression above can be rewritten as

$$\sum_{i=0}^{n-1} \sum_{j>i} 2^{i+j} x_j = \sum_{j=0}^{n-1} x_j \sum_{i=0}^{j-1} 2^{i+j} = \sum_{j=0}^{n-1} x_j (2^{2j} - 2^j).$$

Substituting this back in and folding the result into the middle sum yields

$$x^2 = -x + \sum_{i=0}^{n-1} (2x_i - 1) 2^{2i+1} (h_i + x_i),$$

since $(2x_i - 1)x_i = x_i$ for $x_i \in \{0, 1\}$. From this expression, we see that squaring can be implemented using an initial $(n+1)$-bit subtraction of x followed by a sequence of n operations that add $h_i + 1$ if $x_i = 1$ and subtract h_i if $x_i = 0$. Starting with the least-significant bit allows us to keep these operations shorter than the full $2n$ output register. Instead, these operations modify a sliding window of length $n+1, n, ..., 2$ moving the higher end of the window by one position toward the most-significant bit in each step.

We implement these operations as conditionally-inverted additions, similar to [54], but instead of inverting the output, we bit-invert the input h_i (padded to match the length of the active output window) conditionally on x_i, mapping

$$h_i \mapsto 2^{n+1-i} - 1 - h_i$$

if $x_i = 1$. We then perform a subtraction unconditionally, before undoing the conditional inversion of the input register. As a result, the needed +1-offset for the $x_i = 1$ case is handled automatically and does not need additional non-Clifford gates.

Before each of the first $(n-1)$ operations computing the sum above, we sign-extend the product register using a CNOT gate (from index $n+i$ to $n+i+1$). The final step $i = (n-1)$ can be implemented directly using a single CNOT gate.

Assuming we use the adder from [36], which needs $w-1$ Toffoli gates to add two w-qubit numbers, the Toffoli count of this squarer is

$$T(n) = n + \sum_{i=0}^{n-2} (n-i) = \frac{1}{2} n(n+3) - 1,$$

saving approximately 50% of the non-Clifford gates compared to a direct use of the multiplier from [54].

B. Modular square subtraction

To realize the operation $|x\rangle|y\rangle \mapsto |x\rangle|(y - x^2) \bmod p\rangle$, our circuit uses one Karatsuba decomposition as follows. We treat input values representing n-bit registers as integers less than 2^n that are not necessarily canonical representatives less than p. Also intermediate results are represented by integer values of their respective bit sizes.

Let n be even and $B = 2^{n/2}$. We use the notation $x_{[t,t+\ell)} = \lfloor x/2^t \rfloor \bmod 2^\ell$ to denote the ℓ-bit slice of the non-negative integer x starting with bit t. Write $x_L = x_{[0,n/2)}$ and $x_H = x_{[n/2,n)}$ for the integers representing the low bits and high bits of x. If $x < 2^n$, then $x = x_L + B x_H$, with $0 \le x_L, x_H < B$. Since $B^2 = 2^n \equiv c \pmod p$, we have

$$\begin{aligned} -x^2 &= -B^2 x_H^2 - 2B x_H x_L - x_L^2 \\ &\equiv (B-c) x_H^2 + (B-1) x_L^2 \\ &\quad - B(x_H + x_L)^2 \pmod p. \end{aligned}$$

Using the squaring operation from the previous section, we then compute the exact integer square x_L^2 in an n-qubit auxiliary register, then add $B x_L^2$ to y modulo p, subtract x_L^2, and uncompute the register. This yields $y + (B-1)x_L^2 \bmod p$ in the y-register. The analogous steps for x_H add $(B-c)x_H^2$. After that, we compute the $(n/2+1)$-bit sum $x_L + x_H$ in place, square it into an $(n+2)$-qubit scratch space, and subtract $B(x_L + x_H)^2$ from y modulo p. We then uncompute the square and restore x_H by subtracting x_L.

Algorithm 7 in Appendix A gives the integer-level operation order in detail. It uses the components $\mathsf{AddSlice}^q_\sigma$, which invokes the phase correction $\mathsf{PhaseGE}^q$, as well as $\mathsf{AddBMultiple}^q_\pm$ and $\mathsf{AddCMultiple}^q_-$. These operations are given in Algorithms 8, 9, 10, and 11 in Appendix A.

Part IV
Compilation toolchain

VIII. COMPILER OVERVIEW

In this part, we describe our logical compiler, which, for a given quantum algorithm, synthesizes quantum circuits for the target logical instruction set architecture (ISA). Specifically, we compile the elliptic-curve circuits of Part III to the specialized Walking Cat architecture of Part V.

For our fault-tolerant architecture, the allowed instruction set consists of logical Pauli measurements, tracked Clifford updates, and magic-state consumption. From these instructions, the compiler produces an executable, architecture-legal measurement schedule with exact resource accounting. We separately generate measurement schedules for each algorithm-level component and compose them through end-to-end integrated routing at 5% overhead. Table I provides a total cost breakdown per component. Lowering to native operations on the system graph (transport, physical gates, and readout) happens below the compiler, at runtime, using the library of ISA primitives.

Optimally compiling to such an ISA becomes intractable at the scale of hundreds of logical qubits, because the target algorithms inflate to millions of operations, and implementation choices compound while having to adhere to architectural constraints. Compilers that flatten the algorithm into a gate-level netlist discard the structure this optimization requires, leaving only local rewrites, while high-level resource-estimation frameworks preserve the structure but do not lower it to executable instructions. We bridge this gap with a hierarchical approach built from reusable compilation primitives that are selected, lowered, and scheduled with full knowledge of the target architecture.

The compiler uses the hierarchical instruction model of Ref. [1], similar to those of Qualtran [60, 61], QREF [62, 63], and ProjectQ [64], as well as the classical hierarchical compilation framework MLIR [65]. Within this model, the architecture-based library of operations can be extended with higher-level logical components. Each component has a *spec* defining its semantic inputs, outputs (its *ports*), and parameters, and one or more *defs* describing decompositions into other smaller component specs. Low-level gates are components that are legal on the architecture, and high-level algorithmic structures like square-subtract are comprised of many abstraction levels of progressively smaller defs and specs. Any component may be conditioned on a classical outcome. Defs declare their own ancillae, so alternative implementations can expose different depth, width, non-Clifford-cost, and layout tradeoffs without changing the inputs and outputs of the component. This system lets the compiler select among reusable and hand-optimized implementations according to the architecture and the surrounding circuit context as it recursively lowers the root component into a synthesis tree [1].

The rest of this part follows the following structure. Sec. IX describes the architectural constraints we compile against, and the scheduling routine that enforces them. Sec. X describes layouts of algorithmic components and the integrated routing that connects them. Sec. XI describes how we prove defs realize their advertised spec using independent "contracts", and how compiled schedules are checked against resource and architecture constraints.

IX. ARCHITECTURAL CONSTRAINTS AND SCHEDULING

Specs and defs are stateless descriptions of the component hierarchy: they record what a component does, how it can be decomposed, and its semantic action on its qubits (like returning any declared ancilla to $|0\rangle$) but not the assignments of logical qubits to algorithmic qubits, binding of magic states to factories, or whether a def's child specs must be decomposed further on the architecture. That state is tracked during circuit synthesis, where we bind qubits and resources, enforce architectural constraints, and schedule operations.

A. Architectural constraints

Our architecture, described in Part V, provides the following capabilities and constraints to our compiler:

1. Memory blocks use the Q102 code and provide 22 logical qubits.

2. Entangling Cliffords within a memory block may be executed via frame tracking. Empirically validated for our structured arithmetic circuits and discussed in Sec. XII D 3.

3. Each memory block can participate in up to three logical measurements meeting compatibility conditions. As above, validated and discussed in Secs. XII D 1 and XII D 3.

4. The processor can perform a single CCZ injection per layer, with factory-side cleanup taking place the following layer. The lone exception, the CCZ gates injected during lookup tables, have consequences discussed in Sec. XII D 3.

5. The processor provides 24 memory-side cat bundle pairs.

6. The processor provides 69 memory blocks.

7. A structured logical CliNR sweep performs logical frame clearing, chasing after structured serial arithmetic (discussed further in Sec. XII D 2).

B. Scheduling

As the compiler decomposes the root component into a tree, it ensures that we meet both memory block capacity and leaf operation legality constraints. We then emit the decomposed tree serially into the measurement scheduling routine, which greedily packs measurements into layers, with each layer followed by frame-tracked Clifford updates. The scheduler enforces per-block parallelism limits, schedule-time measurement compatibility conditions, CCZ injection capacity, and global memory-side cat bundle parallelism over the course of the complex scheduling procedure we describe below.

Algorithm 1: Measurement scheduling routine.
Notation: For a measurement M, `floor` is M's causal lower bound; `seed` is its viable starting placement; i is the layer under examination for placement; `best` is the earliest viable placement layer found; and CanPlace(M, i) tests architecture constraints: injection count, cat bundle parallelism, block capacity, and disjoint measurement compatibility.

```
foreach leaf emitted from tree do
    if (leaf is a measurement M) then
        Find floor and seed for M
        (best, i) ← (seed, seed)
        while (i > floor) do
            Propagate and transform M through prior Cliffords
            if (a conditional Clifford changes the basis of M) then
                break
            else if (M crosses an anticommuting conditional Pauli) then
                XOR that condition to its virtual meaning
            if (M anticommutes with a measurement in layer i) then
                break
            if (parallel injection measurements stop being compatible) then
                break
            if (CanPlace(M, i)) then
                best ← i
            i ← i − 1
        Place M in best
    else if (leaf is a Clifford) then
        if (it is conditioned on virtual measurements) then
            Resolve virtual measurements
        Place it
```

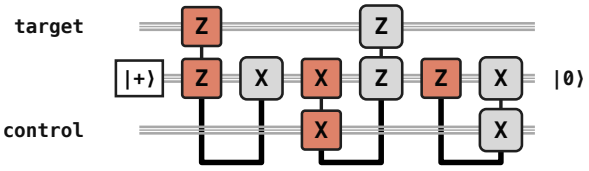


FIG. 2: An arbitrary Pauli-controlled-Pauli (PcP) gate can be decomposed using three measurement layers and an ancilla, as shown for a CNOT gate. The first measurement is akin to copying the first qubit onto a clean ancilla, providing the intuition for our measurement-based integrated routing operations.

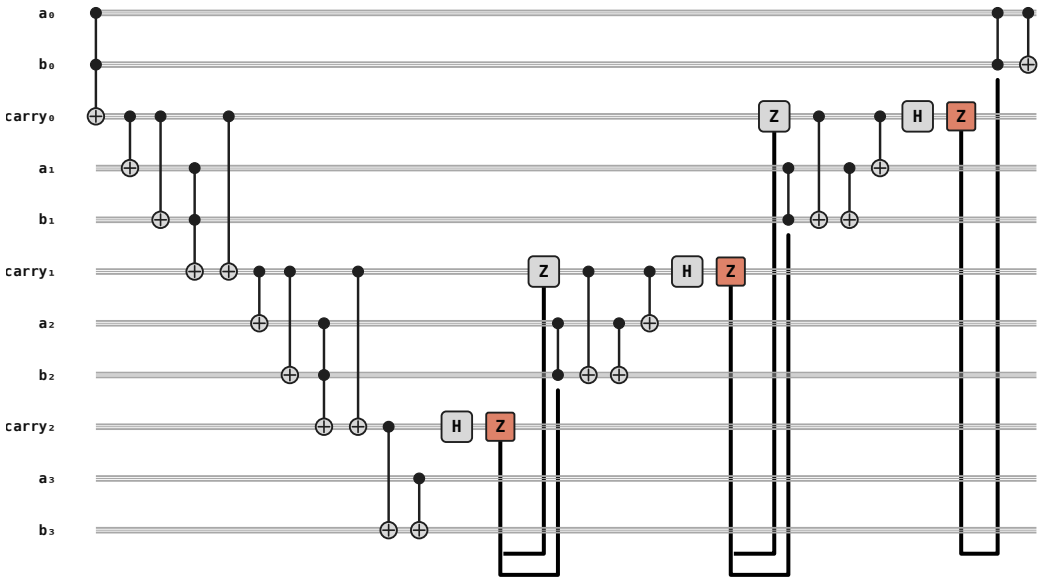
X. ALGORITHMIC COMPONENTS AND INTEGRATED ROUTING

We leverage layout plans (appendix D 5) to provide high-level strategic reasoning to the compiler. The layout plan suggests optimized mappings of algorithmic to logical qubits and may optionally force the decomposition of a spec into a chosen def. Each algorithmic component is compiled and optimized for its own authored layout plan designed to reduce the measurement-layer depth on the target architecture. The algorithm, however, may have to execute algorithmic components with incompatible layouts in sequence, and indeed most algorithmic components have consecutive child components with incompatible layouts. Connecting them efficiently is the integrated-routing problem addressed in this section.

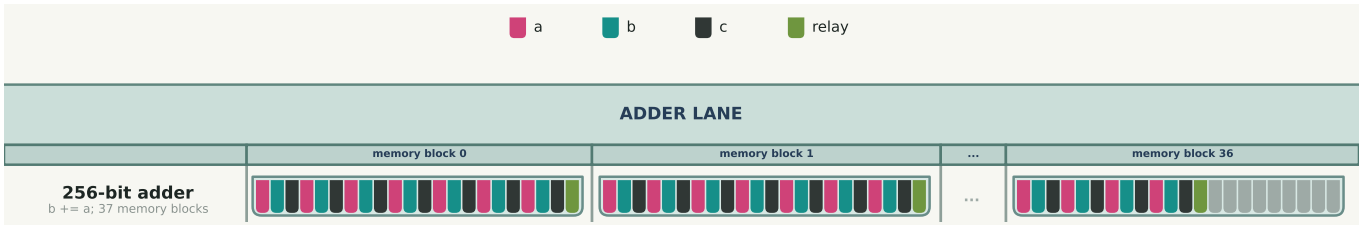
A. Routing an adder

A component's layout maps the component's ports to qubits in one or multiple memory blocks. We can minimize the measurement depth required to execute a component by leveraging entangling Clifford tracking within memory blocks and exploiting parallelism between different blocks. To show why this is powerful, consider the traditional decomposition of an entangling Clifford, or a Pauli-controlled-Pauli gate (see Fig. 2). This protocol requires three measurements and an ancilla to form the link between qubits. Through careful routing, we manage to avoid this decomposition for every single CNOT or CZ gate in our algorithm. Fig. 3 shows an adder with its natural layout to demonstrate how our architecture can best support a simple structured def.

The addition of two quantum registers is at the core of the arithmetic of Part III. The adder specification declares two n-qubit semantic registers $|x\rangle$ and $|y\rangle$ and adds the first into the second modulo 2^n. Distinct defs realize that specification with different resource trade-offs: a ripple adder in the style of Cuccaro et al. [29], a carry-lookahead adder [66], a Gid-



(a) Quantum circuit of a 4-bit Gidney adder with measurement-based uncomputes. Note that entangling Cliffords either act on pairs of qubits a_i and b_i or link c_i to a_{i+1} and b_{i+1}, and Toffoli gates similarly act on triples of a_i, b_i, and c_i.



(b) Layout plan for the Gidney adder. We designate a run of successive memory blocks as an adder lane, where we interleave a_i, b_i, and $carry_i$. This fits as seven triples per memory block, with the 22nd qubit serving as a hole for relaying data cheaply when a carry c_i is needed for CNOT gates targeting qubits in the next block.

FIG. 3: Gidney adder layout and circuit.

ney temporary-AND adder [36], an ancilla-free Takahashi adder [30], a logarithmic-depth Remaud–Vandaele adder [67], and hybrids between them. Controlled and constant-operand adders live alongside as their own families with different semantic specs.

For simplicity, routing tricks, and consistency of our logical frame analysis, we leverage Gidney-style ripple-carry adders, even for constant adders (often at little or no cost, as in Fig. 4). An optimized layout of a Gidney adder is shown in Fig. 3b: packing each x_i, y_i, and $carry_i$ in successive qubits places the classically controlled CZ gates from both CCZ injections and the measurement-based uncomputation (MBU) of the carries during the rippling uncompute section of the adder within memory blocks, allowing them to be executed in software via frame tracking. Similarly, nearly all CNOT gates in the adder land within a single memory block and are naturally executable via frame tracking. The remaining cross-block CNOT gates are mitigated by integrated routing as follows: whenever a given $carry_i$ controls CNOT gates that would cross memory blocks, we leverage the next memory block's 22nd qubit to quickly copy $carry_i$ forward. The surprisingly lossless nature of this additional measurement is discussed further at the end of the next section.

B. Integrated routing

While an individual low-level component may be mapped to memory blocks in a natural way to maximally exploit entangling Clifford tracking, satisfying the natural layouts of consecutive components without stalling the serial execution of the overall algorithm is highly nontrivial. We develop integrated routing to address this challenge—in particular, we leverage the serial rippling nature of the arithmetic and architectural-level measurement parallelism to hide the cost of logically moving data between memory blocks.

There are a few basic data movement operations, each

decomposed into a low number of measurements:

- **Copy**: Create an entangled logical basis copy of a qubit with a single cross-block ZZ joint measurement.
- **Fanout**: Create many copies of a qubit across blocks by creating bell pairs with a layer of cross-block XX joint measurements and entangling them with the original with a layer of ZZ joint measurements, forming a two-layer measurement brickwork.
- **MBU**: Measurement-based uncomputation clears a qubit by measuring it in the X basis and applying a conditional Z Pauli correction to the qubit's source.
- **Move**: Move a logical basis qubit from one block to another by making a logical basis copy and MBUing the original; cf. Fig. 12 of Ref. [68].
- **Swap**: Swap two qubits through one or more temp qubits by moving qubits in a sequence. If we use temps that are already local, a swap may be thought of as two parallel moves plus locally-tracked Clifford swaps.

First, because the compiler can propagate conditional Paulis through measurements with classical bookkeeping, the cost of an MBU from a **Move** is often hidden and the cost of our movement operations is typically just the number of serial cross-block measurement layers. Second, because we may perform up to three measurements per block in parallel, we can reconfigure memory blocks for the next component as the rippling arithmetic of the prior component is collapsing back to the low qubits. Third, logical CliNR (see Sec. XII D 2) is performing this same rippling sweep and may be leveraged to both incorporate local swaps and guarantee that the moves are cheaply done with empty logical frames. Fourth, logical CliNR incurs short stalls of approximately 7.73 layers at certain critical points between components, which we already account for in our costs reported in Table I and which give additional time for routing moves to complete. Altogether, integrated routing allows us to push the layer depth of an algorithmic component very close to the serial layer depth of the rippling arithmetic (typically, the number of Toffolis and their associated rippling measurement-based uncomputations).

C. Routing the approximate modular adder

Here, we show a worked example of a modular adder, demonstrating how we leverage integrated routing to approach the serial lower bound. We leverage an approximate modular adder similar to that of [2] to compute $|x, y\rangle \to |x, y + x \bmod p\rangle$, whose def has the following sequence of operations:

Algorithm 2: Phase-approximate modular adder

1. Compute $|x, y\rangle \to |x, x + y\rangle$ with a 256-bit Gidney-style carry out adder.
2. Unload the low 65 bits of x into the work qubits in the middle of the adder.
3. Fan the carry out qubit onto the low-region relay qubits and use it as the control for a controlled load of the pseudo-Mersenne modular correction c.
4. MBU the block-local copy of each carry out qubit to make room for the Gidney adder's block-local relay qubit.
5. Use a 65-bit Gidney-style register adder to compute $|c, x + y\rangle \to |c, x + y + c \bmod 2^{65}\rangle$.
6. MBU the local copy of c.
7. Return the low bits of x from the packed middle region.
8. Run a phase comparator on the high 32 bits of x and y to correct the phase of all MBUs.

The exact details of this sequence are shown in Fig. 4. Now, the serial lower bound of the rippling arithmetic of the adder is given by twice the ripple depth of the 256-bit adder, plus twice the ripple depth of the 65-bit adder, plus the ripple depth of the phase comparator. Note that we parallelize half of the phase comparator against the closing ripple of the low-width constant adder. That gives a total lower bound of approximately $510 + 128 + 31 = 669$, and our phase approximate mod adder compiles to a layer depth of 674. The logical CliNR penalty adds approximately 7.73 layers to the seam between the 256-bit adder and the low-width constant adder (as discussed in Sec. XII D 2), causing enough delay to fully hide the five-layer routing overhead. This demonstrates the power of integrated routing and why leveraging Gidney-style register arithmetic even for constant adders is nigh lossless in measurement depth. Our measurement scheduler also demonstrates its capabilities in this example—the scheduling routine achieves this depth, even while needing to relay data between memory blocks, by parallelizing every relay alongside the very first CCZ injection into a later block. While the missing relay data in a given block prevents us from interpreting the results of the injection measurements, applying tracked conditional CZ gates, and then injecting the next CCZ gate, we may still free up the factory resource. Then only the relay between the final two memory blocks increases the serial depth of the adder. Our scheduler finds this optimization automatically.

D. End-to-end integrated routing for algorithmic components

Here we give a brief description of the integrated routing strategy internal to each algorithmic component documented in Table I, excluding the phase-approximate

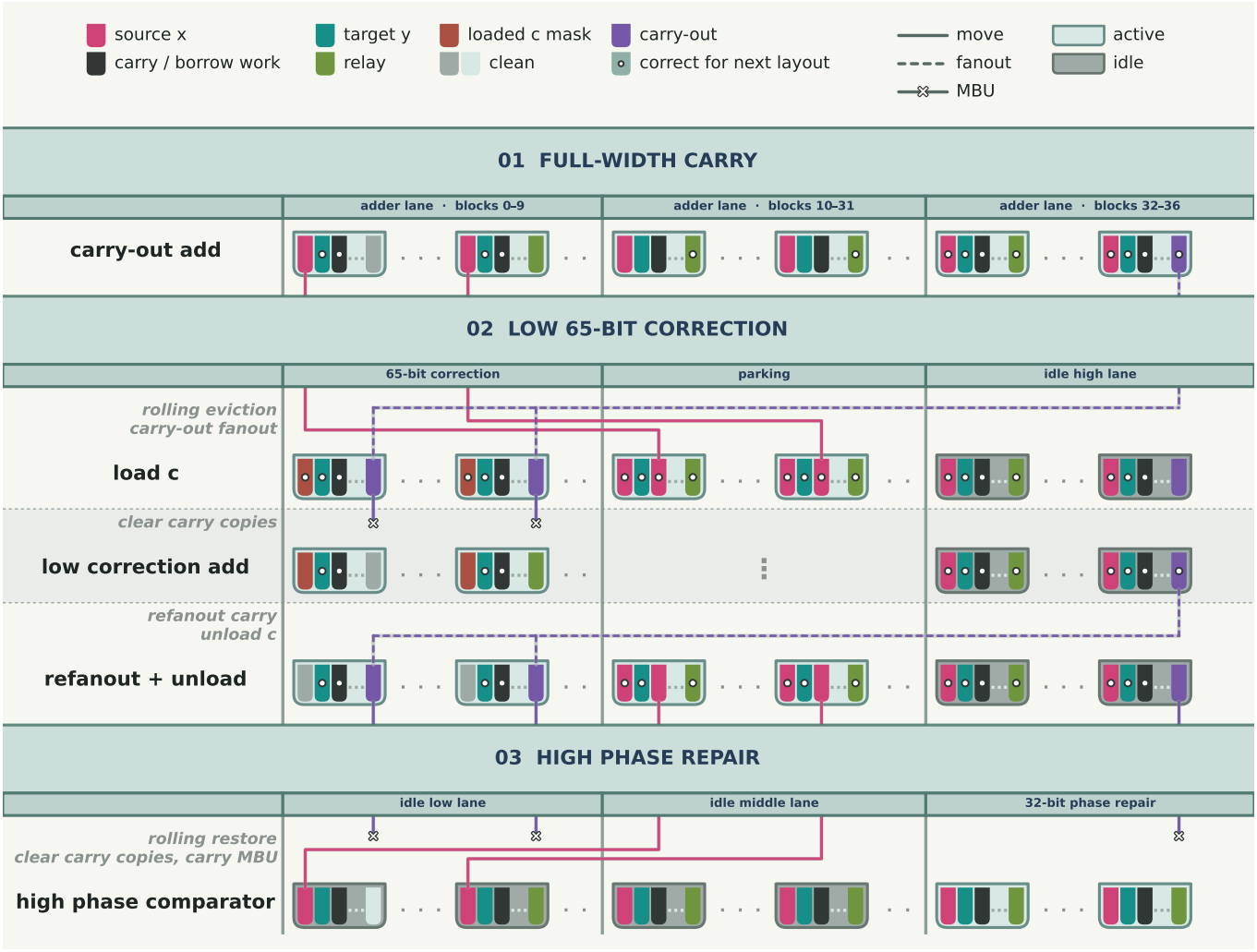


FIG. 4: Routing of the phase-approximate modular adder as described in algorithm 2.

modular adder.

Lookup. Lookups consist of a depth-first traversal over the index bits, using scratch qubits to help with partial answers. Putting scratch qubits and the lowest index qubit in the same block keeps all traversal CNOT gates block-local. To do the same for CCZ injections, we cycle copies of relevant index qubits into and out of the block with **copy** and **MBU** operations. We pack the controlled mask-load CNOT gates into blocks by leveraging **fanout** and **MBU** to relay each control bit to the load target blocks in turn. We further split the $k = 16$ lookup into two parallel $k = 15$ lookups, following the zipper construction of Ref. [68], with copies of the index bits and a second relay qubit in each target block.

Approximate modular subtraction. This works similarly to the modular adders, except we perform a modular complement before and after the adder. Modular complement is done by bitwise inversion using frame-tracked X gates and unconditional low-width constant addition of $2^{65} - c$, where c is the pseudo-Mersenne modular correction. We again pack and restore the low 65 bits of x using the middle of the adder around each constant correction.

In-place multiplication. Our implementation of the in-place multiplication [2, 52] uses Gidney adders [36] for the arithmetic on the u and v registers, which we place in the adder lane, and computes history bits nearby. The adder lane shrinks as we reduce u and v, freeing up more qubits that we slowly load y into. We compress each set of six local history qubits into five [2] using a dedicated compute zone, and we pack compressed history qubits into long-term storage. When u and v are reduced away, we unpack the rest of the y register into the adder lane, play back the history to perform in-place multiplication or division, and then repack y while working back up the $u - v$ sequence to uncompute the history. This is the widest portion of the algorithm—at its peak, we have 68 active memory blocks, plus one for the algorithm's 16 idling index bits.

Square-subtract. Our square-subtract routine lever-

ages a single-level Karatsuba split, as described in Sec. VII. The routine accumulates squares of parts of a register y into a product register, then adds different alignments of the squared terms into the real accumulator x before uncomputing each square. We again leverage Gidney-based arithmetic—our central squaring routine repeatedly shifts and performs conditionally-inverted addition. After each square of a 128-bit portion of y (or the sum of the high and low halves of y), the product register has been shifted nearly 128 times, bringing it nearly in alignment to represent the shifted value $B \cdot \text{product}$. We may then exchange y and x to accumulate $B \cdot \text{product}$ and any constant multiples $c \cdot \text{product}$, realized as shifted adds or subtractions of product at different offsets.

Approximate modular negation. Approximate modular negation is nearly the usual approximate modular complement, with one correction—we have a rippling chain of Toffolis, with a similar structure to an adder, used to detect the edge case $x = p$ and replace p with 0 [2]. Though the basic routing details are covered in prior components, we mention that we use this routine to avoid the traditional exact modular arithmetic used to finalize a round of windowed point addition. This preserves our maximum circuit width while still allowing us to leverage Gidney-style register addition throughout the entire circuit.

Integrated Routing for windowed point addition. We compose integrated routing for an entire window of the point addition routine—the total integrated routing penalty per window is nearly **zero**. We compile each of the major algorithmic components separately and add their costs together, adding a conservative cost of 20 layers per window. Moreover, layer depth reductions from compressing the algorithmic components across their seams would far outweigh any penalty from integrated routing. Our routing strategy is best described visually in Fig. 21 within the Appendix.

Of note, because we unconditionally execute all phase comparators, we have no conditionally-executed measurements to account for, outside of our iQFT implementation (Sec. XII E 4). Because we parallelize the first half of each phase comparator against other rippling serial arithmetic elements with the strategy shown in Fig. 4, we achieve the same average cost as conditional execution.

XI. VERIFICATION

Trusting the definition and compilation of fault-tolerant-scale circuits is a hard problem. We address it by splitting the problem in two halves: first, prove that the top-level component is decomposed into a correct tree, and second, prove that the scheduler converts the tree to a valid program executable on our architecture.

A. Hierarchical verification of the tree

The only operation that synthesis performs on the component tree is replacing some spec with one of its defs. This leads to a powerful insight—by verifying that every single def implements its spec, we can trust every operation the tree performs, and thereby trust the final tree's correctness. To do this, we leverage another key insight: many components may be described without introducing genuine quantum behavior like superposition or entanglement. In particular, for our circuits, nearly every component is a quantum implementation of a reversible classical circuit, enabling classical emulation as in Refs. [64, 69].

For every spec, we specify a *contract*: a machine-checkable reference for the semantic action on the spec's ports. Implementation details like ancillae do not appear in the contract. These contracts take the form of functions defined on some declared domain and come in three types of escalating complexity:

1. **Basis contract**: the spec maps computational basis states to computational basis states. Reversible classical circuits fall into this category, including CNOT gates and Toffolis. Contracts of this type are nearly trivial to verify classically.

2. **Basis-plus-phase contract**: a two-dimensional variant of a basis contract. Now, alongside the basis mapping, the spec may also apply a phase function to any given basis state. The common occurrence in our circuits is MBUs and their associated phase correction components.

3. **Operator contract**: a full operator, for components that create superpositions or entanglement. All measurement-based protocols eventually fall into this category, but the overall action of a small component relying on measurements is almost always in the prior two categories.

For each def, the verifier expands its immediate children, including its structured control flow, and composes their contracts in execution order. It maps each comparison input, together with the implementation ancillae, into the def's internal register layout, evaluates the child composition, projects the result back onto the semantic ports, and compares it with the spec contract. Clean ancillae must be returned to $|0\rangle$. Child calls are also required to satisfy their own declared input domains. If a spec has a basis contract but a given def has children with a basis-plus-phase contract, verification proves that the phase functions cancel, reducing to a basis contract with an identity phase function.

For the reversible arithmetic that dominates the ECC pipeline, the contract is simply a function on basis values. The adder specification's contract, for example, is $(x, y) \mapsto (x, x + y \bmod 2^n)$. By representing all measurement-based protocols with simple basis contracts

(e.g., **move**, **swap**, **copy**, etc.), and MBUs and CCZ injections as basis-plus-phase contracts, we avoid opening defs whose children have operator contracts directly. We independently certify that each small measurement-based primitive performs the correct action.

The verification routine runs either exhaustively for components with a small input domain, or randomly sampled over components with a large input domain, including approximate components. For a two-bit adder def, for instance, the verifier sweeps all sixteen (x, y) basis inputs with the def's scratch at $|0\rangle$, evaluates the composition of the children's own contract functions, and requires the output $(x,\ x+y \bmod 4)$ with the scratch returned to $|0\rangle$. In contrast, at the massive scale of our application, we randomly sample input states. The approximate modular adder for the secp256k1 field, whose modulus is $p = 2^{256} - 2^{32} - 977$, is checked on 100,000 reproducibly sampled inputs. Such sampling is both strong evidence of our implementation's accuracy over all $\sim 2^{512}$ input pairs as well as affirmation that our approximate arithmetic holds at scale. We verify both large- and small-scale versions of all components.

B. Scheduling verification

We prove our schedules accurately preserve the tree's semantics with a combination of different checks and unit tests, and we perform these checks at multiple levels of abstraction and record their results with the compiled data.

1. We have a variety of tests for the pipeline itself, including but not limited to compiling Shor's integer factoring algorithm for 3- and 5-bit numbers to executable programs and simulating the resulting programs exactly.

2. We verify that the compiler's schedule is *physical*, that is, conforms to the set of constraints imposed by the architecture, including cat bundle consumption and magic-state injection limits.

3. We bound resource figures with analytic envelopes derived from the algorithm; in a Gidney-style adder, for instance, we expect two measurement layers (appendix D) per Toffoli (to be exact, given our block-aware decomposition, $2n-1$ layers, where n is the number of bits in the adder). The envelope imposes a floor as well as a ceiling, both to assert that information-theoretic bounds are respected and to ensure we do not unknowingly introduce a performance regression.

4. We ensure our routing applies correctly and we avoid naive decomposition of any cross-block entangling Cliffords.

5. We run small-scale component-level simulations of the schedule, ensuring that our final programs preserve circuit semantics. This includes simulations of the 674-layer schedule of the 256-bit approximate modular adder discussed in Sec. X with computational basis state inputs, as well as phase-aware simulations with superpositions of inputs at reduced scale.

Every reported figure is produced by a reproducible pipeline. A run is declared in a suite file that fixes all inputs: the entry-point component and its parameters, the architecture, the layout plan, and the verification checks to apply. Executing the declaration writes a run directory holding the compiled result, its measurement schedule, the recorded outcomes of the checks described above, and a manifest of the resolved arguments, including the serialized architecture, the git commit of every repository involved, and the execution environment. Runs are identified by a content hash over the declared inputs: two executions of the same declaration on the same code hash identically, so any reported number can be traced to the configuration that produced it. Because the measurement schedule is retained alongside the result, a reported depth can be re-examined without recompiling.

Part V
Architecture

XII. ARCHITECTURE OVERVIEW

A. The secp256k1 Device

We consider the secp256k1 device, based on a modified version of the Walking Cat architecture optimized for solving a secp256k1 discrete-logarithm instance. Because the elliptic curve point addition circuits developed in this work account for most of the algorithm's logical operations, we tailor the architecture to their resource requirements rather than to those of other stages, such as the quantum Fourier transform. The resulting architecture enables execution of an end-to-end secp256k1 discrete-logarithm instance with the resource totals summarized in Table II.

TABLE II: Resource totals for solving secp256k1 with the secp256k1 device.

Metric	Value
Physical-qubit footprint	19,397
Runtime per attempt	$\sim$ 26 days
Logical failure probability per attempt	$\sim$ 26%

The device contains 69 Q102 memory blocks, 24 cat-state bundle pairs, 12 Bell-state bundles for LM2 measurements, four logical CliNR blocks, and four magic factories, each with two local six-qubit C6 Bell-state sources. Including local and global reloading reservoirs, these components give the 19,397-physical-qubit footprint summarized in Table III. Details and justification for runtime, logical failure probability, and the physical-footprint total are provided in appendix E 2, appendix E 3, and appendix E 1, respectively.

TABLE III: Component allocations and physical-qubit footprint of the secp256k1 device.

Component	Count	Qubits per component	Qubits in allocation
Memory blocks	69	207	14,283
Cat-state bundle pairs	24	120	2,880
Bell-state bundles	12	8	96
Logical CliNR blocks	4	207	828
Magic factories	4	319	1,276
Reloading-reservoir qubits	34	1	34
Physical-qubit footprint			19,397

The architecture uses the three-ring QEC codes summarized below to support its memory and magic-state factory blocks. The Q102 code is from the Walking Cat Architecture paper [1], Q66 is a new generalized bicycle code introduced in this work, and C6 is Knill's $[\![6, 2, 2]\!]$ code [70]. C6 can be expressed as a generalized bicycle code, making it a three-ring code, and therefore compatible with the Walking Cat architecture.

TABLE IV: Three-ring QEC codes used in the secp256k1 device. Logical error rates are quoted per SEC for Q102 and Q66, and per accepted encoded $|H\rangle^{\otimes 2}$ pair for C6. Logical error rates are estimated for the moving qubits model with physical error rate $p = 10^{-4}$ including transport noise, leakage and loss.

Code	Parameters	SEC depth (POCs)	Logical error rate
Q102	$[\![102, 22, 9]\!]$	27.0	9.34×10^{-12}
Q66	$[\![66, 4, 10]\!]$	17.0	2.63×10^{-11}
C6	$[\![6, 2, 2]\!]$	5.4	1.30×10^{-8}

The secp256k1 device specializes the Walking Cat architecture around the resource demands of the elliptic curve point addition circuits. In particular, we prioritize reducing its physical footprint while retaining enough measurement and magic-state throughput to avoid introducing new runtime bottlenecks. This specialization rests on four changes to the baseline architecture.

First, we introduce the CLAW ISA, an extension of the Walking Cat architecture ISA [1] that addresses bottlenecks in elliptic curve point addition circuits, as detailed in Sec. XII D. By studying the compiled logical measurements, we find that each has an accessible physical representative in the Q102 code even when every Clifford gate within a block is tracked virtually. Consequently, Clifford gates account for less than 10% of our final logical-measurement count. Because Toffoli gates account for the bulk of the remaining measurements, we optimize their execution with a new parallel logical-measurement operation, an idle-frame-clearing scheme, and a depth-one CCZ gate.

We introduce the lightweight Eastinthillation factory in Sec. XII E, which supports the depth-one CCZ gate.

Then, we adapt cat-state distribution to the elliptic curve point addition circuits' limited global logical-measurement parallelism. Rather than permanently provisioning every code block with both sets of cat states and verification ancillas, the device moves the cat states required for logical measurements across the device as detailed in Sec. XII C.

Finally, in Sec. XII B, we replace the baseline architecture's qubit-loss model [1] with the swap-loss model, in which an interaction can move a loss but cannot spread it. This model relaxes the hardware requirements associated with qubit loss: an implementation of the secp256k1 device need neither guarantee which output retains the surviving qubit nor ensure that an attempted interaction with a missing qubit ejects or disables its partner. Preventing a single loss from branching also permits lower-overhead loss-detection protocols than the beacon proto-

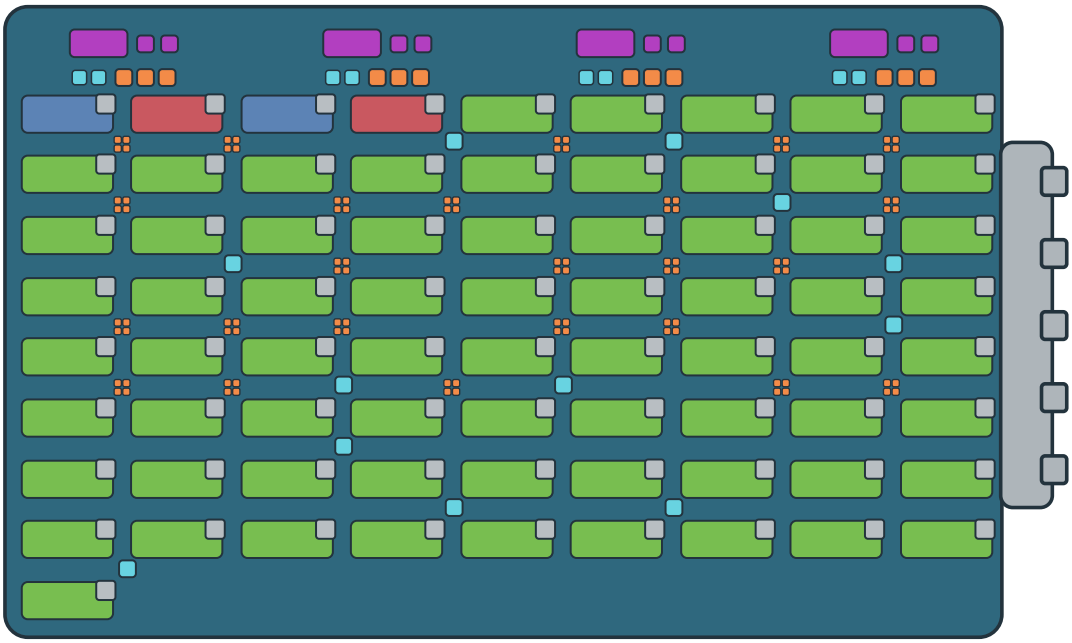


FIG. 5: Full-scale `secp256k1` device. Purple blocks are magic-state factories required for CCZ state generation, green blocks are memory blocks, orange blocks are cat-state factories, and blue blocks are Bell-state bundles for LM2 measurements. Each group of four orange blocks forms a mobile cat-state bundle pair. The two cyan sites inside each magic-factory group are local six-qubit C6 Bell-state sources. The cobalt and crimson blocks are reserved for two Logical CliNR instances used for Clifford-frame clearing. The small gray box attached to each code block is its local ion reservoir, while the full loading reservoir runs along the right edge of the device.

col developed in [1]. For a code block with n data qubits, these new protocols reduce the physical-qubit count from $3n$ under the beacon protocol to $2n$, thereby reducing the footprint of every memory and factory block in the device.

B. Swap-Loss Model

We replace the loss-propagation rule in the moving-qubit noise model of the Walking Cat architecture [1] with the *swap-loss model*. In this model, a two-qubit gate involving exactly one lost qubit can exchange the locations of the loss and the surviving qubit but cannot spread the loss. Because qubit loss no longer spreads, the number of lost qubits arising from a single qubit-loss event no longer grows exponentially with circuit depth. Consequently, loss need not be checked using the beacon protocol after every scheduled layer; instead, we introduce a modified *leakage-detection unit* (LDU) that detects both leakage and loss under the swap-loss model. The new loss-propagation rule is specified in Sec. XII B 1; Sec. XII B 2 gives the LDU and corresponding SEC procedure; and Sec. XII B 3 reports the memory performance of the Q102 and Q66 code blocks in this model.

1. Swap-Loss Model

The moving-qubit noise model of the Walking Cat architecture [1] supplements circuit-level Pauli noise with independent qubit-loss and leakage processes. Let p_{loss} and p_{leak} denote their respective rates per physical operation cycle (POC). After a parallel round of depth Δ POCs, each non-lost qubit is marked lost with probability $\min\{1, \Delta p_{\mathrm{loss}}\}$; among the remaining qubits, each qubit in the computational subspace is marked leaked with probability $\min\{1, \Delta p_{\mathrm{leak}}\}$. We retain the Walking Cat leakage rules: a single- or two-qubit gate involving a leaked qubit is replaced with the identity operation while its associated physical noise is still applied, and a subsequent loss changes the qubit's status from leaked to lost.

In the original moving-qubit noise model, any non-measurement operation involving a lost qubit is replaced with the identity operation, and any two-qubit gate involving a lost qubit propagates the loss to the other qubit involved in the gate. Consequently, if every lost qubit participates in a two-qubit gate, the number of lost qubits can double from one two-qubit-gate layer to the next.

The *swap-loss model* changes only this loss-propagation rule. Suppose a two-qubit gate acts on locations i and j, exactly one of which is marked lost. The intended two-qubit gate is replaced with the identity operation. With probability p_{swap}, however, the

contents of the two locations, including their tracked loss and leakage statuses and tableau entries, are exchanged; with probability $1 - p_{\text{swap}}$, they remain unchanged. With no loss of generality, we set $p_{\text{swap}} = 0.5$ in our simulations; sensitivity analysis of memory performance with the swap-resilient LDU shows that the logical error rate is insensitive to the precise value of p_{swap}, as shown in Fig. 22.

In simulation, this update is implemented by swapping the lost and leaked status flags and applying a SWAP gate to the corresponding entries in the underlying tableau state.

The original Walking Cat architecture's loss propagation rule models a QCCD merge/split operation between an occupied and an empty potential well, which can strongly heat and ultimately eject the surviving ion. The swap-loss model instead assumes potential wells sufficiently deep to retain the heated ion.

2. *Swap-Resilient Loss Detection and Syndrome Extraction*

Our goal is to construct a circuit that detects loss or leakage of a data qubit with probability close to one, even when a loss can swap between the data and ancilla wires. In the absence of loss or leakage, the circuit should preserve the state of the data qubit under observation.

The teleportation-based leakage-detection unit (LDU) used in the Walking Cat architecture [1] assumes that the data and ancilla are swapped/teleported to definite locations. The SWAP-based and teleportation-based syndrome-extraction schemes of Ref. [71] make the same assumption. However, under the swap-loss model, a loss can move to an unmeasured location while a surviving qubit moves to the measured location, so the loss may be missed.

We instead modify Preskill's leakage-detection circuit [72]. The issue with the original LDU is as follows. If the loss swaps with the surviving ancilla qubit at both entangling gates, that qubit returns to its original location in state $|1\rangle$ (which normally indicates a healthy outcome), producing a false negative. At $p_{\text{swap}} = 1/2$, this double-swap trajectory has probability $p_{\text{swap}}^2 = 1/4$, which is too large to neglect.

The *swap-resilient LDU* shown in Fig. 6 adds an X_A gate between the two entangling gates. A computational ancilla outcome 0 accepts the data qubit, whereas outcome 1, 'lost', or 'leaked' triggers recovery. We assume throughout that augmented readout perfectly distinguishes computational, leaked, and lost states.

Proposition 1. *In the absence of additional faults within the gadget, the swap-resilient LDU preserves and accepts every computational input state and flags every leaked or lost input for all possible loss-swap patterns.*

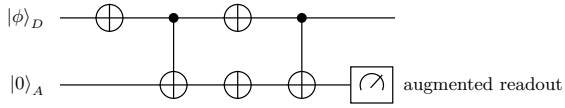


FIG. 6: Swap-resilient LDU for data qubit D and aligned ancilla qubit A. Its final measurement is augmented with loss and leakage readout.

Proof. For $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, the action of the circuit is

$$\begin{aligned} |\phi, 0\rangle &\xrightarrow{X_D} \alpha|1,0\rangle + \beta|0,0\rangle \\ &\xrightarrow{\text{CNOT}_{D\to A}} \alpha|1,1\rangle + \beta|0,0\rangle \\ &\xrightarrow{X_D X_A} \alpha|0,0\rangle + \beta|1,1\rangle \\ &\xrightarrow{\text{CNOT}_{D\to A}} |\phi, 0\rangle, \end{aligned}$$

where each ordered ket lists D before A. Notably, the data state is preserved and the ancilla outcome is 0.

Write ℓ and $\varnothing$ for leaked and lost statuses. For a leaked input data qubit, all gates involving D are suppressed, leaving only

$$(\ell, 0)_{DA} \xrightarrow{X_A} (\ell, 1)_{DA}.$$

The same argument gives ancilla outcome 1 for a lost input with no swaps. With exactly one swap, the surviving ancilla replaces the lost data qubit on D, while the loss moves to A, so augmented readout invariably reports 'lost'.

In the two-swap case, each entangling gate is suppressed and its loss-swap event exchanges the two locations. The intermediate X_A is suppressed while the loss occupies A, whereas X_D flips the surviving qubit:

$$(\varnothing, 0)_{DA} \xrightarrow{\text{swap}_1} (0, \varnothing)_{DA} \xrightarrow{X_D} (1, \varnothing)_{DA} \xrightarrow{\text{swap}_2} (\varnothing, 1)_{DA}.$$

Thus the surviving qubit returns to the ancilla location in state 1. Hence the gadget flags every non-computational input, as summarized in Table V. □

TABLE V: Ideal swap-resilient LDU outcomes, excluding additional faults within the gadget.

Input status of D	Swap events	Ancilla readout
Computational	0	0
Leaked	—	1
Lost	0	1
Lost	1	'lost'
Lost	2	1

The gadget has a nominal depth of three POCs: two entangling-gate layers and one augmented measurement/reset layer. The single-qubit X operations are implemented virtually by physical Pauli-frame tracking, and

therefore do not contribute to the POC depth. A flagged leakage requires one additional POC for leakage reset; a flagged loss additionally requires a fresh ion from the local reservoir. However, since these occur rarely (at rates $\approx p_{\text{leak}} \cdot n \cdot 2$ or $p_{\text{loss}} \cdot n \cdot 2$), they do not affect the nominal depth of the gadget significantly.

We modify the SEC circuit of Ref. [1] to use the swap-resilient LDU in place of the beacon protocol and teleportation-based LDU, as detailed in algorithm 3.

3. Memory Performance of Code Blocks

We simulate the swap-loss model and append the swap-resilient LDU to every SEC. The LDU contributes three POCs of depth. The LDU is simulated with the same Pauli gate noise as in the moving-qubit model, together with the corresponding idle-noise, leakage, and loss channels on idle qubits. Unless stated otherwise, we set

$$p_{\text{loss}} = p/1000, \qquad p_{\text{leak}} = p/10, \qquad p_{\text{swap}} = 0.5. \quad (1)$$

For a physical justification of these parameters, see Ref. [1].

The generalized-bicycle code instances and SEC schedules used in the memory simulations are specified in Table VI.

We evaluate the performance of our primary code blocks, Q102 and Q66, under the swap-loss model at the assumed physical noise rate $p = 10^{-4}$. Direct simulation at this noise rate would require prohibitively many shots, so we extrapolate the swap-loss data from $p \in \{5\times10^{-4}, 8\times10^{-4}, 10^{-3}, 2\times10^{-3}\}$, and the Pauli-noise-only baselines from $p \in \{10^{-3}, 2\times10^{-3}, 3\times10^{-3}\}$, using the three-parameter ansatz $p_{\log}(p) = p^5 \exp(a+bp+cp^2)$. We simulate the Pauli-noise-only model with Stim [73] and the loss- and leakage-enhanced model with the bespoke simulator described in the original Walking Cat architecture paper [1]. We decode both data sets using the beam-search decoder [74] with beam width 32, at most 10 search rounds, 40 initial iterations, and 30 iterations per subsequent round. For Q102, the fitted ansatz is

$$p_{\log}(p) = p^5 \exp\big(20.4344 + 2262.86p - 5.13283\times10^5 p^2\big),$$

which gives a logical error rate of 9.34×10^{-12} per SEC at $p = 10^{-4}$, as shown in Fig. 7.

For Q66, the fitted swap-loss ansatz is

$$p_{\log}(p) = p^5 \exp\big(21.6356 + 574.572p - 9.79513\times10^4 p^2\big).$$

It gives a logical error rate of 2.63×10^{-11} per SEC at $p = 10^{-4}$; the corresponding extrapolated Pauli-noise-only rate is 2.47×10^{-12} per SEC. The sampled data and both fits are shown in Fig. 7.

C. Reusable Cat State Factories with Parallel Layout

We modify the cat-state-factory layout to support the three parallel measurements required for depth-one CCZ injection. The resulting layout provides this parallelism with fewer physical qubits than would be required under the default Walking Cat architecture.

1. Edge-Packed Cat-State Factories for Parallel Measurements

Depth-one injection of the CCZ gates described in Sec. XII D requires three parallel logical measurements. The simplest layout required to accomplish this places three cat-state factories beside each code block, but this arrangement relaxes the Walking Cat architecture's assumption that each cat-state factory is at fixed distance from its associated code block. Longer transport distances may require additional cat-state copies in flight to pipeline logical measurements without increasing gate latency. We therefore place the factories along the code-block edges and determine the number of cat-state copies required by the resulting transport geometry.

In the Walking Cat architecture, measured cat-state qubits circulate back to their factories through a device-wide outer loop so that their carriers can be reused [1]. We introduce a new loop design that minimizes the transport distance: every code block and its assigned factories share a local loop confined to their region.

Let $w_{\text{cat}} \in \mathbb{N}$ denote the cat-state weight, namely the number of physical qubits comprising the cat state. We call a reusable allocation of w_{cat} qubits that holds one weight-w_{cat} cat state a *cat-state register*. A register cycles through cat-state preparation, delivery, measurement, and then returns. Verification ancillas are factory resources and are not part of the register.

Consider an n-qubit code block on which k logical measurements are performed simultaneously. We index the measurements by $j \in \{1,\dots,k\}$ and assign factory $\mathcal{F}_j$ to measurement j. Each factory supplies weight-w_{cat} cat states at the cadence of one state per syndrome extraction cycle (SEC), whose duration is $T_{\text{SEC}}(n)$ POCs. Let r be the number of weight-w_{cat} cat-state registers reserved for each of the k parallel logical measurements, let m be the number of verification rounds, and let $\eta = \tau_{\text{tr}}/\tau_{\text{POC}}$ denote the duration of one transport step in POC units.

For even w_{cat}, the Walking Cat protocol of Ref. [1] prepares and verifies a cat state in

$$T_{\text{prep}}(w_{\text{cat}}, m) = \underbrace{\lceil \log_2 w_{\text{cat}} \rceil + 3m + 3}_{\text{preparation and verification}} + \underbrace{\eta\left(\frac{w_{\text{cat}}}{2} + m - 1\right)}_{\text{factory-internal transport}} \quad (2)$$

POCs.

Algorithm 3: Syndrome extraction with the loss- and leakage-detection gadget only

Data: A data register D, an ancilla register A, and a valid maximally parallel schedule $\Sigma = \big((T_X^{(1)}, T_Z^{(1)}), \ldots, (T_X^{(w_{\mathrm{check}})}, T_Z^{(w_{\mathrm{check}})})\big)$ for a three-ring code with $a = 2$

Prepare the ancilla qubits in $|0\rangle$;
Apply the loss- and leakage-detection gadget of Fig. 6 in parallel between each data qubit and its aligned ancilla qubit;
Measure the gadget ancillas using augmented loss/leakage readout;
Let $\mathcal{K}_1$ be the set of indices whose gadget ancilla returns computational outcome 1;
for $i \in \mathcal{K}_1$ **do**
    Check data qubit D_i to determine whether it is leaked or lost;
    if D_i is lost **then**
        Replace D_i by a reloaded qubit;
    Reset D_i to the maximally mixed state $I/2$;
Let $\mathcal{K}_{\mathrm{lost}}$ be the set of indices whose gadget ancilla readout reports 'lost';
for $i \in \mathcal{K}_{\mathrm{lost}}$ **do**
    Check data qubit D_i for loss;
    Reload every qubit in $\{A_i, D_i\}$ that is lost;
    Reset D_i to the maximally mixed state $I/2$;
Prepare all ancilla qubits in $|+\rangle$;
Relabel the ancilla in software so that the initial alignment matches the first schedule pair in Σ;
Apply the scheduled parallel gates for $(T_X^{(1)}, T_Z^{(1)})$: CNOT gates from each aligned X-check ancilla to its data qubit, and CZ gates between each aligned Z-check ancilla and its data qubit;
for $\tau \in \{2, \ldots, w_{\mathrm{check}}\}$ **do**
    Compute the common long-ring shift r from the family change between rounds $\tau - 1$ and τ, together with (s_X, t_X) from $T_X^{(\tau-1)}$ and $T_X^{(\tau)}$ and (s_Z, t_Z) from $T_Z^{(\tau-1)}$ and $T_Z^{(\tau)}$;
    Apply the transport step consisting of a long-ring cycle by r on all ancilla qubits together with the medium/short shifts (s_X, t_X) on $\mathcal{A}_X$ and (s_Z, t_Z) on $\mathcal{A}_Z$;
    Apply the scheduled parallel gates for $(T_X^{(\tau)}, T_Z^{(\tau)})$: CNOT gates from each aligned X-check ancilla to its data qubit, and CZ gates between each aligned Z-check ancilla and its data qubit;
Measure all ancilla qubits in the X basis using augmented loss/leakage readout;
Reset any ancilla qubit whose final readout reports 'leaked' or 'lost' before the next syndrome extraction round;

TABLE VI: Generalized-bicycle code instances and syndrome-extraction schedules used in the memory simulations. The symbols A_i and B_i index the monomials of $a(x)$ and $b(x)$, respectively, in the order shown; $^{\mathsf{T}}$ denotes a transposed interaction.

Name	Code specification	Schedule permutation Σ
Q102	$[\![102, 22, 9]\!]$, $(l, m) = (51, 1)$, $a(x) = x^{22} + x^{26} + x^{37} + x^{50}$, $b(x) = x^{19} + x^{28} + x^{29} + x^{35}$	$\big((A_1, A_2^{\mathsf{T}}), (B_1, B_4^{\mathsf{T}}), (B_2, B_3^{\mathsf{T}}), (A_3, A_4^{\mathsf{T}}), (A_4, A_3^{\mathsf{T}}), (B_3, B_2^{\mathsf{T}}), (B_4, B_1^{\mathsf{T}}), (A_2, A_1^{\mathsf{T}})\big)$
Q66	$[\![66, 4, 10]\!]$, $(l, m) = (33, 1)$, $a(x) = 1 + x + x^3 + x^{10}$, $b(x) = 1 + x + x^8$	$\big((B_2, B_1^{\mathsf{T}}), (A_2, A_1^{\mathsf{T}}), (A_3, A_4^{\mathsf{T}}), (B_3, B_3^{\mathsf{T}}), (A_4, A_3^{\mathsf{T}}), (A_1, A_2^{\mathsf{T}}), (B_1, B_2^{\mathsf{T}})\big)$

We pack factories along the long edges of the code block, as shown in Fig. 8. For placement, a weight-w_{cat} factory is assigned a width of w_{cat} carrier sites. One long edge can therefore accommodate

$$s(n, w_{\mathrm{cat}}) = \left\lfloor \frac{n}{w_{\mathrm{cat}}} \right\rfloor \tag{3}$$

factories. We place factories $\mathcal{F}_1, \ldots, \mathcal{F}_{\min\{k,s\}}$ consecutively along the top edge and, when $k > s$, continue with $\mathcal{F}_{s+1}$ along the bottom edge. Two long edges suffice whenever $k \le 2s(n, w_{\mathrm{cat}})$, without occupying either short edge on the sides of the code blocks. The memory and cat-factory components of Ref. [1] each span four physical qubit rows. At the granularity of the present transport model, an n-qubit code block therefore occupies a $4 \times n$ rectangle and the loop immediately surrounding it has length

$$P_{\mathrm{loop}}(n) = 2n + 8 \tag{4}$$

transport steps.

This routing policy also accommodates representatives whose supports are interleaved across the data qubits,

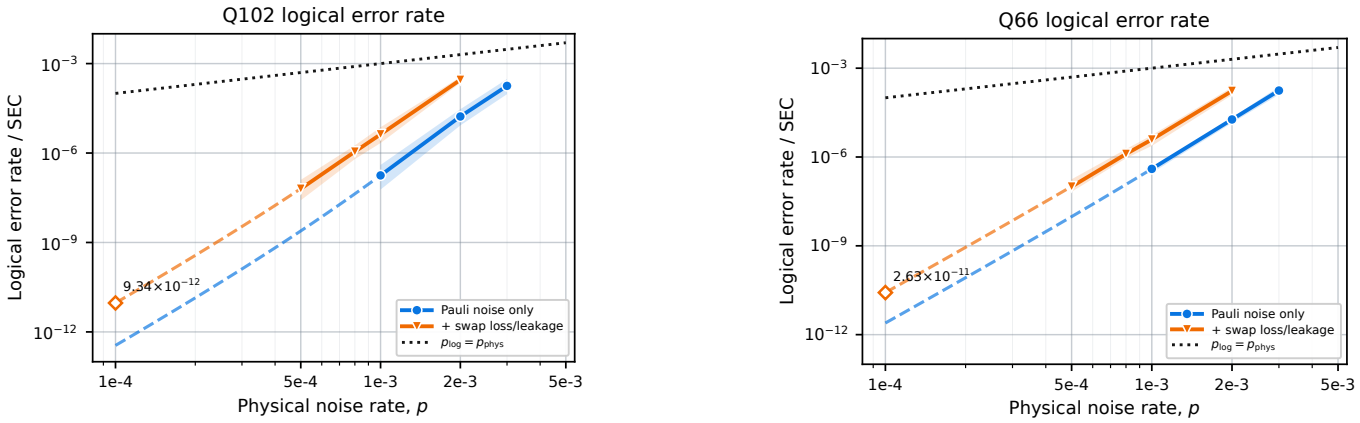


FIG. 7: Memory performance of the $[[102, 22, 9]]$ Q102 code (left) and the $[[66, 4, 10]]$ Q66 code under the swap-loss model. For each code, the Pauli-noise-only baseline is compared with the corresponding swap-loss model, with $p_{\text{leak}} = p/10$, $p_{\text{loss}} = p/1000$, and $p_{\text{swap}} = 0.5$. Dashed curves extrapolate each series to $p = 10^{-4}$ using the three-parameter fifth-order ansatz $p_{\log}(p) = p^5 \exp(a + bp + cp^2)$.

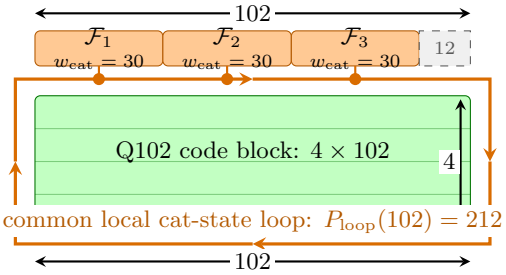


FIG. 8: Local cat-state circulation for Q102. Three weight-30 factories occupy 90 of the 102 sites along the top edge, leaving 12 sites unused; the bottom edge is not needed. In general, factories are packed along the top edge up to the capacity in eq. (3), then continued along the bottom edge. Note that every factory has the same circulation distance $2n + 8$.

because the interchangeable qubits within each cat state can be staged in support order. We now determine how many registers are needed to sustain this delivery schedule.

Proposition 2. *Assume that a successful cat-state preparation is available after $T_{\text{prep}}(w_{\text{cat}}, m)$ POCs and that the local loop can transport the cat states required for all k parallel logical measurements without contention. Assume also that the physical representatives for the parallel measurements have pairwise-disjoint supports. Measure time from the instant at which the cat state for the current round is consumed. Assume that the next cat state is not required until the next cat-based measurement, one SEC later, and that the register can return, be prepared and verified, and attempt redelivery concurrently with that intervening SEC. The smallest register count for each measurement that sustains one weight-w_{cat} cat state per SEC is*

$$r = \left\lceil \frac{T_{\text{prep}}(w_{\text{cat}}, m) + \eta P_{\text{loop}}(n)}{T_{\text{SEC}}(n)} \right\rceil. \tag{5}$$

Proof. Suppose register A supplies the cat state for the current measurement round. From the moment that state is consumed, the register is unavailable for the turnaround time

$$T_{\text{prep}}(w_{\text{cat}}, m) + \eta P_{\text{loop}}(n)$$

while it returns to its original cat-state factory around the local loop, undergoes preparation and verification, and then travels along the loop to the next measurement. The return followed by the next delivery has total length $P_{\text{loop}}(n)$. Let

$$q = \left\lceil \frac{T_{\text{prep}}(w_{\text{cat}}, m) + \eta P_{\text{loop}}(n)}{T_{\text{SEC}}(n)} \right\rceil.$$

During this turnaround, $q - 1$ other registers supply the intervening demands at times $T_{\text{SEC}}(n), \ldots, (q - 1)T_{\text{SEC}}(n)$, and register A returns for the demand at time $qT_{\text{SEC}}(n)$. Thus $r = q$ registers suffice. For example, when $q = 2$, register B is consumed one SEC after A, and A is consumed again after two SECs. With fewer registers, A would be reused before it is ready. □

Every factory requires only one register if

$$T_{\text{prep}}(w_{\text{cat}}, m) + \eta(2n + 8) \le T_{\text{SEC}}(n). \tag{6}$$

Under this condition, proposition 2 gives $r = 1$ for every measurement, for k cat-state registers in total.

The concrete Q102 and Q66 register counts used in the component footprints are calculated in appendix E 1.

2. Reusing Cat States

The elliptic-curve circuit's limited global logical-measurement parallelism leaves many cat-state resources idle for long periods of time under the default Walking Cat architecture's per-code-block cat-state allocation policy. This motivates provisioning only enough cat states to meet the circuit's peak measurement parallelism, then transporting and reusing them across the device as needed. The basic unit of reuse is the complete set of resources needed to pipeline one logical measurement around the local loop: r weight-w_{cat} cat-state registers, with r given by proposition 2, together with one bank of w_{cat} verification ancillas for successive cat-state preparations. We call this a *cat-state bundle*.

For the global routing analysis, we arrange the targets of each of the k parallel logical measurements into an ordered stream. Each stream is served by a pool of reusable cat-state bundles that may travel between code blocks. We ask how many bundles each stream needs to avoid delaying logical measurements.

The registers within each bundle allow one cat state to be prepared and verified while another is consumed, as in proposition 2; the ancilla bank travels with the registers so that the bundle can operate beside any memory block. Upon arrival at the destination code block, the bundle docks at one of the cat-state factories along the target block's edge and undergoes the preparation and verification procedure of Sec. XII C 1. At least one bundle remains beside the active block throughout its logical measurement, but the bundle that begins the protocol need not be the one that finishes it. The bundles assigned to each measurement stream follow a handoff schedule, illustrated for two bundles in Fig. 9. The lookahead bundle begins the measurement at the current target while the trailing bundle travels to that same block and is prepared and verified there. The trailing bundle then takes over the ongoing measurement, releasing the lookahead bundle to advance to the following target. Repeating this handoff moves the bundles through the ordered stream of logical measurements.

Consider M memory blocks packed in an R-by-C rectangle, where $M \le RC$ and unused positions, if any, are left empty. As above, each block occupies a 4-by-n rectangle. We leave one additional horizontal routing row between neighboring block rows. The maximum Manhattan distance between corresponding ports of two memory blocks is therefore

$$D_{\max}(R, C, n) = n(C-1) + 5(R-1) \tag{7}$$

transport steps. Let T_{LM} be the duration in POCs of the shortest logical measurement in the workload.

Proposition 3. *Let $L = \eta D_{\max}(R, C, n) + T_{\mathrm{prep}}(w_{\mathrm{cat}}, m)$ and assume $L \le T_{\mathrm{LM}}$. Assume also that, under the ouroboros block layout introduced for the idle-frame-clearing schedule in Sec. XII D 2, the transport routes serving simultaneous logical measurements do not cross one another. For a computation with at most k simultaneous logical measurements, the handoff schedule can execute any sequence of measurement locations without transport delays using at most*

$$N_{\mathrm{bundle}}(k, R, C, n, w_{\mathrm{cat}}, m) = k\left(1 + \left\lceil \frac{2\left[\eta D_{\max}(R, C, n) + T_{\mathrm{prep}}(w_{\mathrm{cat}}, m)\right]}{T_{\mathrm{LM}}} \right\rceil\right). \tag{8}$$

In particular, two bundles per measurement stream suffice whenever

$$2\left[\eta D_{\max}(R, C, n) + T_{\mathrm{prep}}(w_{\mathrm{cat}}, m)\right] \le T_{\mathrm{LM}}. \tag{9}$$

Proof. Partition the bundles among k independent measurement streams. In each stream, use one trailing bundle and $p = \lceil 2L/T_{\mathrm{LM}} \rceil$ lookahead bundles. At the start of a logical measurement, the next lookahead bundle is already prepared at the target block and begins the measurement. The trailing bundle, released when the preceding logical measurement ended, travels to this same target and is prepared and verified in at most L POCs. Because $L \le T_{\mathrm{LM}}$, it can take over before the current measurement ends.

This handoff releases the active lookahead bundle at most L POCs after the measurement began. That bundle is next required p measurements later, so it has at least $pT_{\mathrm{LM}} - L \ge L$ POCs to travel to and prepare at its next assigned target. It is therefore ready before that measurement begins. Repeating the handoff gives a delay-free pipeline with $p+1$ bundles per lane, which yields eq. (8). When $2L \le T_{\mathrm{LM}}$, $p = 1$: the leading bundle starts each target, the trailing bundle takes over, and the released leading bundle advances again. □

The `secp256k1` device packs its 69 Q102 memory blocks and four Q102 Logical CliNR blocks in a nine-by-nine grid, as shown in Fig. 5. Consequently,

$$D_{\max}(9, 9, 102) = 102(9-1) + 5(9-1) = 856 \tag{10}$$

transport steps, or $\eta D_{\max} = 42.8$ POCs for $\eta = 1/20$. The subsequent local preparation and verification takes $T_{\mathrm{prep}}(30, 2) = 14.8$ POCs, for a total lookahead time of 57.6 POCs. The two successive pipeline legs—bringing the trailing bundle to the active block and then sending the released lookahead bundle onward—take at most $2(57.6) = 115.2$ POCs. At the physical error rate $p = 10^{-4}$, a Viterbi measurement with cat-state weight $w_{\mathrm{cat}} \ge 16$ targeting a logical-measurement error rate of $\epsilon = 10^{-10}$ requires at least five SECs [1], hence $T_{\mathrm{LM}} \ge 5T_{\mathrm{SEC}} = 135$ POCs for Q102. Therefore

$$N_{\mathrm{bundle}}(k, 9, 9, 102, 30, 2) = k\left(1 + \left\lceil \frac{2(42.8 + 14.8)}{135} \right\rceil\right) = 2k. \tag{11}$$

Thus, two bundles suffice for each Q102 measurement stream. We call this stream-level allocation a *cat-state*

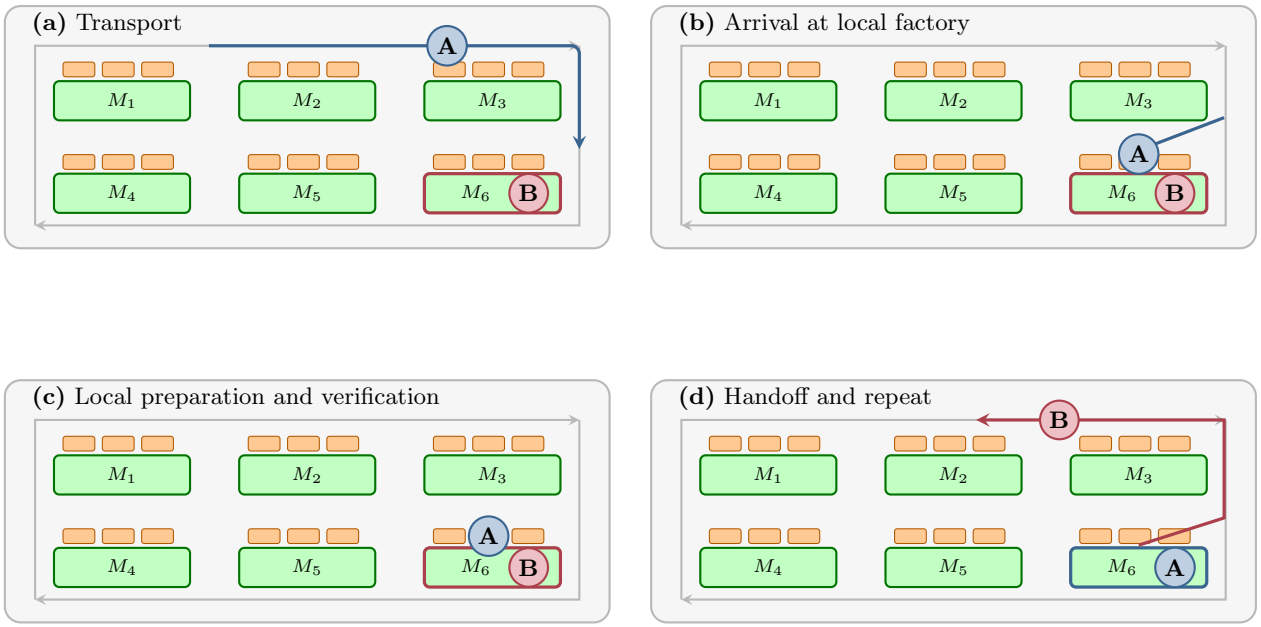


FIG. 9: Device-level handoff schedule for reusing two cat-state bundles in a measurement stream. Each memory block has three cat-state factories, shown in orange. Lookahead bundle B begins the logical measurement at M_6 while bundle A follows the highlighted route toward that same block (a). A arrives at one of M_6's local factories (b) and is prepared and verified as B continues measuring (c). A then takes over the ongoing measurement at M_6, releasing B to advance toward the following target (d).

bundle pair. One set of k bundles can start the measurements at the current target blocks while the other set arrives, is prepared and verified, and takes over; the released set then moves ahead to the next k targets. For the $k = 3$ measurements used by depth-one CCZ injection, this requires three mobile cat-state bundle pairs (six individual bundles) rather than the $73k = 219$ block-local bundles of the default Walking Cat architecture, a factor-36.5 reduction in spatial overhead without increasing logical-measurement latency.

The lookahead-based pipelining policy described above assumes that the target block of each lane is known at least L POCs before its measurement begins. Feedforward can violate this assumption when a measurement outcome determines which previously inactive code block is measured next: the released lookahead bundle has no unique destination until that outcome is available. A measurement on the selected block can then incur a stall of up to $\eta D_{\max} + T_{\text{prep}}(w_{\text{cat}}, m) = L$ POCs while the leading bundle is transported, prepared, and verified. So, the delay-free guarantee of proposition 3 does not extend across such a conditional change of target block without provisioning bundles for the alternative destinations.

In the Walking Cat architecture, frame tracking does not cross code-block boundaries. Consequently, a tracked conditional Clifford gate may change a subsequent logical measurement by conjugating its observable, but it cannot change its target block. And, otherwise, we observe that no conditional operations in the actual compiled elliptic curve point addition circuits select among alternative destination blocks.

Let $w_{\max}$ be the largest weight among the possible physical representatives of the conjugated logical observable. A weight-$w_{\max}$ cat state can therefore be prepared at the known target before the frame is resolved. If the selected representative has lower weight, we pad it with identity factors to weight $w_{\max}$. The destination and cat-state size are therefore independent of the conditional outcome, so the handoff schedule remains applicable, with its preparation-time and bundle-count bounds evaluated at $w_{\text{cat}} = w_{\max}$.

Under these constraints, k cat-state bundle pairs, comprising $2k$ individual bundles, suffice to pipeline the k parallel logical measurements of depth-one CCZ injection.

D. The CLAW ISA and Depth-One CCZ Injection

The *Clifford-frame and Logical-operator Acceleration for Workloads* (CLAW) ISA extends the logical instruction set of the Walking Cat architecture [1] with operations tailored to elliptic curve point addition circuits. It targets two principal bottlenecks in the compiled circuits. The first is the cost of implementing Clifford gates. Under the default Walking Cat assumptions, only pure-X and pure-Z logical measurements are guaranteed to admit physical representatives of weight at most 30 and hence to be accessible using a weight-30 cat state. Tracking a Clifford gate such as a conditional CZ can conju-

gate these into more general Pauli measurements, so the baseline architecture cannot guarantee that every Clifford correction can be implemented by frame tracking. CLAW instead checks accessibility for the specific logical measurements required in the compiled elliptic curve point addition circuits. We compile each component assuming a clean input frame and use Monte Carlo sampling to check the conjugated logical measurements along its conditional Clifford trajectories. Every measurement encountered has an accessible Q102 physical representative of weight at most 30. *Idle frame clearing*, described in Sec. XII D 2, maintains the clean-input-frame invariant without introducing additional runtime latency. This extension dramatically reduces the number of logical measurements required to implement Clifford gates in the elliptic-curve point-addition circuit, leaving Toffoli gates as the dominant runtime bottleneck.

To address this remaining bottleneck, we extend the CLAW ISA to include an *LMP* operation that measures in parallel any set of logical operators admitting pairwise-disjoint physical representatives; the corresponding measurement protocol is described in Sec. XII D 1. In particular, the LMP operation allows the three disjoint logical measurements required for CCZ state injection to be performed simultaneously, yielding an optimal depth-one implementation of the Toffoli gate as described in Sec. XII D 3.

1. *Parallel Cat-State Measurements*

We extend the cat-state measurement protocol of the Walking Cat architecture [1] to measure several logical Pauli operators in parallel. Let $P_1, \ldots, P_k$ be logical operators with physical representatives $\widetilde{P}_1, \ldots, \widetilde{P}_k$ whose supports are pairwise disjoint. We insist upon disjointness here to enable a simple analysis of logical measurement fault rate–representatives with overlap may introduce correlated error when measured together. Additionally, for simplicity, suppose without loss of generality that every representative has weight w_{cat}. We prepare and verify k distinct weight-w_{cat} cat states, one for each representative. The ath qubit of cat state C_j is coupled to the ath data qubit in the support of $\widetilde{P}_j$. Because the data supports are disjoint, all kw_{cat} data–cat interactions can be scheduled in the same physical gate layer.

The k verified cat states are placed side by side and transported as a single bundle. The bundle is slid into alignment with the union of the k supports, the data–cat gates are applied simultaneously, and all cat qubits are then measured in the X basis in the same readout round. Taking the parity of the outcomes within C_j yields the measurement bit for P_j, as illustrated for two parallel logical measurements in Fig. 10. This protocol then allows us to make k logical measurements in a single physical measurement layer.

Because the physical representatives have disjoint supports, the corresponding measurements commute.

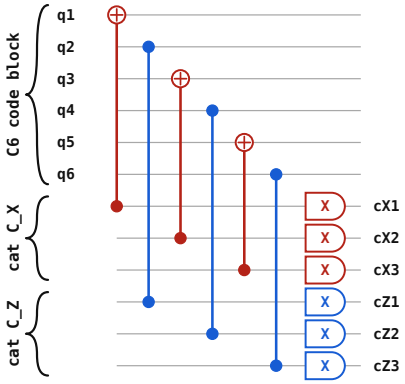


FIG. 10: Parallel cat-state measurement of two disjoint logical representatives in the $[\![6, 2, 2]\!]$ "C6" code. The prepared cat states C_X and C_Z control data–cat gates on the disjoint representatives $\overline{X}_0 = \texttt{XIXIXI}$ and $\overline{Z}_1 = \texttt{IZIZIZ}$, respectively. Red controlled-X gates belong to the logical-X measurement, while blue controlled-Z gates belong to the logical-Z measurement.

Therefore, in the noiseless setting, executing the cat-state measurements in parallel is equivalent to executing them sequentially in any order. This guarantee persists under the swap-loss model because its faults are local. Each logical measurement uses a distinct cat state and a disjoint set of data–cat interactions, and faults are sampled locally for each physical operation. Consequently, no fault arising within one cat-state measurement—including a hook error or a swap induced by a lost ion—can propagate into another gadget. Parallel execution therefore leaves the outcome distribution and error probability of each logical measurement unchanged relative to executing that gadget alone in the same physical layer.

2. *Idle Frame Clearing*

As in the Walking Cat architecture [1], we implement conditional Clifford corrections whenever possible by Clifford frame-tracking rather than applying them physically. The compiler updates the block's Clifford frame U and replaces each subsequent target logical Pauli operator P by $U^\dagger P U$. When a subsequent non-Clifford operation requires a trivial Clifford frame, the tracked Clifford gate can be implemented using logical CliNR [51], thereby teleporting the encoded state into a block whose Clifford frame is reset to $U = I$, as summarized in Fig. 11. We refer to this reset as *clearing* the Clifford frame. Because logical CliNR can be applied only once a block's Clifford frame is known, frame clearing is generally bottlenecked by resolving the classical conditions associated with conditional Clifford gates. That generally means it's only safe to apply logical CliNR once the block becomes idle, or we must actively pause

execution to clear the frame. Logical CliNR uses two auxiliary memory blocks, A and B, as resource blocks; both are encoded using the same code as the data block whose frame is being cleared. At completion, the logical state of the cleaned block has been teleported to B, whose Clifford frame is trivial, while the original data block and A can be reset and reused.

Optimizing the CLAW ISA for elliptic curve point addition circuits requires identifying the precise set of logical Pauli operators actually measured by each compiled component. To make this tractable, we compile components independently under an invariant: each memory block in a component's support has a trivial Clifford frame immediately before its first non-Clifford use. To enforce this invariant, we must use frame clearing on each block in a component's support.

Frame clearing can run in parallel with logical measurements on other blocks. Because most components of the elliptic-curve point-addition circuit are serial, each block is idle long enough to clear its frame without extending the critical path. More generally, the elliptic-curve circuits consist of a sequence of components that act serially on ordered sets of logical blocks: each component proceeds down its blocks and then uncomputes in reverse order, with occasional shallow layers of parallel gates between components; see Fig. 12. We call these components *serial components*.

We therefore pipeline frame clearing ahead of each serial component's execution, as illustrated in Fig. 13.

However, in order to keep up with the serial execution, we must ensure that logical CliNR can clear a block in less time than the serial execution takes to process all measurements in a code block. To achieve this, we optimize logical CliNR by measuring MECM stages in parallel when they are compatible under the rules of parallel cat-state measurements. Within each MECM stage, we connect two outputs in a compatibility graph when their physical representatives can be chosen with disjoint supports on both CliNR resource blocks and with weight at most 30, then find a maximum matching; unmatched stages are measured individually. By sampling uniformly from the Clifford frames encountered in a simulation of the compiled elliptic-curve point-addition circuit, we find that the optimized MECM portion of a frame clear takes 41.242 ± 0.276 Q102 SEC layers. The transversal CNOT gate is followed by one additional Q102 SEC for decoding, so a complete frame clear takes 42.242 ± 0.276 Q102 SECs. Using the joint CCZ injection duration of approximately 5.46 Q102 SECs derived in eq. (26), this corresponds to approximately 7.73 logical-measurement intervals to clear a block. The serial components expose a new block after approximately 7.5 such intervals; it therefore requires at least two instances of logical CliNR to keep up with the serial execution.

Because we apply logical CliNR only to serial components, the compiler can lay out the data blocks used by those components as an ordered register following their serial traversal order. The timing estimate above shows that two logical CliNR instances are required to ensure that frames are always cleared ahead of serial execution. We divide the register between them by alternating blocks: one instance clears the blocks in even positions, and the other clears those in odd positions. Thus, each instance clears only every other block. Since, typically, execution of serial components in the elliptic curve point addition circuits reaches a new block every 7.5 logical-measurement intervals, each CliNR instance has $2 \times 7.5 = 15$ intervals before it is needed again, enough time to clear a block in approximately 7.73 intervals.

The transversal CNOT gate in logical CliNR requires data qubits to be transported close to each other to perform pairwise physical CNOT gates. We therefore keep the A resource block in the logical CliNR protocol adjacent to the data block during injection. Each data qubit then moves by at most the width of one Q102 block to interact with its corresponding resource qubit, minimizing errors due to idling, loss, and leakage during transport.

The `secp256k1` device introduced in Sec. XII A has a 69-data-block register. Assigning alternating blocks to the two logical CliNR instances divides this register into groups of 35 and 34 data blocks. Adding one A/B resource pair, or two blocks, to each group gives 37 and 36 blocks. Each group fits on a closed nearest-neighbor path through a 4×10 rectangle, which provides 40 cells. The two rectangles therefore leave three and four cells empty, respectively. Stacking them gives an 8×10 layout containing the 69 data blocks, four resource blocks, and seven empty cells. We call this assignment of the alternating block subsequences and their resource pairs to closed nearest-neighbor paths the *ouroboros block layout*. As the application processes further serial components, the compiler can append each new traversal to the end of the current execution path, which wraps in a circular pattern around the data blocks—hence the name "ouroboros."

Under the ouroboros block layout, it remains to show that each logical CliNR instance can follow the path of serial execution on its data blocks while retaining nearest-neighbor locality with the next target (and, with this condition, execute a transversal CNOT gate with minimal overhead). A single application of logical CliNR results in a swap of the B and data blocks, leaving A adjacent to the next data block. If D_i has no frame to clear, one can simply cyclically shift the blocks, implemented by swapping corresponding physical qubits between the blocks. Induction over these local permutations shows that the resource pair remains adjacent to its next target for the entire sequence of clearing a serial application component ahead of execution. Fig. 14 illustrates the ouroboros block layout for cleaning three partially overlapping serial components.

The ouroboros block layout also accommodates a component that starts within the support of its predecessor. In this case, the logical CliNR instance follows the predecessor's execution across the "ascending" execution of logical measurements, cleaning the frame of blocks with

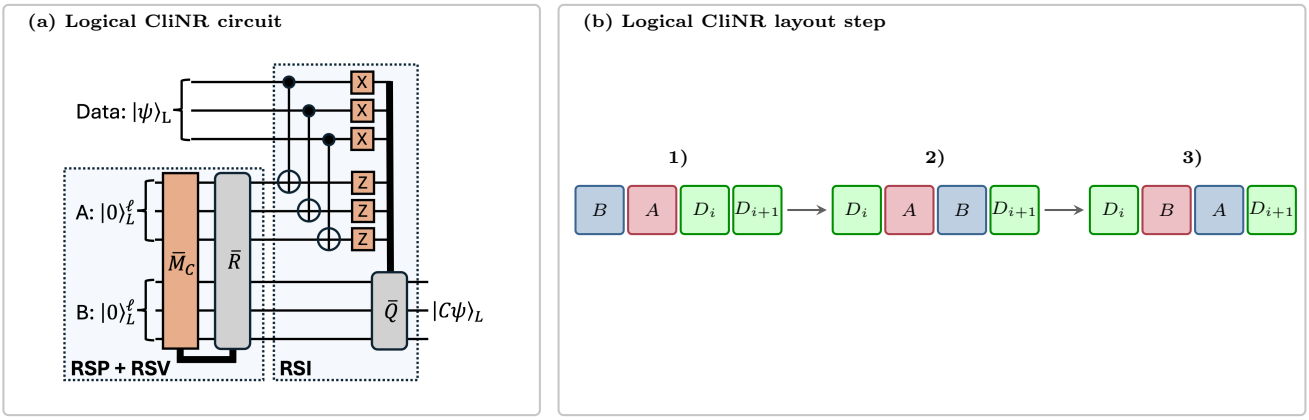


FIG. 11: Logical CliNR and its compiler-level motion primitive. (a) The logical CliNR circuit of Webster and Delfosse [51]. Joint stabilizer measurements and a Pauli correction prepare and verify a logical resource state across auxiliary blocks A and B. A transversal CNOT gate from the data block to A, followed by destructive X- and Z-basis measurements and the Pauli correction $\overline{Q}$, teleports $C|\psi\rangle_L$ into block B. (b) One forward step begins with adjacent ordering $[B, A, D_i, D_{i+1}]$. Exchanging the B carrier with D_i gives $[D_i, A, B, D_{i+1}]$. The physical carriers are then kept fixed while the semantic roles A and B are freely exchanged, yielding $[D_i, B, A, D_{i+1}]$. The renewed triple $[B, A, D_{i+1}]$ is therefore ready for the next CliNR step.

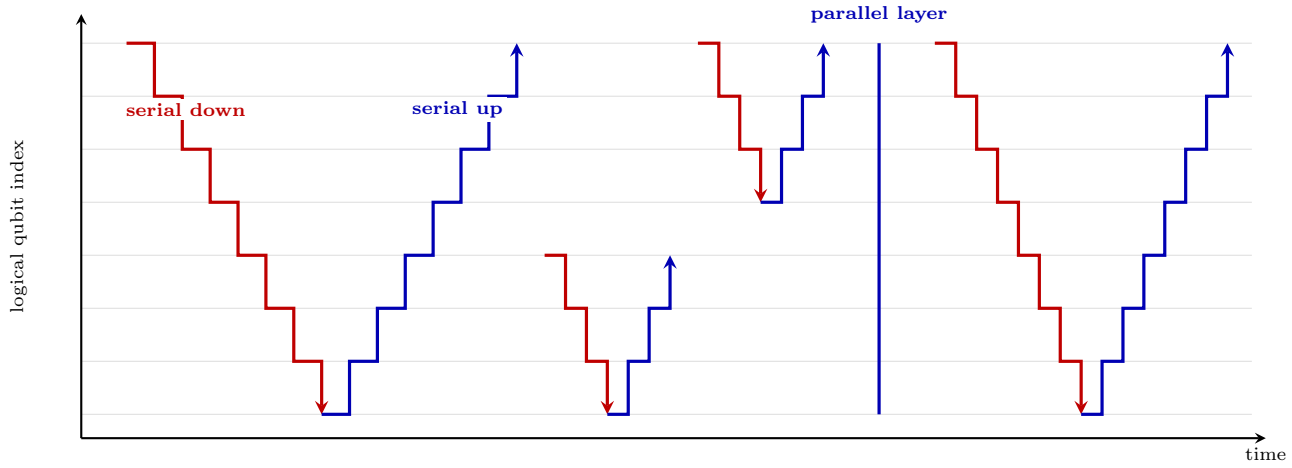


FIG. 12: Schematic temporal structure of the compiled elliptic-curve point-addition circuits. Time runs from left to right, and each row denotes a logical-qubit index. A serial component (e.g., an adder) first visits its support in descending index order (red) and then returns in ascending order (blue). In between components, we may have shallow layers of operations ran all in parallel (e.g., large controlled multi-target gates for conditional inversions), and/or dense layers of operations on a single block (e.g., a table lookup's depth-first selector traversal).

a nontrivial frame and simply making full-block swaps to the intervening blocks whose frames are already clean. They may stop as soon as they reach the first target of the later component; see Fig. 15. In the worst case, the last block released by the earlier component is also the first block required by the later one. Its clearing cannot be hidden behind earlier execution, so the transition stalls for one logical CliNR operation, or approximately 7.73 logical-measurement intervals.

3. *Depth-One* CCZ *Injection*

As shown in Fig. 16, a Toffoli gate is a CCZ gate conjugated by Hadamard gates on its target qubit. The CCZ gate can be injected via one-bit gate teleportation [75, 76] by measuring a joint logical- Z operator between each qubit of the CCZ resource state and the corresponding data qubit, followed by destructive X-basis measurements of the resource qubits and outcome-dependent corrections. The three joint logical- Z measurements have disjoint supports and can therefore be performed in parallel, so the injection requires only one logical-

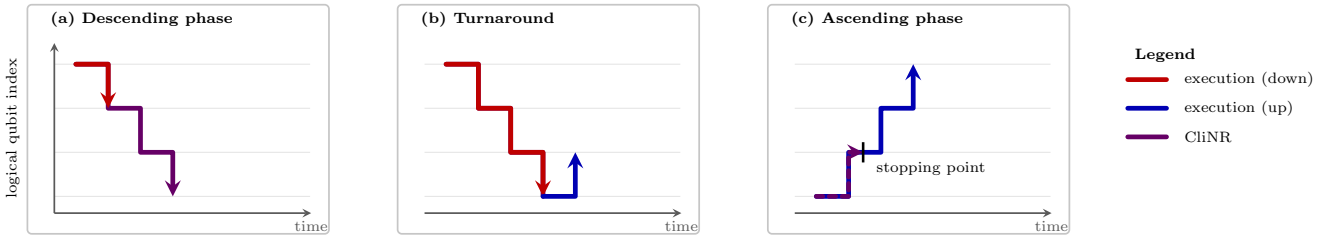


FIG. 13: Idle frame clearing around a serial component. Time runs from left to right, and vertical position denotes logical-qubit index. (a) CliNR cleans the frames of code blocks that are pending execution in a serial component. (b) Execution in a serial component reaches the end of the code blocks, and must now uncompute them, iterating in reverse order. (c) CliNR cleans the frames of the uncomputed blocks up to the first block of the next serial component.

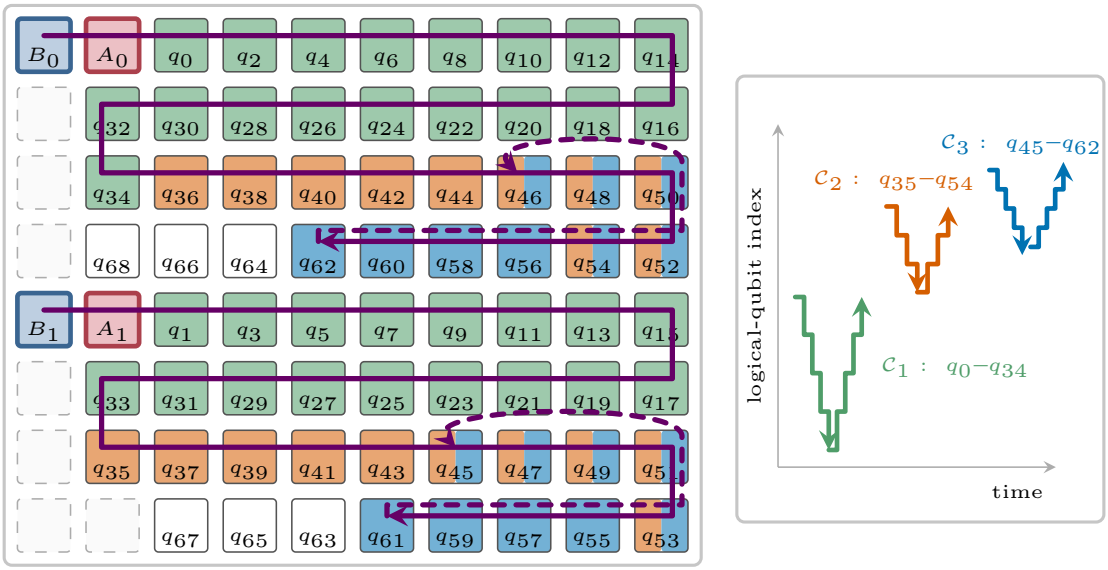


FIG. 14: Logical CliNR for three serial components with partially overlapping supports. Each pair (B_i, A_i) provides the two resource blocks for one CliNR instance. The diagram shows all 69 data blocks, $q_0, \ldots, q_{68}$; the white blocks $q_{63}, \ldots, q_{68}$ are unused. The diagram on the right shows the three serial components executing in sequence in application order. The first component acts on $q_0, \ldots, q_{34}$. The second acts on $q_{35}, \ldots, q_{54}$, and the third acts on $q_{45}, \ldots, q_{62}$, overlapping the second component on $q_{45}, \ldots, q_{54}$. The two CliNR instances begin at the first even- and odd-indexed blocks of each support, then clear frames ahead of execution along the solid purple paths. Because the third component overlaps the second, the resource pairs must backtrack to the start of the overlap before they can begin clearing the third component; the dashed purple paths show this backtracking.

measurement layer. The subsequent X-basis outcomes induce only Pauli corrections, which can be tracked in a Pauli frame and need not be available until the following gate has completed. Their readout is generally faster than the 5.46 Q102 SECs required for the joint logical measurements, and hence does not add another layer to the critical path. Lastly, the compiler ensures that both operands of every conditional CZ correction reside in the same memory block, as described in Sec. X, so each correction can be absorbed into the block's Clifford frame without adding a logical-measurement layer.

We quantify the measurement depth after compilation using Monte Carlo simulation. For each independently compiled circuit component, we draw 10,000 samples of random outcomes for conditional Clifford corrections. Each sample begins with a trivial Clifford frame on every input block, as required by the compilation invariant and guaranteed during execution by the idle-frame-clearing procedure of Sec. XII D 2. After propagating the sampled Clifford frames and scheduling the resulting logical measurements, we find that every sampled conjugated logical operator has an accessible physical representative. The empirical frequency resolution is therefore 10^{-4} per component; under independent sampling, observing no inaccessible trajectory gives the one-sided 95% upper confidence bound 3×10^{-4} on the probability that a trajectory contains an inaccessible logical operator, for each component. Through this simulation of all the ap-

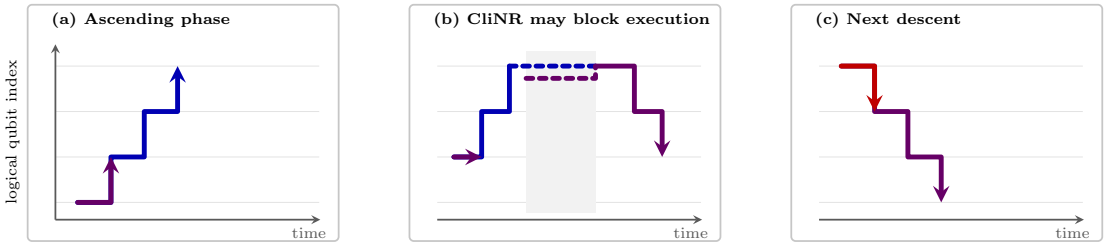


FIG. 15: Logical CliNR on consecutive serial components with overlapping support. Time runs from left to right, and vertical position denotes logical-qubit index. (a) Execution of a serial component begins uncomputing results, iterating across blocks in reversed order. (b) If the last block of a serial component aligns with the start of the next one, the logical CliNR instance working on the first component must block execution. Transitions to disjoint or already prepared supports avoid this. (c) The next execution begins once its first target is clean, while logical CliNR continues just in time ahead of it.

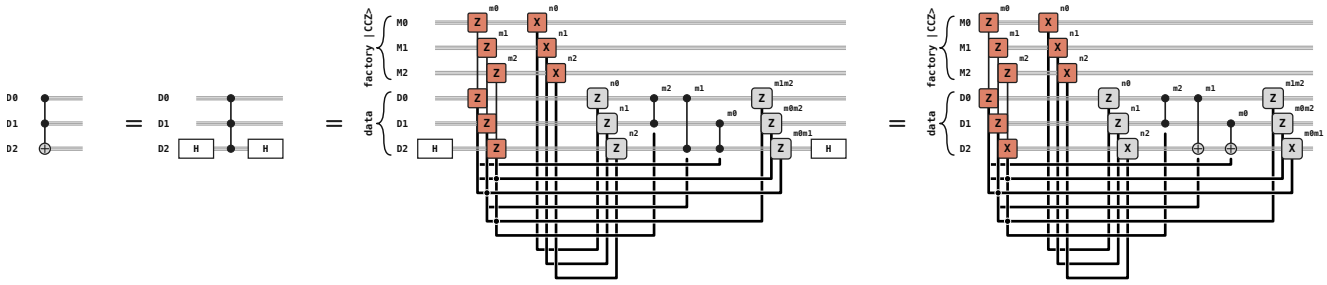


FIG. 16: Circuit equivalence between a Toffoli gate, a CCZ gate conjugated by Hadamard gates on the target qubit, a CCZ injection circuit conjugated by Hadamard gates on the target qubit, and the same circuit with Hadamards implemented via conjugation with the logical measurements of the CCZ injection circuit.

plication components, we find that 93.6% of Toffoli gates have measurement depth one. The remaining gates require depth two, giving a mean measurement depth of 1.07.

E. Eastinthillation CCZ Factory

We design a dedicated "Eastinthillation" factory (Eastin + synthillation) that produces CCZ states directly in a three-ring code block. We note that the alternative name "East Mode" may be used in less formal contexts— for instance, in fan-made songs about this paper. The Eastinthillation factory's resource totals and performance estimates are summarized below.

The factory applies the Eastin Toffoli-state synthillation protocol [77, 78] to eight T states produced by zero-level distillation in the $[\![6, 2, 2]\!]$ code (henceforth referred to as the C6 code). This T-state distillation procedure is detailed in Sec. XII E 1. On acceptance, the protocol outputs a Toffoli state. Applying a Hadamard to its target qubit converts it into a CCZ resource state. The Eastinthillation factory circuit and its physical measurement schedule are described in Sec. XII E 2, while Sec. XII E 3 analyzes the factory's logical error and rejection rates, depth, and physical-qubit footprint. Finally we describe how to adapt the Eastinthillation factory to produce the T states required for the QFT stage of Shor's algorithm in Sec. XII E 4.

TABLE VII: Resource and performance summary for one Eastinthillation factory.

Metric	Value
CCZ injection depth	147.48 POCs
Per-attempt restart rate	$\lesssim 15.94\%$
Average depth per accepted state	555.75 POCs
Number of factories for a continuous stream	4
Physical-qubit footprint per factory	319
Logical error rate	$\lesssim 9.36 \times 10^{-10}$

1. 0-Level T-State Distillation

We prepare each pair of $|H\rangle$ states using the protocol of Dasu et al. [79]. The protocol uses Goto's arbitrary-state encoder [80] to encode the states in the C6 code before joint logical-Hadamard verification and syndrome extraction. The two verified $|H\rangle$ states remain encoded

and are consumed through the dual-T-state injection circuit of the Walking Cat architecture [1]. The joint verification measures $H_{\mathrm{L},0}H_{\mathrm{L},1}$ using a two-qubit Bell state. A flagged readout followed by C6 syndrome extraction makes this measurement fault-tolerant.

The preparation circuit for two logical $|H\rangle$ states in the C6 code is shown in Fig. 17. As a three-ring code, we may implement the C6 code's SEC circuit with cyclic shifts, but the encoding circuit's connectivity cannot be achieved with just cyclic shifts. We instead propose using a short sequence of column swaps to implement the encoding circuit. Separate loss and leakage checks are unnecessary for this error-detecting code block because every qubit is destructively measured with loss-and-leakage detection during either state preparation or consumption.

We simulated the state-preparation, verification, and SEC circuit using the statevector method in Qiskit Aer. The logical error and rejection rates in Table VIII are Monte Carlo estimates with 10^9 shots using the swap loss noise model at $p = 10^{-4}$. The simulator appends ideal, destructive measurements of the two logical Hadamard operators $H_{\mathrm{L},0}$ and $H_{\mathrm{L},1}$ to estimate logical fidelity. The results of this simulation are included in Table VIII, which also reports the depth and physical-qubit footprint of the circuit.

TABLE VIII: Performance estimates and resource totals for one zero-level preparation of an encoded $|H\rangle^{\otimes 2}$ pair using the swap-loss noise model at $p = 10^{-4}$.

Quantity	Symbol	Value
Logical error rate per accepted pair	$p_{\mathrm{L}}^{(0)}$	1.30×10^{-8}
Rejection rate	$r_{\mathrm{rej}}^{(0)}$	0.358%
C6 SEC rejection rate	$r_{\mathrm{SEC}}^{\mathrm{C6}}$	0.323%
Pair preparation and verification depth	$T_{H^2}^{(0)}$	6.6 POCs
SEC depth	$T_{\mathrm{SEC}}^{(0)}$	5.4 POCs
Total depth	$T_{\mathrm{dist}}^{(0)}$	12.0 POCs
Physical-qubit footprint	$n_{\mathrm{phys}}^{(0)}$	12

2. *Eastinthillation factory circuit*

The Eastinthillation factory circuit, based on the Eastin Toffoli-state synthillation protocol [77], is shown in Fig. 18. The protocol uses eight $|H\rangle$ states from the zero-level factories based on the C6 code to implement two Margolus–Toffoli gates with shared controls and distinct targets. Let $p_{Y,\mathrm{inj}}$ denote the stochastic logical-Y fault probability per $|H\rangle$-state injection used to implement a logical $R_Y(\pi/4)$ rotation. A final "acceptance" parity check on the targets detects any single stochastic-Y input fault and yields a Toffoli state with error probability of order $p_{Y,\mathrm{inj}}^2$. We apply a Hadamard to the target qubit of the accepted Toffoli state. This final Clifford converts it into a CCZ resource state compatible with the depth-one CCZ gate of Sec. XII D.

We implement the Eastinthillation factory circuit in a single Q66 code block using the measurement-layer sequence shown in Fig. 19. The measurements use the LM1, LM2, and DM operations from the Walking Cat instruction set [1], mediated by EDM (error-detecting measurement) and Viterbi measurement schemes. We extend these schemes to include the Multi-Pauli error-corrected measurement operation (MECM) introduced by Webster and Delfosse [51].

Every cat-state-mediated measurement layer in Fig. 19 admits physical representatives whose supports on the GB66 block are pairwise disjoint, with each representative having weight at most 16. The complete representative certificates are given in appendix G. The relatively small width of these representatives allows us to use only 3 rounds of EDM for each double-$R_Y(\pi/4)$ injection to achieve a p_H of order 10^{-8} at physical error rate $p = 10^{-4}$. Similarly, we require only 5 rounds of Viterbi measurements to achieve logical error rates of order 10^{-13}. Each double-$R_Y(\pi/4)$ injection ends in a DMX – a destructive readout of all data qubits in the H6 block, an operation which takes only 1 POC to execute regardless of width.

$R_Y(\pi/4)$ injection. Each pair of $R_Y(\pi/4)$ rotations is implemented with the double-$R_Y(\pi/4)$ injection gadget of the Walking Cat architecture [1], which involves a joint YY measurement followed by a DMX operation on the C6 block. To avoid correlated input errors, each Q66 block has two dedicated C6 source blocks. A double-$R_Y(\pi/4)$ layer takes one logical qubit, prepared in $|H\rangle$, from each C6 block rather than taking both inputs from one encoded $|H\rangle^{\otimes 2}$ pair. The terminal DMX consumes both C6 blocks, including the unused logical qubit in each block. The two blocks are then reinitialized and their next encoded $|H\rangle^{\otimes 2}$ pairs are prepared and verified in parallel for the same Q66 block. We use dedicated C6 blocks rather than sharing them among multiple Q66 blocks. This choice simplifies the factory analysis by eliminating contention for shared resources.

Within either source C6 block, the two logical-Y operators have representatives

$$\overline{Y}_0 = \texttt{YIYIYI}, \qquad \overline{Y}_1 = \texttt{IYIYIY}. \tag{12}$$

These representatives have disjoint supports and weight three. Nevertheless, either operator can be measured using a two-qubit cat-state segment (a Bell state) by reusing one cat qubit for two data interactions. This reuse does not reduce the measurement distance. A fault on the reused cat qubit can propagate to two C6 data qubits. We choose the data-qubit support of each cat-state qubit so that no resulting hook error is a nontrivial C6 logical operator. Every such error has a nonzero syndrome and is rejected by the following SEC.

Combining the two-qubit C6 segment with a Q66 representative of weight at most 16 gives a weight-18 cat

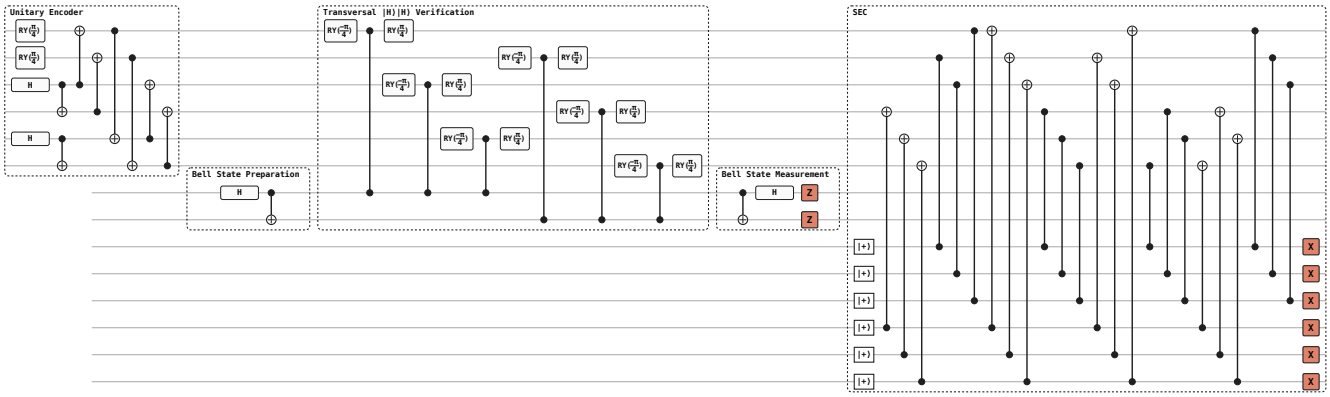


FIG. 17: Preparation of two logical $|H\rangle$ states in the C6 code.

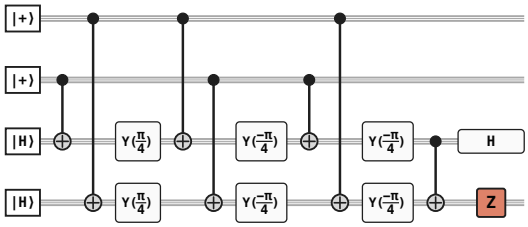


FIG. 18: The Eastinthillation factory circuit.

state for the joint $Y_{\mathrm{C6}}Y_{\mathrm{Q66}}$ observable in the double-$R_Y(\pi/4)$ injection.

Each Q66 cat-state factory contains two cat-state registers and one verification bank, all provisioned at width 16. Increasing one factory's width to 18 therefore requires $3(18-16) = 6$ additional qubits. We keep a six-qubit *C6 Bell-state source* next to each C6 block. For C6 state verification, two of these qubits form the Bell state described above. During the joint $Y_{\mathrm{C6}}Y_{\mathrm{Q66}}$ measurement, however, all six qubits are loaned to the weight 16 Q66 cat-state factory so it may create a weight 18 cat state. These two operations do not overlap, so the same C6 Bell-state source can support both.

The logical measurement still contains 19 data–cat interactions, so the logical-measurement error estimate below uses effective width $w_{\mathrm{eff}} = 19$.

At $w_{\mathrm{eff}} = 19$ and $p = 10^{-4}$, the fitted EDM model of Ref. [1] gives the single-round outcome-flip probability

$$p_{\mathrm{flip}} \simeq 2.1 w_{\mathrm{eff}} p = 3.99 \times 10^{-3}. \tag{13}$$

It takes three EDM rounds to suppress the measurement-induced error to the same order of magnitude as the state preparation error (p_{H^2}), since $p_{\mathrm{flip}}^2 = 1.59 \times 10^{-5}$ whereas $p_{\mathrm{flip}}^3 = 6.35 \times 10^{-8}$. We therefore use EDM-3 for each $R_Y(\pi/4)$ injection. Its logical-outcome error rate is

$$p_{\mathrm{EDM3}} \simeq p_{\mathrm{flip}}^3 = 6.35 \times 10^{-8}. \tag{14}$$

An EDM-3 outcome sequence is accepted when its three outcomes are all correct or all flipped. The two inputs to a double injection come from the two dedicated C6 blocks of the same Eastinthillation factory. Each input undergoes three C6 SECs, and the factory accepts only if both C6 blocks accept all three. Using the common C6 SEC rejection rate $r_{\mathrm{SEC}}^{\mathrm{C6}} = 0.323\%$, the per-injection acceptance factor is therefore

$$a_{R_Y} = \left[(1-p_{\mathrm{flip}})^3 + p_{\mathrm{flip}}^3\right](1 - r_{\mathrm{SEC}}^{\mathrm{C6}})^3 = 0.978534. \tag{15}$$

The corresponding rejection rate is $r_{R_Y} = 1 - a_{R_Y} = 2.147\%$. The factor $a_{R_Y}^8$ accounts for the eight EDM-3 outcome sequences and the 24 C6 SECs required by one CCZ preparation attempt.

The parity check at the end of the Eastinthillation factory can only detect single-logical-qubit Y or X errors; and all errors in the synthillation protocol can effectively be traced to $R_Y(\pi/4)$ injections. So we would like to ensure that $R_Y(\pi/4)$ injection errors are only either Y- or X-type errors. We rely on the injected-Y noise model of the Walking Cat architecture [1]: conditioned on the EDM and SEC acceptance tests, an injection fault is represented by a stochastic logical Y on the corresponding Q66 qubit. This follows from analyzing the conditional corrections of the logical measurements involved in the $R_Y(\pi/4)$ injection. A wrong terminal DMX result erroneously applies a Y correction. A wrong EDM result applies a $Z_{\mathrm{C6}}R_Y(\pi/2)_{\mathrm{Q66}}$ error; the extra Z_{C6} triggers the subsequent C6 X readout and therefore also applies a Y error via DMX Y correction (so, at the very least, is always correlated with a Y error). The residual Q66 operation is $R_Y(\pi/2)Y \doteq R_Y(-\pi/2)$, or the inverse rotation. The source $|H\rangle$ state has zero Y expectation. The two outcomes of the joint $Y_{\mathrm{C6}}Y_{\mathrm{Q66}}$ measurement, and hence the two residual rotations, are equiprobable. Averaging the corresponding channels gives

$$\frac{1}{2}\mathcal{R}_{Y,\pi/2} + \frac{1}{2}\mathcal{R}_{Y,-\pi/2} = \frac{1}{2}\mathcal{I} + \frac{1}{2}\mathcal{Y}, \tag{16}$$

where $\mathcal{R}_{Y,\theta}$ is the channel induced by $R_Y(\theta)$ and $\mathcal{Y}$ is conjugation by logical Y. We therefore bound the stochastic

logical-Y fault rate by the full EDM-3 logical-outcome error probability.

The two input states in each injection layer have independent preparation errors because they come from distinct C6 blocks. Because faults at these locations are sampled independently, the preparation errors are also independent. Consequently, for a single injection, the marginal stochastic logical-Y probability is bounded by the sum of the preparation and EDM-3 error probabilities:

$$p_{Y,\mathrm{inj}} \lesssim p_{H^2} + p_{\mathrm{EDM3}} = 7.65 \times 10^{-8}. \tag{17}$$

Acceptance Measurement. The acceptance measurement uses a LM1 operation mediated by Viterbi measurement. Its representative has weight 16, so $w_{\mathrm{cat}} = 16$, as shown in Table XVI. At $p = 10^{-4}$, the single-round bit-flip model of Ref. [1] gives

$$p_{\mathrm{flip},16} = 2.1 w_{\mathrm{cat}} p = 2.1 \times 16 \times 10^{-4} = 3.36 \times 10^{-3}. \tag{18}$$

For target error $\varepsilon = 10^{-11}$, the Viterbi stopping condition requires vote margin

$$d_{16} = \left\lceil \frac{\log[(1-\varepsilon)/\varepsilon]}{\log[(1-p_{\mathrm{flip},16})/p_{\mathrm{flip},16}]} \right\rceil = 5. \tag{19}$$

The logical error is therefore

$$p_{\mathrm{Vit},16} = \left[1 + \left(\frac{1-p_{\mathrm{flip},16}}{p_{\mathrm{flip},16}} \right)^5 \right]^{-1} = 4.36 \times 10^{-13}. \tag{20}$$

The cat-state rejection model of Ref. [1] gives $p_{\mathrm{miss},16} = 5 w_{\mathrm{cat}} p = 8.0 \times 10^{-3}$. Writing $\rho_{16} = p_{\mathrm{flip},16}/(1-p_{\mathrm{flip},16})$, the average duration is

$$\tau^{\mathrm{avg}}_{\mathrm{Vit},16} = \frac{1}{1-p_{\mathrm{miss},16}} \frac{d_{16}}{1-2p_{\mathrm{flip},16}} \frac{1-\rho_{16}^{d_{16}}}{1+\rho_{16}^{d_{16}}} = 5.07 \text{ SEC}. \tag{21}$$

CCZ **Injection.** The three CCZ injection measurements follow the protocol of Sec. XII D. The largest joint representative has weight 36, with weight 20 from the Q102 logical Z operator and weight 16 from the Q66 logical Z operator. We therefore use $w_{\mathrm{cat}} = 36$ for all three measurements to deduce an upper bound on the logical error of the measurements. At $p = 10^{-4}$, the single-round bit-flip probability is

$$p_{\mathrm{flip},36} = 2.1 \times 36 \times 10^{-4} = 7.56 \times 10^{-3}. \tag{22}$$

For target error $\varepsilon = 10^{-10}$, the Viterbi stopping condition requires vote margin

$$d_{36} = \left\lceil \frac{\log[(1-\varepsilon)/\varepsilon]}{\log[(1-p_{\mathrm{flip},36})/p_{\mathrm{flip},36}]} \right\rceil = 5. \tag{23}$$

The logical error per measurement is therefore

$$p_{\mathrm{Vit},36} = \left[1 + \left(\frac{1-p_{\mathrm{flip},36}}{p_{\mathrm{flip},36}} \right)^5 \right]^{-1} = 2.57 \times 10^{-11}. \tag{24}$$

The cat-state rejection model of Ref. [1] gives $p_{\mathrm{miss},36} = 5 w_{\mathrm{cat}} p = 1.8 \times 10^{-2}$. Writing $\rho_{36} = p_{\mathrm{flip},36}/(1-p_{\mathrm{flip},36})$, the average duration of one Viterbi measurement is

$$\tau^{\mathrm{avg}}_{\mathrm{Vit},36} = \frac{1}{1-p_{\mathrm{miss},36}} \frac{d_{36}}{1-2p_{\mathrm{flip},36}} \times \frac{1-\rho_{36}^{d_{36}}}{1+\rho_{36}^{d_{36}}} = 5.16982 \text{ SEC}. \tag{25}$$

The three measurements run in parallel, so their duration is set by the slowest measurement. Let $F(t)$ be the probability that one weight-36 Viterbi measurement has halted by SEC t. We obtain $F(t)$ by iterating the same vote-margin process used above. A cat-state attempt is rejected with probability $p_{\mathrm{miss},36}$; it then yields no measurement outcome, so the vote margin is unchanged in that SEC. Since the three measurements use independent cat states, the probability that all three have halted by SEC t is $F(t)^3$. Summing the probability that at least one measurement is still running after SEC t gives

$$\tau^{\mathrm{avg}}_{\mathrm{CCZ}} = \sum_{t=0}^{\infty} \left[1 - F(t)^3 \right] = 5.46218 \text{ SEC}. \tag{26}$$

This is only 0.29 SEC longer than the mean duration of one measurement. A union bound gives a logical-error contribution of at most $3p_{\mathrm{Vit},36} = 7.70 \times 10^{-11}$ from the three measurements.

Phase Fixups. After the three CCZ injection measurements, we measure the three commuting resource-block observables P_0, P_1, P_2 obtained by conjugating the logical-X operators by the tracked Clifford frame. This frame includes one conditional Clifford correction from each of the eight $R_Y(\pi/4)$ injections, so we analyze all $2^8 = 256$ frame branches.

We call a destructive readout that measures each physical data qubit in an independently chosen Pauli basis a *generalized destructive measurement* (DMG). A single DMG measures logical Pauli observables with pairwise-disjoint physical representatives in parallel: on the support of each representative, the physical qubits are measured in the corresponding single-qubit Pauli bases. A DMG takes only one POC–less than a full SEC.

Pairwise-disjoint representatives of all three P_i are not available on every branch. When they are available, a single DMG measures the three P_i in parallel. If exactly one pair overlaps, we measure one member of that pair using a single Viterbi measurement. We then measure the other two observables, whose representatives are disjoint, in parallel with a single DMG. If every pair overlaps, we measure products of the P_i that admit disjoint representatives. In both cases requiring nondestructive logical measurement layers, only one such layer precedes the terminal DMG. The phase fixups therefore require at most one measurement layer.

We describe the MECM and Viterbi cases using the scheduler-matrix convention of Ref. [51]: for $G \in \mathbb{F}_2^{3\times 3}$,

column j specifies the observable

$$Q_j = \prod_{i=0}^{2} P_i^{G_{ij}}. \tag{27}$$

If the eigenvalues of P_i and Q_j are $(-1)^{u_i}$ and $(-1)^{v_j}$, respectively, then

$$v = uG, \qquad u = vG^{-1}. \tag{28}$$

An invertible G therefore replaces the P_i with an equivalent set of logical observables Q_j. Measuring the Q_j gives the same logical information because u can be recovered from v, and it requires no additional measurement layer.

Only two scheduler matrices are required:

$$G_{\rm id} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad G_{\rm prod} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{29}$$

$G_{\rm id}$ measures $Q_i = P_i$ – it is no different from an ordinary Viterbi measurement. On the 32 branches whose original generators ordinarily require three measurement layers, $G_{\rm prod}$ replaces P_2 with the equivalent generator P_0P_2, so that it measures

$$(Q_0, Q_1, Q_2) = (P_0, P_1, P_0P_2). \tag{30}$$

Since $G_{\rm prod}^{-1} = G_{\rm prod}$, the original outcomes are

$$u_0 = v_0, \qquad u_1 = v_1, \qquad u_2 = v_0 + v_2 \pmod 2. \tag{31}$$

.

Table IX gives the resulting schedule for all 256 frame branches. On 32 branches, the three Q_j share a physical product basis and are obtained in parallel with a single DMG. On each of the other 224 branches, one Q_j is measured by a Viterbi measurement before the remaining pair is obtained in parallel with a single DMG. On the branches using $Q_2 = P_0P_2$, Q_1 and Q_2 have compatible, disjoint physical representatives even though the three original generators cannot be partitioned into fewer than three compatible layers. All Q_j commute, so the Viterbi measurement layer can be measured before the DMG layer. Physical representatives for the MECM and DMG layers are given in Tables XVIII to XX.

Because only one Q_j is measured via MECM on each two-layer branch, its decoder reduces to the single-bit Viterbi stopping rule. As established in appendix G, the physical representative has weight at most 16, so its outcome-flip probability is $p_{\rm flip,16} = 3.36 \times 10^{-3}$. For target error $\varepsilon = 10^{-10}$, the required vote margin is

$$d_{\rm MECM} = \left\lceil \frac{\log[(1-\varepsilon)/\varepsilon]}{\log[(1-p_{\rm flip,16})/p_{\rm flip,16}]} \right\rceil = 5. \tag{32}$$

The logical-outcome error contribution from the MECM is

$$p_{\rm MECM,16} = \left[1 + \left(\frac{1-p_{\rm flip,16}}{p_{\rm flip,16}}\right)^5\right]^{-1} = 4.36 \times 10^{-13}. \tag{33}$$

TABLE IX: Final-X scheduler and chronological layer partition for all 256 frame branches. A dash denotes an absent MECM layer.

Branch class	Count	Scheduler	Viterbi layer	DMG layer
One layer	32	$G_{\rm id}$	—	Q_0, Q_1, Q_2
Pair 01	76	$G_{\rm id}$	Q_2	Q_0, Q_1
Pair 02	68	$G_{\rm id}$	Q_1	Q_0, Q_2
Pair 12	48	$G_{\rm id}$	Q_0	Q_1, Q_2
$Q_2 = P_0P_2$	32	$G_{\rm prod}$	Q_0	Q_1, Q_2

Using $p_{\rm miss,16} = 8.0\times10^{-3}$ and $\rho_{16} = p_{\rm flip,16}/(1-p_{\rm flip,16})$, the average number of SECs on a branch requiring MECM is

$$N_{\rm SEC}^{\rm MECM} = \frac{1}{1-p_{\rm miss,16}} \frac{d_{\rm MECM}}{1-2p_{\rm flip,16}} \frac{1-\rho_{16}^{d_{\rm MECM}}}{1+\rho_{16}^{d_{\rm MECM}}} = 5.07. \tag{34}$$

Averaging over the frame branches gives

$$\overline{N}_{\rm SEC}^{\rm phase} = \frac{256-32}{256} N_{\rm SEC}^{\rm MECM} = 4.44. \tag{35}$$

The corresponding branch-averaged MECM error contribution is $(224/256)p_{\rm MECM,16} = 3.81 \times 10^{-13}$.

3. Analysis

Per-attempt failure rate. A CCZ preparation attempt accepts only if its eight $R_Y(\pi/4)$ injections, and final input-parity check all accept. From eq. (15), the rejection probability of one injection is $r_{R_Y} = 1 - a_{R_Y} = 2.147\%$. The final parity check rejects an odd number of injection faults, with probability

$$r_{\rm par} = \sum_{j \text{ odd}} \binom{8}{j} p_{Y,\rm inj}^j (1-p_{Y,\rm inj})^{8-j} \tag{36}$$

$$= \frac{1-(1-2p_{Y,\rm inj})^8}{2} \lesssim 6.12 \times 10^{-7}. \tag{37}$$

The per-attempt acceptance and failure probabilities therefore satisfy

$$1 - p_{\rm fail}^{\rm Eastin} \gtrsim (1-r_{R_Y})^8 (1-r_{\rm par}) = 84.06\%,$$
$$p_{\rm fail}^{\rm Eastin} \lesssim 15.94\%. \tag{38}$$

Factory depth. We first compute $T_{\rm prep}^{(0)}$, the POC depth required to prepare one CCZ state when the attempt detects no failure and the factory does not restart. The superscript (0) denotes this no-restart case. We obtain this depth by counting the Q66 SECs in the preparation schedule and converting that count to POCs. Preparing the Q66 resource block requires one logical zero preparation (LZ), as defined in the Walking Cat instruction set [1], four sequential pairs of $R_Y(\pi/4)$ injections, and the final acceptance measurement. LZ takes

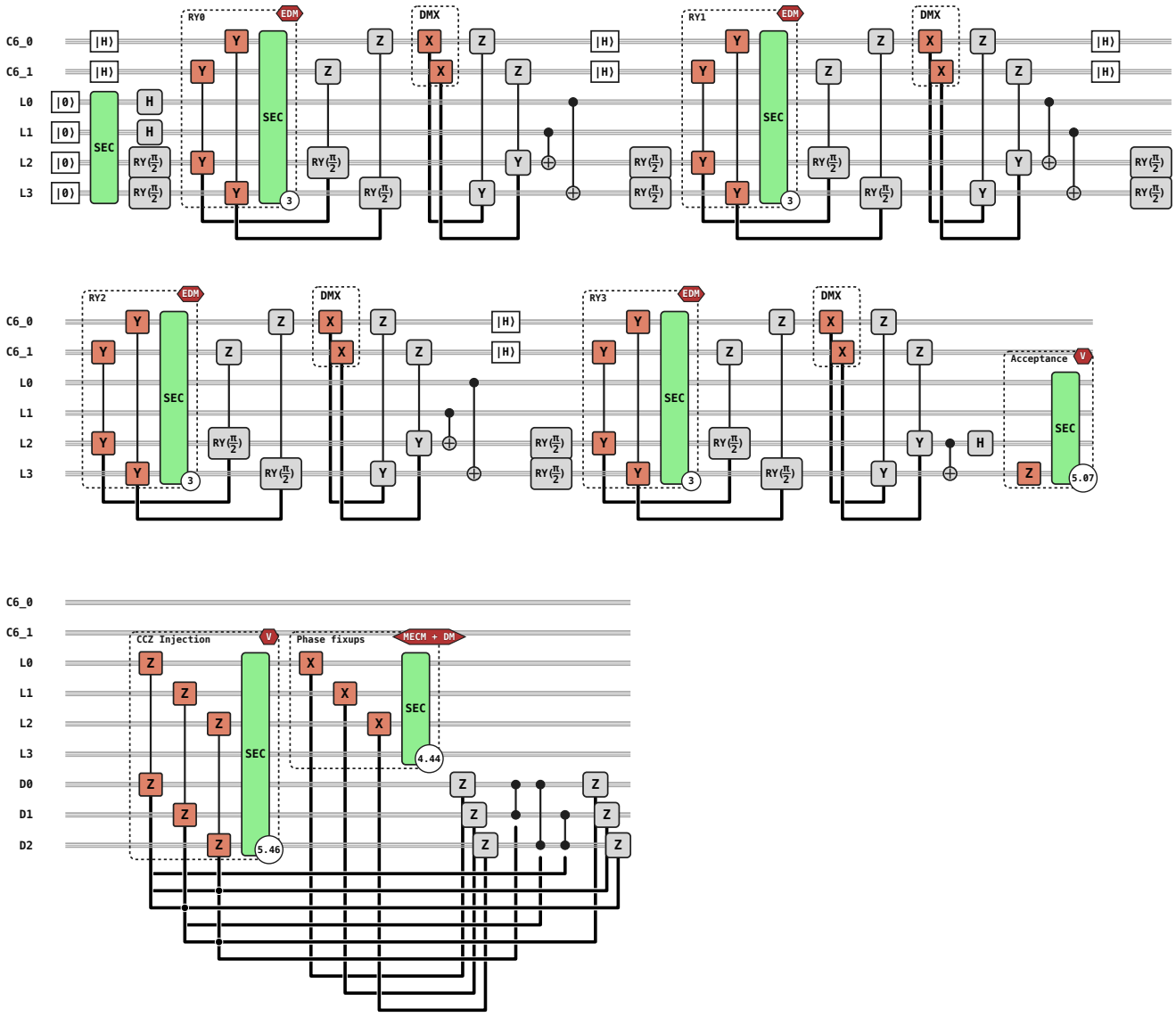


FIG. 19: Physical measurement schedule for the Eastinthillation factory. $C6_0$ and $C6_1$ are logical qubits drawn from the two distinct C6 code blocks dedicated to this factory's Q66 block.

one Q66 SEC. Each pair of rotations is performed in parallel by EDM-3 and therefore takes three Q66 SECs. Once the final C6 SEC finishes, we prepare the input states for the next pair of $R_Y(\pi/4)$ injections in parallel with the final Q66 SEC. This preparation takes only 12.0 POCs and therefore finishes before the next Q66 SEC begins. $\tau^{\text{avg}}_{\text{Vit},16} = 5.07$ SECs of the acceptance measurement, the Q66 block is occupied for an average of 18.07 Q66 SECs during preparation. Finally, as a Q66 SEC has POC depth $T^{\text{Q66}}_{\text{SEC}} = 17$:

$$\begin{aligned} N^{\text{prep}}_{\text{SEC}} &= 1 + 4 \times 3 + 5.07 = 18.07, \\ T^{(0)}_{\text{prep}} &= N^{\text{prep}}_{\text{SEC}} T^{\text{Q66}}_{\text{SEC}} \\ &= 18.07 \text{ SEC} \times 17 \, \frac{\text{POCs}}{\text{SEC}} \\ &= 307.19 \text{ POCs}. \end{aligned} \tag{39}$$

The three CCZ injection measurements are performed in parallel. Their joint average duration is $\tau^{\text{avg}}_{\text{CCZ}} \approx 5.46$ Q102 SECs from eq. (26).

$$T_{\text{inj}} = \tau^{\text{avg}}_{\text{CCZ}} T^{\text{Q102}}_{\text{SEC}} = 147.48 \text{ POCs}. \tag{40}$$

The phase-fixup schedule uses an average of 4.44 Q66 SECs, followed by the one-POC terminal DMG. Its av-

erage depth is

$$T_{\text{phase}} = \overline{N}_{\text{SEC}}^{\text{phase}} T_{\text{SEC}}^{\text{Q66}} + 1 = 4.44 \times 17 + 1 = 76.48 \text{ POCs}. \tag{41}$$

The depth from LZ through release of the Q66 block–assuming no restarts–is therefore

$$T_{\text{factory}}^{(0)} = T_{\text{prep}}^{(0)} + T_{\text{inj}} + T_{\text{phase}} = 531.15 \text{ POCs}. \tag{42}$$

Next, we account for the depth added from restarting the protocol in the event of a detected error. Each of the four $R_Y(\pi/4)$-injection layers performs two EDM-3 measurements in parallel. An error may be detected after each of the three associated SECs or after the final acceptance measurement. Let $k \in \{1, 2, 3, 4\}$ index the injection layer and let $j \in \{1, 2, 3\}$ index its SEC. The accumulated depth after SEC j of layer k is

$$d_{k,j} = 1 + 3(k-1) + j. \tag{43}$$

The three injection checkpoints therefore occur at depths $(2, 3, 4)$, $(5, 6, 7)$, $(8, 9, 10)$, and $(11, 12, 13)$ Q66 SECs for layers one through four. The final acceptance measurement reports a failure at depth

$$d_{\text{par}} = 1 + 4 \times 3 + \tau_{\text{Vit},16}^{\text{avg}} = 18.07 \text{ SECs}. \tag{44}$$

Next, we calculate the error-detection probability after each SEC of the two parallel EDM-3 measurements in a $R_Y(\pi/4)$-injection layer. After the first SEC, a restart occurs only if either C6 SEC reports an error. After the second and third SECs, a restart occurs if either C6 SEC reports an error or if the EDM outcomes have a parity mismatch. We write the latter two cases as disjoint events:

$$\begin{aligned}\Pr(\text{restart at SEC}) = {} & \Pr(\text{C6 SEC error}) \\ & + \Pr(\text{C6 SECs accept}) \\ & \times \Pr(\text{EDM mismatch}).\end{aligned} \tag{45}$$

An EDM mismatch cannot be detected after the first SEC because each EDM-3 sequence has only one outcome at that point. The second and third SECs can detect a disagreement among the accumulated outcomes.

Let $f = p_{\text{flip}} = 3.99 \times 10^{-3}$. One EDM-3 sequence agrees through m SECs when all m outcomes are correct or all m outcomes are flipped. The probability that both parallel EDM-3 sequences agree through $m \in \{2, 3\}$ SECs is

$$\begin{aligned} u_m &= \left[(1-f)^m + f^m\right]^2, \\ u_2 &= 0.984167, \qquad u_3 = 0.976298. \end{aligned} \tag{46}$$

The probability that both C6 SECs in one round accept is

$$c = (1 - r_{\text{SEC}}^{\text{C6}})^2 = 0.993550, \qquad 1 - c = 0.00644957. \tag{47}$$

Here $1 - c$ is the probability that at least one C6 SEC reports an error. Conditional on reaching a given SEC, the restart probabilities after the first, second, and third SECs are

$$\begin{aligned} e_1 &= 1 - c, \\ e_2 &= (1-c) + c(1-u_2) = 1 - cu_2, \\ e_3 &= (1-c) + c\left(1 - \frac{u_3}{u_2}\right) = 1 - c\frac{u_3}{u_2}. \end{aligned} \tag{48}$$

The first term in each expression is the C6 SEC rejection probability. The second term in e_2 and e_3 is the probability that both C6 SECs accept but the EDM outcomes disagree.

Each injection layer requires two zero-level input-pair preparations. Both preparations accept with probability $h = (1 - r_{\text{rej}}^{(0)})^2$. If either preparation rejects, the parallel preparation batch is repeated. The first batch has depth $T_{\text{dist}}^{(0)} = 12$ POCs and is hidden inside the 17-POC Q66 scheduling window. Each additional batch adds 12/17 Q66 SECs. For the geometrically distributed number of batches, the mean additional critical-path depth per reached injection layer is

$$\delta_{H^2} = \frac{12}{17}\left(\frac{1}{h} - 1\right) = 0.00508 \text{ Q66 SECs}. \tag{49}$$

So, a C6 rejection delays the injection layer but does not require a full-fledged factory restart. Once both inputs are available, a parallel injection layer completes with probability

$$a = c^3 u_3 = a_{R_Y}^2 = 0.957529. \tag{50}$$

Conditional on starting the parallel EDM-3 measurements, the probabilities that the attempt terminates after the first, second, or third SEC are

$$\begin{aligned} g_1 &= e_1, \\ g_2 &= ce_2, \\ g_3 &= c^2 u_2 e_3. \end{aligned} \tag{51}$$

These probabilities satisfy $\sum_{j=1}^{3} g_j = 1 - a$. Layer k is reached with probability a^{k-1}, so the unconditional restart probability at checkpoint j of that layer is $a^{k-1} g_j$.

The acceptance measurement is reached with probability a^4 and detects an injection fault with probability r_{par}. Its unconditional restart probability is

$$q_{\text{par}} = a^4 r_{\text{par}} = 5.14 \times 10^{-7}. \tag{52}$$

An attempt succeeds with probability

$$A = a^4(1 - r_{\text{par}}) = 0.840636. \tag{53}$$

The expected number of failures at each checkpoint per accepted state is the unconditional checkpoint probability divided by A. The average preparation depth is there-

fore

$$\overline{N}^{\text{prep}}_{\text{SEC}} = N^{\text{prep}}_{\text{SEC}} + \frac{1}{A}\left[\sum_{k=1}^{4} a^{k-1}\left(\delta_{H^2} + \sum_{j=1}^{3} g_j d_{k,j}\right) + q_{\text{par}} d_{\text{par}}\right]$$
$$= 18.07 + 1.44730 = 19.5173 \text{ SECs},$$
$$\overline{T}_{\text{prep}} = \overline{N}^{\text{prep}}_{\text{SEC}} T^{\text{Q66}}_{\text{SEC}} = 331.79 \text{ POCs}. \quad (54)$$

The CCZ injection and phase-fixup stages are executed only after the parity check accepts. Their Viterbi repetition costs are already included in eqs. (40) and (41). The complete average depth per accepted and consumed CCZ state is consequently

$$\overline{T}_{\text{factory}} = \overline{T}_{\text{prep}} + T_{\text{inj}} + T_{\text{phase}} = 555.75 \text{ POCs}. \quad (55)$$

Factory footprint. We say that a number of factories is sufficient to produce a continuous stream of CCZ states if, collectively, their average time to produce a magic state is shorter than the average time per state injection. The point-addition circuit executes at most one Toffoli gate at a time. A Q102 memory block occupies the CCZ injection interface for $\tau^{\text{avg}}_{\text{CCZ}} \approx 5.46$ Q102 SECs and can therefore consume at most one CCZ state every $T_{\text{inj}} = 147.48$ POCs. Since one factory supplies an accepted state every 555.75 POCs on average, the number of factories required to maintain this consumption rate is

$$N_{\text{CCZ}} = \left\lceil \frac{\overline{T}_{\text{factory}}}{\tau^{\text{avg}}_{\text{CCZ}} T^{\text{Q102}}_{\text{SEC}}} \right\rceil = \left\lceil \frac{555.75}{147.48} \right\rceil = 4. \quad (56)$$

Each Eastinthillation factory uses one Q66 block, three weight-16 cat-state factories, two dedicated zero-level C6 source blocks, and one six-qubit C6 Bell-state source per C6 block. As derived in appendix E 1, this gives 319 physical qubits per factory and 1,276 physical qubits for all four factories. The resulting layout is shown in Fig. 20.

Logical Error Rate. Eastin's exact output-error formula (assuming only Y- or X-type errors from $R_Y(\pi/4)$ injections) gives $28p^2_{Y,\text{inj}}$. Using eq. (17) gives

$$p^{\text{Eastin}}_{\text{L,in}} \lesssim 28 p^2_{Y,\text{inj}} \lesssim 28\left(7.65 \times 10^{-8}\right)^2 = 1.64 \times 10^{-13}. \quad (57)$$

The Q66 logical error rate is $p^{\text{Q66}}_{\text{L,SEC}} = 2.63 \times 10^{-11}$ per SEC. The Q66 block is exposed for $\overline{N}^{\text{prep}}_{\text{SEC}}$ Q66 SECs during preparation. During CCZ injection, one Q102 SEC lasts $T^{\text{Q102}}_{\text{SEC}}/T^{\text{Q66}}_{\text{SEC}} = 27/17$ Q66 SECs, and the phase-fixup schedule adds $\overline{N}^{\text{phase}}_{\text{SEC}}$ Q66 SECs. A union bound over the input-error, acceptance-measurement, CCZ injection, phase-fixup, and Q66 SEC contributions gives

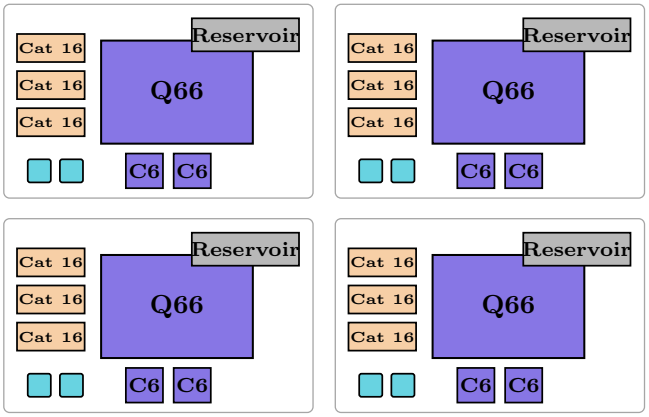


FIG. 20: Layout of the four Eastinthillation factories required to sustain the application consumption rate. Each Q66 block is adjacent to three weight-16 cat-state factories and has two dedicated zero-level C6 T-state factories. The two cyan sites at the lower left of each factory are its local six-qubit C6 Bell-state sources.

$$p^{\text{Eastin}}_{\text{L}} \lesssim p^{\text{Eastin}}_{\text{L,in}} + p_{\text{Vit,16}} + 3p_{\text{Vit,36}} + \frac{224}{256} p_{\text{MECM,16}} + \left(\overline{N}^{\text{prep}}_{\text{SEC}} + \tau^{\text{avg}}_{\text{CCZ}} \frac{T^{\text{Q102}}_{\text{SEC}}}{T^{\text{Q66}}_{\text{SEC}}} + \overline{N}^{\text{phase}}_{\text{SEC}}\right) p^{\text{Q66}}_{\text{L,SEC}} \quad (58)$$
$$= 7.80 \times 10^{-11} + \left(19.5173 + \tau^{\text{avg}}_{\text{CCZ}} \frac{27}{17} + 4.44\right)\left(2.63 \times 10^{-11}\right)$$
$$= 9.36 \times 10^{-10}. \quad (59)$$

Under the error model above, the logical error rate satisfies

$$p^{\text{Eastin}}_{\text{L}} \lesssim 9.36 \times 10^{-10}. \quad (60)$$

Q66 memory errors give the largest contribution to this bound by two orders of magnitude; the scheme is otherwise capable of producing CCZ gates with logical error rate of order 1e-13. This floor can be met by higher-distance code or a decoder that uses loss and leakage information, e.g. [81]. Or, one may exploit soft information in the decoder: the factory can reject and restart low-confidence attempts before their outputs are used [82, 83].

4. *T-state factory conversion*

The windowed semiclassical iQFT requires arbitrary-angle Z-rotations, which can be synthesized approximately with T gates using the mixing+fallback

method [84]. We only need 16 Z-rotations per window if we execute them sequentially, feedforward measurement results and calculate the angles from the accumulated measurement outcomes. The total number of Z-rotations across all iQFTs is 464, with $0.57\log_2(1/\varepsilon)+8.83$ T gates per rotation, where ε is the approximation error per rotation.

However, the Eastinthillation factory can prepare only CCZ states. We can repurpose its components to produce T states instead. The local C6 T-state factories inject the required T states into the Q66 blocks, where they can be safely stored in memory. The states are then teleported to the Q102 blocks as required. Using the measured logical error rate of 1.30×10^{-8} per accepted pair, and minimizing for the total diamond-norm distance, we obtain the optimal $\varepsilon = 5.30 \times 10^{-9}$ and the average T-sequence length 24.5. This results in an expected total of 5,684 required T pairs.

This gives the total iQFT-stage failure probability, by the union bound, at most

$$p_{\mathrm{iQFT}} \leq 5{,}684\left(1.30 \times 10^{-8}\right) + 464\left(5.30 \times 10^{-9}\right) = \\ = 7.64 \times 10^{-5}. \tag{61}$$

Part VI
Conclusion

We presented optimized quantum circuits for solving the ECDLP on `secp256k1` using Shor's algorithm, derived rigorous bounds on the success probability of the implementation—taking into account phase- and output-errors due to approximate arithmetic—and compiled our implementation to our application-specific extension of the Walking Cat architecture [1] using our compilation toolchain, allowing us to accurately account for the overheads due to layout and routing. By combining quantum-circuit, compilation, and architecture improvements, we showed that a trapped-ion quantum computer with 20,000 qubits should be able to solve the ECDLP on `secp256k1` within 26 days. Because we intentionally prioritize simple, uniform implementations over exhaustive co-optimization, we expect future work to improve the runtime, physical-qubit footprint, and success probability reported here.

In our resource estimates, we have taken into account hardware details that have typically been ignored in the literature. This enables us to be more confident about our runtime and footprint estimates. For instance, one of the major issues with trapped-ion architectures (and one we prominently addressed in the original Walking Cat architecture) is a loss cascade. When an ion is lost, if its well is merged with that of a second ion to perform a two-qubit gate, the electrode configuration will not be correct for optimal trapping. This can result in substantial heating of the remaining ion, which can lead to ejection from the trap. The leakage and loss reduction units (LLRUs) developed in that architecture assume that this ejection happens 100% of the time. However, for sufficiently deep traps, this might not be the case. Here, we use a different loss propagation model in which the ion is not ejected but its final position after a split is randomized, and we develop a new LLRU to address this model of loss propagation. We expect that the physical situation is more nuanced, with heating and ejection considered using a detailed microarchitecture, but we leave this to future work.

Several of our techniques could also reduce the resource requirements of other fault-tolerant quantum algorithms, e.g., for chemistry, condensed matter physics, and optimization [9, 14]. In particular, the Clifford frame clearing using logical CliNR and our integrated routing can be included in the general-purpose Walking Cat architecture without any additional hardware requirements. The increased logical-measurement parallelism obtained using non-overlapping cat-based measurements is straightforward to include in the Walking Cat architecture at the price of adjusting the block allocation to allow for increased cat state throughput in the vicinity of the target blocks. Lastly, our fast CCZ implementation provides a 31× speedup over a CCZ compiled into Clifford and T gates. We leave it to future work to quantify the impact of these optimizations on quantum computing applications beyond the ECDLP.

ACKNOWLEDGMENTS

The authors thank Chris Ballance, Tom Harty, and Matt Keesan for discussions and support. MR thanks Christof Zalka for friendship and inspiration and Franziska Burkhalter for discussions.

The authors acknowledge the use of generative AI tools to assist with code and figure generation, to explore different proof strategies, and to review and proofread the manuscript. The manuscript was written entirely by the authors, who take full responsibility for its content.

[1] F. Tripier, W. C. Chung, J. Young, S. Alam, B. Bjork, A. Brodutch, F. L. Buessen, N. J. Coble, T. Dellaert, D. Maslov, M. Roetteler, E. Tham, M. Webster, M. Ye, J. Gamble, A. Maksymov, J. P. Marceaux, and N. Delfosse, Fault-tolerant quantum computing with trapped ions: The walking cat architecture, arXiv preprint arXiv:2604.19481 (2026).

[2] A. Schrottenloher, Optimized point addition circuits

for elliptic curve discrete logarithms, arXiv preprint arXiv:2606.02235 (2026).
[3] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM Journal on Computing **26**, 1484 (1997).
[4] V. von Burg, G. H. Low, T. Häner, D. S. Steiger, M. Reiher, M. Roetteler, and M. Troyer, Quantum computing enhanced computational catalysis, Physical Review Research **3**, 033055 (2021).
[5] J. Lee, D. W. Berry, C. Gidney, W. J. Huggins, J. R. McClean, N. Wiebe, and R. Babbush, Even more efficient quantum computations of chemistry through tensor hypercontraction, PRX Quantum **2**, 030305 (2021).
[6] L. Zhao, J. J. Goings, W. Aboumrad, A. Arrasmith, L. Calderin, S. Churchill, D. Gabay, T. Harvey-Brown, M. Hiles, M. Kaja, M. Keesan, K. Kulesz, A. Maksymov, M. Maruo, M. Muñoz, B. Nijholt, R. Schiller, Y. de Sereville, A. Smidutz, F. Tripier, G. Yao, T. Zaveri, C. Collins, M. Roetteler, E. Epifanovsky, A. Kovyrshin, L. Tornberg, A. Broo, J. R. Hammond, Z. Chandani, P. Khalate, E. Kyoseva, Y.-T. Chen, E. M. Kessler, C. Y.-Y. Lin, G. Ramu, R. Shaffer, M. Brett, B. Huang, M. R. Hugues, and T. Y. Takeshita, Quantum-classical Auxiliary Field Quantum Monte Carlo with matchgate shadows on trapped ion quantum computers, Physical Review Research **8**, 033061 (2026), see also arxiv.org preprint https://arxiv.org/abs/2506.22408.
[7] H. Liu, G. H. Low, D. S. Steiger, T. Häner, M. Reiher, and M. Troyer, Prospects of quantum computing for molecular sciences, Materials Theory **6**, 11 (2022).
[8] M. E. Beverland, P. Murali, M. Troyer, K. M. Svore, T. Hoefler, V. Kliuchnikov, G. H. Low, M. Soeken, A. Sundaram, and A. Vaschillo, Assessing requirements to scale to practical quantum advantage, arXiv preprint arXiv:2211.07629 (2022).
[9] A. M. Dalzell, S. McArdle, M. Berta, P. Bienias, C.-F. Chen, A. Gilyén, C. T. Hann, M. J. Kastoryano, E. T. Khabiboulline, A. Kubica, G. Salton, S. Wang, and F. G. S. L. Brandão, *Quantum algorithms: A survey of applications and end-to-end complexities* (Cambridge University Press, 2025).
[10] J. Iaconis, S. Ray, S. Sekwao, C. Girotto, and M. Roetteler, Practical Quantum Topological Data Analysis with applications to high-dimensional feature extraction and time series analysis (2026), arXiv:2607.27206.
[11] S. Ray, J. Jojo, J. Iaconis, A. Gnanasekaran, A. Tiwari, M. Roetteler, C. Hill, and J. Pathak, Quantum Lattice Boltzmann solutions for transport under 3D spatially varying advection on trapped ion hardware, in *Proceedings IEEE Quantum Week (QCE'26)* (2026) accepted for publication. See also arxiv.org preprint https://arxiv.org/abs/2604.28121.
[12] J. Chen and G. K. Chan, A framework for robust quantum speedups in practical correlated electronic structure and dynamics, arXiv preprint arXiv:2508.15765 (2025).
[13] D. Zhu, M. A. Lopez-Ruiz, F.-H. Rouet, C. Girotto, W. Aboumrad, R. Lucas, A. Kaushik, and M. Roetteler, End-to-end performance of quantum-accelerated large-scale linear algebra workflows, in *Proceedings IEEE Quantum Week (QCE'26)* (2026) accepted for publication. See also arxiv.org preprint https://arxiv.org/abs/2603.15515.
[14] R. Babbush, R. King, S. Boixo, W. Huggins, T. Khattar, G. H. Low, J. R. McClean, T. O'Brien, and N. C. Rubin, The grand challenge of quantum applications, arXiv preprint arXiv:2511.09124 (2025).
[15] Y. Alexeev, D. Bacon, K. R. Brown, R. Calderbank, L. D. Carr, F. T. Chong, B. DeMarco, D. Englund, E. Farhi, B. Fefferman, A. V. Gorshkov, A. Houck, J. Kim, S. Kimmel, M. Lange, S. Lloyd, M. D. Lukin, D. Maslov, P. Maunz, C. Monroe, J. Preskill, M. Roetteler, M. Savage, and J. Thompson, Quantum computer systems for scientific discovery, PRX Quantum **2**, 017001 (2021).
[16] M. Ekerå and J. Håstad, Quantum algorithms for computing short discrete logarithms and factoring rsa integers, in *International Workshop on Post-Quantum Cryptography* (Springer, 2017) pp. 347–363.
[17] A. May and L. Schlieper, Quantum period finding is compression robust, arXiv preprint arXiv:1905.10074 (2019).
[18] C. Gidney and M. Ekerå, How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits, Quantum **5**, 433 (2021).
[19] C. Chevignard, P.-A. Fouque, and A. Schrottenloher, Reducing the number of qubits in quantum factoring, in *Annual International Cryptology Conference* (Springer, 2025) pp. 384–415.
[20] N. C. Jones, R. Van Meter, A. G. Fowler, P. L. McMahon, J. Kim, T. D. Ladd, and Y. Yamamoto, Layered architecture for quantum computing, Physical Review X **2**, 031007 (2012).
[21] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, Surface codes: Towards practical large-scale quantum computation, Physical Review A **86**, 032324 (2012).
[22] C. Gidney, How to factor 2048 bit RSA integers with less than a million noisy qubits, arXiv preprint arXiv:2505.15917 (2025).
[23] P. Webster, L. Berent, O. Chandra, E. T. Hockings, N. Baspin, F. Thomsen, S. C. Smith, and L. Z. Cohen, The pinnacle architecture: Reducing the cost of breaking RSA-2048 to 100 000 physical qubits using quantum LDPC codes, arXiv preprint arXiv:2602.11457 (2026).
[24] J. Proos and C. Zalka, Shor's discrete logarithm quantum algorithm for elliptic curves, arXiv preprint quant-ph/0301141 (2003).
[25] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter, Quantum resource estimates for computing elliptic curve discrete logarithms, in *Asiacrypt 2017* (Springer, 2017) pp. 241–270.
[26] T. Häner, S. Jaques, M. Naehrig, M. Roetteler, and M. Soeken, Improved quantum circuits for elliptic curve discrete logarithms, in *PQCrypto 2020* (Springer, 2020) pp. 425–444.
[27] D. Litinski, How to compute a 256-bit elliptic curve private key with only 50 million Toffoli gates, arXiv preprint arXiv:2306.08585 (2023).
[28] R. Babbush, A. Zalcman, C. Gidney, M. Broughton, T. Khattar, H. Neven, T. Bergamaschi, J. Drake, and D. Boneh, Securing elliptic curve cryptocurrencies against quantum vulnerabilities: Resource estimates and mitigations, arXiv preprint arXiv:2603.28846 (2026).
[29] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, A new quantum ripple-carry addition circuit (2004) arXiv:quant-ph/0410184.
[30] Y. Takahashi, S. Tani, and N. Kunihiro, Quantum addition circuits and unbounded fan-out, Quantum Information and Computation **10**, 872 (2010), arXiv:0910.2530.
[31] C. Chevignard, P.-A. Fouque, and A. Schrottenloher, Re-

ducing the number of qubits in quantum discrete logarithms on elliptic curves, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer, 2026) pp. 371–401.

[32] H. Luo, Z. Yang, J. Luo, Z. Wang, Y. Su, X. Sun, L. Li, and T. Li, Quantum algorithm for elliptic curve discrete logarithms with space-efficient point addition, arXiv preprint arXiv:2607.13816 (2026).

[33] M. Cain, Q. Xu, R. King, L. R. Picard, H. Levine, M. Endres, J. Preskill, H.-Y. Huang, and D. Bluvstein, Shor's algorithm is possible with as few as 10,000 reconfigurable atomic qubits, arXiv preprint arXiv:2603.28627 (2026).

[34] S. Baek, S.-H. Lee, D. Min, and J. Kim, Sdqc: Distributed quantum computing architecture utilizing entangled ion qubit shuttling, arXiv preprint arXiv:2512.02890 (2025).

[35] É. Gouzien, D. Ruiz, F.-M. Le Régent, J. Guillaud, and N. Sangouard, Performance analysis of a repetition cat code architecture: Computing 256-bit elliptic curve logarithm in 9 hours with 126 133 cat qubits, Physical Review Letters **131**, 040602 (2023).

[36] C. Gidney, Halving the cost of quantum addition, Quantum **2**, 74 (2018).

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

[38] J. I. Cirac and P. Zoller, Quantum computations with cold trapped ions, Physical review letters **74**, 4091 (1995).

[39] C. Monroe, D. M. Meekhof, B. E. King, W. M. Itano, and D. J. Wineland, Demonstration of a fundamental quantum logic gate, Physical review letters **75**, 4714 (1995).

[40] D. Kielpinski, C. Monroe, and D. J. Wineland, Architecture for a large-scale ion-trap quantum computer, Nature **417**, 709 (2002).

[41] C. Monroe, R. Raussendorf, A. Ruthven, K. R. Brown, P. Maunz, L.-M. Duan, and J. Kim, Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects, Physical Review A **89**, 022317 (2014).

[42] C. D. Bruzewicz, J. Chiaverini, R. McConnell, and J. M. Sage, Trapped-ion quantum computing: Progress and challenges, Applied physics reviews **6** (2019).

[43] M. Malinowski, D. Allcock, and C. Ballance, How to wire a 1000-qubit trapped-ion quantum computer, PRX quantum **4**, 040313 (2023).

[44] N. P. Breuckmann and J. N. Eberhardt, Quantum low-density parity-check codes, PRX quantum **2**, 040101 (2021).

[45] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, Topological quantum memory, Journal of Mathematical Physics **43**, 4452 (2002).

[46] A. C. Hughes, R. Srinivas, C. Löschnauer, H. Knaack, R. Matt, C. Ballance, M. Malinowski, T. Harty, and R. Sutherland, Trapped-ion two-qubit gates with¿ 99.99% fidelity without ground-state cooling, arXiv preprint arXiv:2510.17286 (2025).

[47] C. Löschnauer, J. Mosca Toba, A. Hughes, S. King, M. Weber, R. Srinivas, R. Matt, R. Nourshargh, D. Allcock, C. Ballance, *et al.*, Scalable, high-fidelity all-electronic control of trapped-ion qubits, PRX Quantum **6**, 040313 (2025).

[48] P. Selinger, Quantum circuits of t-depth one, Physical Review A—Atomic, Molecular, and Optical Physics **87**, 042302 (2013).

[49] C. Jones, Low-overhead constructions for the fault-tolerant Toffoli gate, Physical Review A **87**, 022328 (2013).

[50] D. Maslov, On the advantages of using relative phase Toffolis with an application to multiple control Toffoli optimization, Physical Review A **93**, 022311 (2016).

[51] M. Webster and N. Delfosse, Fast logical operations in quantum LDPC codes using simple resource states, arXiv preprint arXiv:2607.16166 (2026).

[52] T. Khattar, N. Shutty, C. Gidney, A. Zalcman, N. Yosri, D. Maslov, R. Babbush, and S. P. Jordan, Verifiable quantum advantage via optimized DQI circuits, arXiv preprint arXiv:2510.10967 (2025).

[53] A. A. Karatsuba and Y. P. Ofman, Multiplication of many-digital numbers by automatic computers, in *Doklady Akademii Nauk*, Vol. 145 (Russian Academy of Sciences, 1962) pp. 293–294.

[54] D. Litinski, Quantum schoolbook multiplication with fewer Toffoli gates, arXiv preprint arXiv:2410.00899 (2024).

[55] E. Bernstein and U. Vazirani, Quantum complexity theory, in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing* (1993) pp. 11–20.

[56] M. Mosca, *Quantum computer algorithms*, Ph.D. thesis, University of Oxford. 1999. (1999).

[57] R. B. Griffiths and C.-S. Niu, Semiclassical Fourier transform for quantum computation, Physical Review Letters **76**, 3228 (1996).

[58] Y. R. Sanders, D. W. Berry, P. C. Costa, L. W. Tessler, N. Wiebe, C. Gidney, H. Neven, and R. Babbush, Compilation of fault-tolerant quantum heuristics for combinatorial optimization, PRX Quantum **1**, 020312 (2020).

[59] D. Wecker, M. B. Hastings, N. Wiebe, B. K. Clark, C. Nayak, and M. Troyer, Solving strongly correlated electron models on a quantum computer, Physical Review A **92**, 062318 (2015).

[60] M. P. Harrigan, T. Khattar, C. Yuan, A. Peduri, N. Yosri, F. D. Malone, R. Babbush, and N. C. Rubin, Expressing and analyzing quantum algorithms with Qualtran (2024), arXiv:2409.04643 [quant-ph].

[61] M. P. Harrigan, T. Khattar, C. Yuan, A. Peduri, N. Yosri, F. D. Malone, R. Babbush, and N. C. Rubin, Qualtran (2026).

[62] PsiQuantum, Quantum resource estimation format (QREF) (2024), open format for representing quantum algorithms for resource estimation.

[63] PsiQuantum, QREF: Quantum resource estimation format, `https://psiq.github.io/qref/` (2024), JSON-based domain-specific format for representing quantum algorithms and resource estimation workflows.

[64] D. S. Steiger, T. Häner, and M. Troyer, ProjectQ: An open source software framework for quantum computing, Quantum **2**, 49 (2018).

[65] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, MLIR: Scaling compiler infrastructure for domain specific computation, in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (IEEE, 2021) pp. 2–14, arXiv:2002.11054.

[66] T. G. Draper, S. A. Kutin, E. M. Rains, and K. M. Svore, A logarithmic-depth quantum carry-lookahead

adder (2004), arXiv:quant-ph/0406142.
[67] M. Remaud and V. Vandaele, Ancilla-free quantum adder with sublinear depth, in *Reversible Computation (RC 2025)*, Lecture Notes in Computer Science, Vol. 15716 (Springer, 2025) pp. 137–154, arXiv:2501.16802.
[68] T. Häner, V. Kliuchnikov, M. Roetteler, and M. Soeken, Space-time optimized table lookup (2022), arXiv:2211.01133 [quant-ph].
[69] T. Häner, D. S. Steiger, M. Smelyanskiy, and M. Troyer, High performance emulation of quantum circuits, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016).
[70] E. Knill, Quantum computing with realistically noisy devices, Nature **434**, 39 (2005), arXiv:quant-ph/0410199.
[71] G. Baranes, M. Cain, J. P. Bonilla Ataides, D. Bluvstein, J. Sinclair, V. Vuletic, H. Zhou, and M. D. Lukin, Leveraging qubit loss detection in fault tolerant quantum algorithms, Physical Review X **16**, 011002 (2026).
[72] J. Preskill, Reliable quantum computers, Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences **454**, 385 (1998).
[73] C. Gidney, Stim: a fast stabilizer circuit simulator, Quantum **5**, 497 (2021).
[74] M. Ye, D. Wecker, and N. Delfosse, Beam search decoder for quantum low-density parity-check codes, PRX Quantum **7**, 033002 (2026).
[75] P. W. Shor, Fault-tolerant quantum computation, in *Proceedings of the 37th Annual Symposium on Foundations of Computer Science* (IEEE, 1996) pp. 56–65, arXiv:quant-ph/9605011.
[76] X. Zhou, D. W. Leung, and I. L. Chuang, Methodology for quantum logic gate construction, Physical Review A **62**, 052316 (2000), arXiv:quant-ph/0002039.
[77] B. Eastin, Distilling one-qubit magic states into Toffoli states, Physical Review A **87**, 032321 (2013).
[78] E. T. Campbell and M. Howard, Unified framework for magic state distillation and multiqubit gate synthesis with reduced resource cost, Physical Review A **95**, 022316 (2017).
[79] S. Dasu, S. Burton, K. Mayer, D. Amaro, J. A. Gerber, K. Gilmore, D. Gresh, D. DelVento, A. C. Potter, and D. Hayes, Breaking even with magic: Demonstration of a high-fidelity logical non-Clifford gate, arXiv preprint arXiv:2506.14688 (2025).
[80] H. Goto, Step-by-step magic state encoding for efficient fault-tolerant quantum computation, Scientific Reports **4**, 7501 (2014).
[81] P. Liu, S. J. S. Tan, E. Huang, U. A. Acar, H. Zhou, and C. Zhao, Achieving optimal-distance atom-loss correction via Pauli envelope, arXiv preprint arXiv:2603.04156 (2026).
[82] H. Bombín, M. Pant, S. Roberts, and K. I. Seetharam, Fault-tolerant postselection for low-overhead magic state preparation, PRX Quantum **5**, 010302 (2024).
[83] S. C. Smith, B. J. Brown, and S. D. Bartlett, Mitigating errors in logical qubits, Communications Physics **7**, 386 (2024).
[84] V. Kliuchnikov, K. Lauter, R. Minko, A. Paetznick, and C. Petit, Shorter quantum circuits via single-qubit gate approximation, Quantum **7**, 1208 (2023).
[85] C. J. Clopper and E. S. Pearson, The use of confidence or fiducial limits illustrated in the case of the binomial, Biometrika **26**, 404 (1934).
[86] Z. J. Wall, J. D. Piel, S. R. Vizvary, M. Bareian, S. Diaz, E. Mossman, A. Ransford, C. H. Greene, E. R. Hudson, and W. C. Campbell, Barium autoionization for efficient ion trap loading, Physical Review A **114**, 013102 (2026).
[87] N.-C. Chiu, E. C. Trapp, J. Guo, M. H. Abobeih, L. M. Stewart, S. Hollerith, P. L. Stroganov, M. Kalinowski, A. A. Geim, S. J. Evered, S. H. Li, X. Lyu, L. M. Peters, D. Bluvstein, T. T. Wang, M. Greiner, V. Vuletić, and M. D. Lukin, Continuous operation of a coherent 3,000-qubit system, Nature **646**, 1075 (2025).

Part VII
Appendix

Appendix A: Algorithms

1. Affine elliptic curve point addition

This section contains integer-valued representations of the algorithms that comprise the affine elliptic curve point addition circuits in our implementation. Algorithm 4 shows the classical precomputation of the initial accumulator point and the lookup tables for each window in the point addition loop shown in Algorithm 5. There are $L = \lceil n/w \rceil$ windows of size w with the top window possibly containing less than w bits. After sampling the random initialization point $[a]P$ by sampling a uniform random $a \in \mathbb{Z}_r^*$, Algorithm 4 first computes the L tables with multiples of the base point P, then the L tables with multiples of the point Q of which we seek to compute the discrete logarithm with respect to base P. It also computes and stores $3 \cdot x$ for the x-coordinates of all points in the table.

Algorithm 4: Randomized window setup

Input: $P, Q \in E(\mathbb{F}_p)$ of prime order r; width $w \geq 1$ with $2^w < r$
Output: Accumulator A_0 and classical augmented tables $(T_i)_{i=0}^{2L-1}$

$a_0 \xleftarrow{\$} \mathbb{Z}_r^*$; $A_0 \leftarrow [a_0]P$;
for $i \leftarrow 0$ **to** $L-1$ **do**
  $\mu_i \xleftarrow{\$} \mathbb{Z}_r \setminus \{-j2^{iw} \bmod r : 0 \leq j < 2^w\}$;
  for $j \leftarrow 0$ **to** $2^w - 1$ **do**
    $(x_{i,j}, y_{i,j}) \leftarrow [j2^{iw} + \mu_i]P$;
    $T_i[j] \leftarrow (x_{i,j}, y_{i,j}, 3x_{i,j})$;
for $i \leftarrow L$ **to** $2L-1$ **do**
  $\mu_i \xleftarrow{\$} \mathbb{Z}_r \setminus \{-j2^{(i-L)w} \bmod r : 0 \leq j < 2^w\}$;
  for $j \leftarrow 0$ **to** $2^w - 1$ **do**
    $(x_{i,j}, y_{i,j}) \leftarrow [j2^{(i-L)w} + \mu_i]Q$;
    $T_i[j] \leftarrow (x_{i,j}, y_{i,j}, 3x_{i,j})$;
return $A_0, (T_i)_{i=0}^{2L-1}$

Algorithm 5 is a simple loop over all $2L$ windows that just adds the respective table points. Note that the scalars k and l are decomposed into their windowed 2^w-ary representations

$$k = \sum_{i=0}^{L-1} k_i 2^{iw}, \quad l = \sum_{i=0}^{L-1} l_i 2^{iw},$$

and the digits k_i and l_i are used to look up the correct table points that are passed to the affine addition AffineAdd depicted in Algorithm 6. In practice, we use $w = 16$ and we only perform 28 of the 32 point additions as described in [27]: We replace the first point addition by a table lookup and drop the final 3 windowed

Algorithm 5: Windowed double-scalar multiplication

Input: Registers k, l; accumulator $A = A_0$ and tables $(T_i)_{i=0}^{2L-1}$ from Algorithm 4
Output: k, l unchanged; updated accumulator A

for $i \leftarrow 0$ **to** $L-1$ **do**
  $A \leftarrow \mathsf{AffineAdd}(A; k_i, T_i)$;
for $i \leftarrow L$ **to** $2L-1$ **do**
  $A \leftarrow \mathsf{AffineAdd}(A; l_{i-L}, T_i)$;
return k, l, A

point additions in favor of extended classical postprocessing [37]. We start with the w most-significant bits of the first scalar and move toward the least-significant bit in each step, implementing the semi-classical quantum Fourier transform in blocks of size w [57].

The affine addition algorithm AffineAdd uses the following subroutines. BinaryGCDRecord consumes its input and retains the compressed Dialog/add-sub record. BinaryGCDInvMul and BinaryGCDMul replay it to compute the in-place division or in-place multiplication, respectively, and BinaryGCDUnrecord restores the input and clears all record scratch. The two replays use modular halvings and doublings, respectively.

The notation $\mathsf{Lookup}_{(x,y)}$ or Lookup_{3x} loads the indicated coordinates, while $m \leftarrow \mathsf{XClear}(u)$ measures and clears a lookup register and records its X-basis outcome. The final LookupPhaseFix applies the deferred lookup correction. Displayed modular additions, subtractions, and the square-subtract use the approximate circuits. The square-subtract circuit is the one-split Karatsuba version specific for `secp256k1`. Note that we classically sample a uniform random mask $R \in \mathbb{F}_p$ and add it with an approximate quantum-classical adder to the x-register before the lookup of the x-coordinate multiple. This randomizes the accumulator register in the square-sub circuit. The mask is removed after the subtraction of the square by an approximate addition of $-R \bmod p$. The separate addition step $x \leftarrow (x + R) \bmod p$ can be omitted if a fresh offset R_i is sampled for the i-th window in Algorithm 4 during the generation of the lookup tables. The table would then contain $3x_{i,j} + R_i$ instead of $3x_{i,j}$ and the masked value would be looked up, making the separate addition unnecessary.

2. Modular subtraction of a square

This section provides explicit integer-valued algorithm descriptions for the operation that subtracts a square modulo p, i.e., $|x\rangle|y\rangle$ to $|x\rangle|(y - x^2) \bmod p\rangle$. The main algorithm is Algorithm 7. It uses $\mathsf{AddBMultiple}_{\pm}^{q}$ (Algorithm 10), $\mathsf{AddCMultiple}_{-}^{q}$ (Algorithm 11), and $\mathsf{AddSlice}_{\sigma}^{q}$ (Algorithm 8), which in turn uses $\mathsf{PhaseGE}^{q}$ (Algorithm 9) and is itself a component of AddBMultiple and AddCMultiple.

We treat input values as integers less than 2^n that

Algorithm 6: AffineAdd: affine addition with direct-value binary-gcd arithmetic

Input: Accumulator $(x, y) = P_1$; selector j; table T with $T[j] = (x_2, y_2, 3x_2)$ and $P_2 = (x_2, y_2) \neq \mathcal{O}$
Output: On success, $(x, y) = P_1 + P_2$; j unchanged

$(x_2, y_2) \leftarrow \mathsf{Lookup}_{(x,y)}(T, j)$;
$x \leftarrow (x - x_2) \bmod p$;
$m_x^{(1)} \leftarrow \mathsf{XClear}(x_2)$;
$y \leftarrow (y - y_2) \bmod p$;
$m_y^{(1)} \leftarrow \mathsf{XClear}(y_2)$;
$\rho \leftarrow \mathsf{BinaryGCDRecord}(x)$;
$y \leftarrow \mathsf{BinaryGCDInvMul}(\rho, y)$;
$x \leftarrow \mathsf{BinaryGCDUnrecord}(\rho)$;
$R \xleftarrow{\$} \mathbb{F}_p$;
$x \leftarrow (x + R) \bmod p$;
$x_2 \leftarrow \mathsf{Lookup}_{3x}(T, j)$;
$x \leftarrow (x + x_2) \bmod p$;
$m_x^{(2)} \leftarrow \mathsf{XClear}(x_2)$;
$x \leftarrow (x - y^2) \bmod p$;
$x \leftarrow (x + (-R \bmod p)) \bmod p$;
$\rho \leftarrow \mathsf{BinaryGCDRecord}(x)$;
$y \leftarrow \mathsf{BinaryGCDMul}(\rho, y)$;
$x \leftarrow \mathsf{BinaryGCDUnrecord}(\rho)$;
$(x_2, y_2) \leftarrow \mathsf{Lookup}_{(x,y)}(T, j)$;
$y \leftarrow (y - y_2) \bmod p$;
$x \leftarrow (x - x_2) \bmod p$;
$x \leftarrow -x \bmod p$;
$m_x^{(3)} \leftarrow \mathsf{XClear}(x_2)$;
$m_y^{(3)} \leftarrow \mathsf{XClear}(y_2)$;
$(m_x, m_y) \leftarrow (m_x^{(1)} \oplus m_x^{(3)}, m_y^{(1)} \oplus m_y^{(3)})$;
$\mathsf{LookupPhaseFix}(T, j; m_x, m_y, m_x^{(2)})$;
return x, y, j

are not necessarily canonical representatives less than p. Intermediate results are also integer values of their respective bit sizes. The phase ϕ is implicit in all algorithm signatures, except where it is displayed explicitly by PhaseGE. An assignment to ϕ records the phase acquired by the active computational-basis branch.

For the specific parameters of the curve `secp256k1`, $\mathsf{AddBMultiple}_{\pm}^{q}$ and $\mathsf{AddCMultiple}_{-}^{q}$ use the signed binary representation of c in the form of the list

$$T_c = ((32, +1), (10, +1), (6, -1), (4, +1), (0, +1)),$$
$$c = \sum_{(d,\epsilon)\in T_c} \epsilon 2^d = 2^{32} + 2^{10} - 2^6 + 2^4 + 1.$$

Besides the given curve parameters n and c, the carry-propagation width κ, the local carry-padding width ρ, and the phase-comparison width δ are positive integers that can be selected for different trade-offs. We assume that $\rho \geq 0$, $1 \leq \delta \leq \frac{n}{2}$, $\kappa \geq 1$, $2c \leq 2^\kappa \leq 2^n$, $32 + \frac{n}{2} + 2 + \rho < n$, $\kappa + \rho < n$ to ensure that all slice and window invocations in these operations are well-formed.

Algorithm 9: $\mathsf{PhaseGE}^{q}(x, y; \delta)$

Input: $x, y \in \mathbb{Z}$, $0 \leq x, y < 2^\delta$, $\delta \in \mathbb{Z}_{\geq 1}$; control $q \in \{0, 1\}$, phase $\phi \in \{+1, -1\}$
Output: $x, y, \phi \leftarrow x, y, (-1)^{q\mathbb{1}[x\geq y]}\phi$ (on failure-free execution)

$\chi \leftarrow \mathbb{1}[x \geq y]$ in a clean scratch qubit;
$\phi \leftarrow (-1)^{q\chi}\phi$;
$\chi \leftarrow 0$; `// uncompute the exact comparison`
return x, y, ϕ;

Algorithm 10: $\mathsf{AddBMultiple}_{\sigma}^{q}(s, v; n_s; \delta)$

Input: $s, v \in \mathbb{Z}$, $0 \leq s < 2^{n_s}$, $n_s \in \{n, n+2\}$, $0 \leq v < 2^n$; sign $\sigma \in \{+1, -1\}$; control $q \in \{0, 1\}$; requested phase-comparison width $\delta \in \mathbb{Z}_{\geq 1}$, $\delta \leq n/2$
Output: $s, v \leftarrow s, v'$, where $v' \in [0, 2^n)$, $v' \equiv v + q\sigma Bs \pmod p$ (on failure-free execution)

$s_L \equiv s_{[0,n/2)}$; $s_H \equiv s_{[n/2,n_s)}$
`// low and high register views of` s
$w_h \leftarrow n_s - n/2$;
$(s, v) \leftarrow \mathsf{AddSlice}_{\sigma}^{q}(s, v; n_s, 0, n/2, n/2; \delta)$
`// add` σBs_L`; wrap`
foreach $(d, \epsilon) \in T_c$ **do**
  $(s, v) \leftarrow \mathsf{AddSlice}_{\sigma\epsilon}^{q}(s, v; n_s, n/2, w_h, d; \delta)$
  `// add` $\sigma\epsilon 2^d s_H$`; local`
return s, v;

Algorithm 11: $\mathsf{AddCMultiple}_{\sigma}^{q}(s, v; \delta)$

Input: $s, v \in \mathbb{Z}$, $0 \leq s, v < 2^n$; sign $\sigma \in \{+1, -1\}$; control $q \in \{0, 1\}$; requested phase-comparison width $\delta \in \mathbb{Z}_{\geq 1}$, $\delta \leq n/2$
Output: $s, v \leftarrow s, v'$, where $v' \in [0, 2^n)$, $v' \equiv v + q\sigma cs \pmod p$ (on failure-free execution)

foreach $(d, \epsilon) \in T_c$ **do**
  $(s, v) \leftarrow \mathsf{AddSlice}_{\sigma\epsilon}^{q}(s, v; n, 0, n-d, d; \delta)$
  `// direct low part; wrap`
$g \leftarrow ch_s$ in a clean 65-bit register;
$(g, v) \leftarrow \mathsf{AddSlice}_{\sigma}^{q}(g, v; 65, 0, 65, 0; \delta)$
`// omitted high part; local`
$g \leftarrow 0$; `// uncompute` ch_s
return s, v;

Appendix B: Failure case analysis

As in essentially all previous work (see, e.g., [2, 24–28, 35]), the elliptic curve point addition circuits described here are approximate, which means there exist inputs for which the circuits fail to compute the correct result for reasons from two different categories. Failures in the first category occur when the input elliptic curve points constitute an exceptional case where the underlying addition formula fails. Failures in the second category are due to an approximate modular arithmetic compo-

Algorithm 7: One-level Karatsuba square-subtract

Input: $x, y \in \mathbb{Z}$, $0 \le x, y < p$, control $q \in \{0,1\}$; requested phase-comparison width $\delta \in \mathbb{Z}_{\ge 1}$, $\delta \le n/2$
Output: $x, y \leftarrow x, y'$, where $y' \in [0,p)$, $y' \equiv y - qx^2 \pmod p$ (on failure-free execution)

$x_L \equiv x_{[0,n/2)}$; $x_H \equiv x_{[n/2,n)}$; // register views; $x = Bx_H + x_L$
$v \leftarrow x_L^2$; // exact integer square in n-bit scratch
$(v,y) \leftarrow \mathsf{AddBMultiple}_{+}^{q}(v,y;n;\delta)$; // add Bx_L^2
$(v,y) \leftarrow \mathsf{AddSlice}_{-}^{q}(v,y;n,0,n,0;\delta)$; // add $-x_L^2$
$v \leftarrow 0$; // uncompute x_L^2
$v \leftarrow x_H^2$; // exact integer square in n-bit scratch
$(v,y) \leftarrow \mathsf{AddBMultiple}_{+}^{q}(v,y;n;\delta)$; // add Bx_H^2
$(v,y) \leftarrow \mathsf{AddCMultiple}_{-}^{q}(v,y;\delta)$; // add $-cx_H^2$
$v \leftarrow 0$; // uncompute x_H^2
$u \leftarrow 0$; $\widehat{x}_H \equiv x_H + uB$; // append one clean bit; no copy of x_H
$\widehat{x}_H \leftarrow \widehat{x}_H + x_L$; // overwrite the high half of x in place
$v \leftarrow \widehat{x}_H^2$; // exact integer square in $(n+2)$-bit scratch
$(v,y) \leftarrow \mathsf{AddBMultiple}_{-}^{q}(v,y;n+2;\delta)$; // add $-B\widehat{x}_H^2$
$v \leftarrow 0$; // uncompute $\widehat{x}_H^2$
$\widehat{x}_H \leftarrow \widehat{x}_H - x_L$; // restore x_H and clear u
release the clean bit u;
return x, y;

Algorithm 8: $\mathsf{AddSlice}_{\sigma}^{q}(s,v;n_s,o,w,t;\delta)$

Input: $s, v \in \mathbb{Z}$, $0 \le s < 2^{n_s}$, $n_s \in \mathbb{Z}_{\ge 1}$, $0 \le v < 2^n$;
$0 \le o < n_s$; $1 \le w \le n_s - o$; $0 \le t \le n - w$; sign $\sigma \in \{+1,-1\}$; requested phase-comparison width $\delta \in \mathbb{Z}_{\ge 1}$;
either $t + w + \rho < n$ or $t + w = n$ with $\delta \le w$; control $q \in \{0,1\}$
Output: $s, v \leftarrow s, v'$, where $v' \in [0, 2^n)$, $v' \equiv v + q\sigma 2^t s_{[o,o+w)} \pmod p$ (on failure-free execution)

$a \equiv s_{[o,o+w)}$;
if $\sigma = -1$ **then**
  $v_{[0,\kappa)} \leftarrow v_{[0,\kappa)} + c \pmod{2^\kappa}$; // add c in the low correction window
  $v \leftarrow (2^n - 1) - v$; // enter complemented orientation
if $t + w + \rho < n$ **then**
  $v_{[t,t+w+\rho)} \leftarrow v_{[t,t+w+\rho)} + qa \pmod{2^{w+\rho}}$; // zero-pad a by ρ bits
else
  $(v_{[t,n)}, h) \leftarrow v_{[t,n)} + qa$; // here $t + w = n$
  $v_{[0,\kappa)} \leftarrow v_{[0,\kappa)} + hc \pmod{2^\kappa}$; // fold $h2^n$ back as hc
  $m \leftarrow \mathsf{Measure}_X(h)$; // measure and clear the carry
  if $m = 1$ **then**
    $\phi \leftarrow (-1)^q \phi$; // exact measured-carry phase
    $\mathsf{PhaseGE}^q(v_{[n-\delta,n)}, a_{[w-\delta,w)}; \delta)$; // repair the comparison phase
if $\sigma = -1$ **then**
  $v_{[0,\kappa)} \leftarrow v_{[0,\kappa)} + c \pmod{2^\kappa}$; // add c in the low correction window
  $v \leftarrow (2^n - 1) - v$; // leave complemented orientation
return s, v;

nent failing to produce the exact intended result. This section shows that the overall number of failure cases occurring in our proposed randomized circuit for affine point addition is small enough to allow the full Shor algorithm to succeed with high probability. In particular, we show that for each pair of scalars (k, l), the probability that it leads to a failure case is bounded by a constant p_f independent of k and l.

1. Exceptional cases for elliptic curve point addition

As in most previous work, we use the affine point addition formula for the short Weierstrass model $E : y^2 = x^3 + ax + b$. Given $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, the point $P_3 = (x_3, y_3) = P_1 + P_2$ is computed as $x_3 = \lambda^2 - x_1 - x_2$ and $y_3 = (x_2 - x_3)\lambda - y_2$, where $\lambda = (y_2 - y_1)/(x_2 - x_1)$. This formula has exceptional cases. It cannot be used if $x_2 = x_1$, i.e., if $P_1 = \pm P_2$, and also if one of the points is the point at infinity $\mathcal{O}$. Because quantum circuits can only implement reversible operations, the slope value λ is often uncomputed from

the results by re-computing $\lambda = (y_2 + y_3)/(x_2 - x_3)$. This implies another exceptional case coming from the condition $x_3 = x_2$. It follows that $y_3 = \pm y_2$, hence this happens when the sum is equal to $\pm P_2$, so either $P_1 = \mathcal{O}$ or $P_1 = -[2]P_2$. In total, there are four exceptional cases for the point P_1 when we aim to add a point P_2, namely $P_1 \in \{\mathcal{O}, P_2, -P_2, -[2]P_2\}$. Note that Roetteler et al. missed this fourth exceptional case in their analysis of the exceptional case count in [25, Section 4.2].

Let A be the accumulator point represented by the quantum register that will hold the approximate superposition over (k, l) of $f(k, l) = [k]P + [l]Q$ after the computation. As already suggested in [24], we initialize it with a classically precomputed uniform random point $[a_0]P$, where a_0 is sampled uniformly at random with $a_0 \in \mathbb{Z}_r^* = \mathbb{Z}_r \setminus \{0\}$. Using this random offset allows us to avoid the exceptional case of adding a point to $\mathcal{O}$ and to reason about probabilities over the uniform random starting point. After the quantum double-scalar multiplication, the accumulator register holds an approximate superposition over k and l of $f(k, l) + [a_0]P$. Since a_0 is independent of (k, l), the constant shift has no impact on the Fourier-basis measurement that follows and does not have to be subtracted from A.

As described earlier, the double-scalar multiplication is computed via windowed addition of points looked up from precomputed tables of classical elliptic curve points. We fix a window size $w > 0$. The scalars are decomposed into windows of size w as $k = \sum_{i=0}^{L-1} k_i 2^{i \cdot w}$ and $l = \sum_{i=0}^{L-1} l_i 2^{i \cdot w}$ with $k_i, l_i \in \{0, 1, \ldots, 2^w - 1\}$. There are $2L$ separate tables of size 2^w, one for each window index $0 \leq i < 2L$ containing the precomputed classical multiples $[j2^{iw}]P$ for all $j \in \{0, 1, \ldots, 2^w - 1\}$ when $0 \leq i < L$, and $[j2^{(i-L)w}]Q$ when $L \leq i < 2L$. All tables contain the point $\mathcal{O}$ for the cases where one of the $k_i = 0$ or one of the $l_i = 0$. Usually, the special case when the lookup point is $\mathcal{O}$ needs to be treated separately to ensure that the point addition circuit does not change the accumulator, leading to operations that are controlled on the condition $k_i = 0$ (or $l_i = 0$). To avoid these controlled operations, we add a fixed classical masking point M_i to all points in the i-th table. During the classical table generation, the point is chosen uniformly at random via sampling a uniform random $\mu_i \in \mathbb{Z}_r$ and computing $M_i = [\mu_i]P$ if $0 \leq i < L$ (or $M_i = [\mu_i]Q$ if $L \leq i < 2L$) under the additional condition that none of $[j2^{iw}]P + M_i$ (or $[j2^{(i-L)w}]Q + M_i$) are equal to $\mathcal{O}$, i.e., we sample $\mu_i \in \mathbb{Z}_r \setminus \{-2^{iw} j \mid j \in \{0, 1, \ldots, 2^w - 1\}\}$ (or $\mu_i \in \mathbb{Z}_r \setminus \{-2^{(i-L)w} j \mid j \in \{0, 1, \ldots, 2^w - 1\}\}$). Since we assume that $2^w < r$, there exists a point M_i for each table such that none of the shifted points is $\mathcal{O}$. To summarize, the classical table T_i for looking up the point to be added into the accumulator for the i-th window is given as

$$T_i = \begin{cases} \{[j2^{iw} + \mu_i]P \mid j \in [0, 2^w)\}, & \text{if } 0 \leq i < L, \\ \{[j2^{(i-L)w} + \mu_i]Q \mid j \in [0, 2^w)\}, & \text{if } L \leq i < 2L. \end{cases}$$

Using these tables results in another additive shift for the oracle computation. Due to random initialization and table masking, after the double scalar multiplication, the accumulator register holds an approximation to the superposition over all (k, l) of

$$A = [k]P + [l]Q + ([a_0 + \mu_P]P + [\mu_Q]Q), \text{ where}$$
$$\mu_P = \sum_{i=0}^{L-1} \mu_i \text{ and } \mu_Q = \sum_{i=L}^{2L-1} \mu_i.$$

The shift $[a_0 + \mu_P]P + [\mu_Q]Q$ is independent of k and l and has no impact on the subsequent phase estimation step in Shor's algorithm. Therefore, it does not need to be subtracted or uncomputed. Let $\mu = (\mu_0, \mu_1, \ldots, \mu_{2L-1})$ and define

$$\begin{aligned} f_{a_0,\mu}(k, l) &= f(k, l) + [a_0 + \mu_P]P + [\mu_Q]Q \\ &= [k]P + [l]Q + [a_0 + \mu_P]P + [\mu_Q]Q. \end{aligned}$$

For detailed, integer-valued pseudo-code representations of these operations, see Algorithms 4, 5, and 6.

Next, we analyze when a pair (k, l) leads to an exceptional case during the windowed point addition. Let A_i denote the state of the accumulator point during an ideal, correct computation before the i-th windowed point addition, i.e. $A_0 = [a_0]P$, $A_1 = [a_0 + k_0 + \mu_0]P$, $A_L = [a_0 + k + \mu_P]P$, $A_{L+1} = [a_0 + k + \mu_P]P + [l_0 + \mu_L]Q$, etc. Note that for $i > 0$, the accumulator A_i can only assume the value $\mathcal{O}$ in a correct computation if, in the preceding step, A_{i-1} was equal to the negative of the added table point, a case already covered as an exceptional case for the earlier step. Moreover, our random offset ensures that $A_0 \neq \mathcal{O}$. This means that an exceptional point in an addition step that adds the table point C_i to A_i comes from one of the three cases $A_i \in \{C_i, -C_i, -[2]C_i\}$.

A pair (k, l) leads to an exceptional point addition if there exists an index $0 \leq i < 2L$ such that $A_i \in \{C_i, -C_i, -[2]C_i\}$, where $C_i = [k_i 2^{iw} + \mu_i]P$ if $0 \leq i < L$ and $C_i = [l_{i-L} 2^{(i-L)w} + \mu_i]Q$ if $L \leq i < 2L$. Define the set of all $a_0 \in \mathbb{Z}_r^*$ that lead to an exceptional case for given (k, l) as follows:

$$\mathcal{X}_{w,\mu}(k, l) = \{a_0 \in \mathbb{Z}_r^* \mid \exists i < 2L, A_i \in \{C_i, -C_i, -[2]C_i\}\}.$$

Because the map $a_0 \mapsto [a_0]P$ is a bijection $\mathbb{Z}_r \to \langle P \rangle$, each of $C_i, -C_i, -[2]C_i$ can only be reached by A_i for fixed (k, l) starting from a single value for a_0. Therefore, we have

$$|\mathcal{X}_{w,\mu}(k, l)| \leq 3(2L) = 6L.$$

Let $\bar{f}_{w,a_0,\mu}(k, l)$ denote the output of the w-windowed affine-addition circuit for $f_{a_0,\mu}(k, l)$ when every modular-arithmetic subroutine is exact. If $a_0 \notin \mathcal{X}_{w,\mu}(k, l)$, every addition along the ideal trajectory is non-exceptional. Correctness of the affine addition circuit on non-exceptional inputs and induction over the $2L$ additions therefore give $\bar{f}_{w,a_0,\mu}(k, l) = f_{a_0,\mu}(k, l)$. It follows that $\{a_0 \in \mathbb{Z}_r^* \mid \bar{f}_{w,a_0,\mu}(k, l) \neq f_{a_0,\mu}(k, l)\} \subseteq \mathcal{X}_{w,\mu}(k, l)$.

Lemma 1. *Let $n, w \in \mathbb{N}$, $1 \leq w < n$, $2^w < r$, $L = \lceil n/w \rceil$, $\mu = (\mu_0, \ldots, \mu_{2L-1}) \in \mathbb{Z}_r^{2L}$ a fixed mask vector chosen as described above. Let $(k, l) \in [0, 2^n) \times [0, 2^n)$ and let $\mathcal{X}_{w,\mu}(k, l)$ be the set of all $a_0 \in \mathbb{Z}_r^*$ that lead to an exceptional case in a quantum circuit for computing $f_{a_0,\mu}(k, l)$ using windowed point addition with masked precomputed tables and a non-zero initialization point $[a_0]P$ as described above. Then, over uniform random $a_0 \in \mathbb{Z}_r^*$, we have*

$$\Pr_{a_0 \in \mathbb{Z}_r^*}(\bar{f}_{w,a_0,\mu}(k, l) \neq f_{a_0,\mu}(k, l)) \leq \Pr_{a_0 \in \mathbb{Z}_r^*}(a_0 \in \mathcal{X}_{w,\mu}(k, l)) = \frac{|\mathcal{X}_{w,\mu}(k, l)|}{r - 1} \leq \frac{6L}{r - 1}.$$

For our concrete parameters of `secp256k1`, this bound is negligibly small, smaller than 2^{-248}.

In practice, we implement only $29 < 2L = 32$ windows (the first using a simple table lookup, 28 using windowed point additions), but the expression above still holds as a conservative upper bound for the failure probability due to exceptional cases.

2. Approximate modular arithmetic failures

The second category of failures occurs because the modular arithmetic operations that are used in the affine point addition circuit are approximate operations themselves [2]. Again, there exist inputs for which the circuits fail to produce the expected results.

The modular arithmetic circuits presented here make use of the special shape of the pseudo-Mersenne base field prime p. Several simplifications make modular arithmetic circuits more efficient but possibly lead to failures. For example, integer comparison with p for modular reduction is replaced by comparison of the most significant bits and the correction addition that depends on the outcome of the comparison truncates the carry propagation. These optimizations are applied to the modular reduction operations in modular addition, subtraction, doubling, and halving circuits [2]. Similarly, truncated comparisons during phase corrections after measurement-based carry uncomputation can lead to phase failures even if the resulting values are correct. Furthermore, Dialog-based in-place multiplication [2, 52] uses a non-worst-case bound for both the number of iterations and the register sizes used as the algorithm progresses [24]. We begin by discussing the optimizations in modular reduction.

a. Approximate modular reduction

An element in $\mathbb{F}_p$ is usually represented by its unique representative in $\{0, 1, \ldots, p-1\}$ (which we call the least non-negative or canonical representative). Integer arithmetic operations on such representatives may lead to values outside that range and a modular reduction is applied to reduce such a value back to its canonical representation. For example, taking two input operands from $\mathbb{F}_p$ and adding their standard integer representatives results in a value $0 \leq u < 2p - 1$. Modular reduction then subtracts p if $u \geq p$, which means $u \bmod p = u$, if $u < p$, and $u \bmod p = u - p$ if $u \geq p$. Thus, there are two computational components, namely, computing the integer comparison $u < p$ and depending on its outcome an integer subtraction $u - p$.

We work in the specific setting of a pseudo-Mersenne prime p of bitlength n. Let $c \in \mathbb{N}$ be odd, $c \ll 2^n$ such that $p = 2^n - c$. That p has n bits implies $2c \leq 2^n$. For the curve `secp256k1`, we have $n = 256$ and $c = 2^{32} + 2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 1 = 2^{32} + 977$. Subtracting such a p results in $u - p = u - (2^n - c) = u + c - 2^n$ and can be implemented by computing the addition $u + c$ and discarding the $(n + 1)$-th bit. Note that, if $u \geq p$, then $u + c \geq 2^n$ and its $(n + 1)$-th bit is always 1.

The two optimizations above, which were also used in previous work [2], lead to approximate modular reduction circuits that we use in different situations in our circuits. First, the full integer comparison $u < p$ is replaced by a comparison $u < 2^n$, which amounts to simply observing the $(n + 1)$-th bit of u. Instead of computing the integer comparison into a result bit that acts as the control for the modular subtraction, the $(n + 1)$-th bit directly serves as the control bit. This operation differs from exact integer comparison if

$$u \in [p, 2^n).$$

This means that for c of the possible results of an addition, the optimized reduction fails to produce a canonical representative. Note that the result can still be represented with n bits and is correct modulo p but is by p larger than the canonical representative. This behavior does not necessarily lead to a direct arithmetic failure; however, it might lead to failure cases when the larger result is used in further computations. We therefore treat non-canonical representatives for outputs of any operation as a failure case.

Second, for the addition $u + c$, we truncate carry propagation after $\kappa \leq n$ bits for $c < 2^\kappa$ as in [2]. This operation differs from exact addition with full carry propagation if the carry propagates further than κ bits, i.e.,

$$u \bmod 2^\kappa \in [2^\kappa - c, 2^\kappa).$$

This failure case, where the result is wrong modulo p, occurs for a proportion of $c/2^\kappa$ of input values, in which case the result after reduction is 2^κ smaller than the correct one.

Because of the simplified comparison and since our circuits work on n-bit quantum registers, we allow intermediate values modulo p to be represented by integers in $[0, 2^n)$. We assume throughout that $2c \leq 2^\kappa \leq 2^n$. For a

non-negative integer u, define

$$q_u = \lfloor u/2^n \rfloor, \quad r_u = u - q_u 2^n \in [0, 2^n),$$
$$w_u = \begin{cases} 1, \text{ if } (r_u \bmod 2^\kappa) \geq 2^\kappa - c, \\ 0, \text{ otherwise.} \end{cases}$$

If $u \in [0, 2^{n+1})$, $q_u \in \{0, 1\}$ is a single bit and the approximate reduction as described above then returns

$$\mathrm{red}_\kappa(u) = r_u + q_u c - q_u w_u 2^\kappa,$$

where q_u is the $(n+1)$-th bit, the correction is triggered if it is 1, in which case c is added to the first n bits of u and the result is 2^κ smaller than the correct value if the carry when adding c propagates out of the first κ bits. Since the $(n+1)$-th bit was discarded and the carry never propagates beyond the κ-th bit, $\mathrm{red}_\kappa(u) \in [0, 2^n)$. Using $2^n \equiv c \bmod p$ and taking modulo p gives

$$\mathrm{red}_\kappa(u) \equiv u - q_u w_u 2^\kappa \pmod{p}, \tag{B1}$$

which means that the approximate reduction red_κ returns a correct integer representative of u if and only if either no correction occurs ($q_u = 0$) or the carry from the addition of c does not propagate beyond κ bits ($w_u = 0$). The above two failure cases imply incorrect result values. Another type of failure leads to an incorrect phase.

b. *Approximate phase repair*

To uncompute certain bits that are computed in our circuits such as a carry bit b depending on a register $|x\rangle$, we use measurement in the X basis, which changes the phase by a factor $(-1)^{mb}$, where m is the measurement outcome. This means that a phase correction is required if $m = 1$ and $b = 1$. Our circuits use approximate recomputation of the bit b for example by truncating carry propagation or comparing only the higher order bits. If the approximation is incorrect, a phase failure occurs because the circuit fails to correct the phase flip.

We now proceed to deduce bounds on the failure probabilities for the various components in the approximate affine point addition circuit. We start with the simpler modular operations such as addition, subtraction, and negation, then discuss the modular subtraction of a square, and finally, the in-place modular division and multiplication circuits.

c. *Addition, subtraction, and negation*

Addition. The modular addition circuit we use [2] is an approximation of the operation $|u\rangle|v\rangle \mapsto |u\rangle|(u+v) \bmod p\rangle$. The inputs $u, v \in [0, p)$ are canonical integer representatives of finite field elements. Let $s = u + v \in [0, 2p-1) \subseteq [0, 2^{n+1})$ be their integer sum. The circuit applies the approximate reduction described above and returns

$$\mathrm{red}_\kappa(s) = r_s + q_s c - q_s w_s 2^\kappa = s - q_s p - q_s w_s 2^\kappa.$$

The circuit can fail to produce the correct result value in two different ways due to the applied approximations. The first is a failure due to the simplified comparison, where the most significant bit serves directly as the comparison bit. If $p \leq s < 2^n$, no correction is applied. The result is correct modulo p but remains larger than p.

The second failure occurs if the simplified comparison is correct, it causes a correction but the truncated carry propagation modifies the result because the carry propagates further than κ bits. This happens for $q_s w_s = 1$ and the result is 2^κ too small.

To count failure pairs $(u, v) \in [0, p)^2$, i.e., pairs that lead to the above two failures, fix an integer value $s \in [p, 2p-1)$ (no failure occurs for $s \in [0, p)$). For such s, the number of (u, v) with sum $s = u + v$ is $2p - 1 - s$. Summing over all relevant values for s that lead to a simplified comparison failure yields their overall number as

$$\sum_{s=p}^{2^n-1} (2p - 1 - s) = \frac{c(2p - c - 1)}{2}.$$

Failures due to truncated carry propagation occur when $q_s w_s = 1$ and $r_s \bmod 2^\kappa \in [2^\kappa - c, 2^\kappa)$. The relevant sums are of the form $s = 2^n + j2^\kappa - 1 - b$, $1 \leq j \leq 2^{n-\kappa} - 1$, $0 \leq b < c$. This time, the number of pairs $(u, v) \in [0, p)^2$ satisfying $u + v = s$ for fixed s is $2p - 1 - s = 2^n - 2c - j2^\kappa + b$, and again, summing over the relevant values yields their overall count as

$$\begin{aligned} &\sum_{j=1}^{2^{n-\kappa}-1} \sum_{b=0}^{c-1} (2^n - 2c - j2^\kappa + b) \\ &= (2^{n-\kappa} - 1)c(2^n - 2c) - c2^\kappa \frac{2^{n-\kappa}}{2}(2^{n-\kappa} - 1) \\ &\qquad + (2^{n-\kappa} - 1)\frac{c(c-1)}{2} \\ &= c(2^{n-\kappa} - 1)\frac{2^n - 3c - 1}{2} = c(2^{n-\kappa} - 1)\frac{p - 2c - 1}{2}. \end{aligned}$$

The two cases fall into disjoint events because for the first, the simplified comparison does not lead to a correction ($q_s = 0$), whereas for the second, it does ($q_s = 1$).

A third failure case is a (-1)-phase that is not corrected due to a failure in the approximate recomputation of the bit q_s after an X-measurement. This can happen even if the value of the representative is correctly computed. We are only interested in this case because it is disjoint from the previous two, where all value failures have been accounted for. This means that $w_s = 0$, i.e., $\mathrm{red}_\kappa(s) = s - q_s p = r_s + q_s c$ and the reduction yields the canonical representative.

The bit q_s can be precisely recovered using the fact that $q_s = 0$ if and only if $\mathrm{red}_\kappa(s) \geq u$. The comparison

used here is approximate, only taking into account the δ most significant bits of $\mathrm{red}_\kappa(s)$ and u. Therefore, a failed comparison computation can occur if $q_s = 1$ while $\mathrm{red}_\kappa(s) = s - p < u$, but the top δ bits of both are equal. To count the number of pairs $(u, v) \in [0, p)^2$ for which this failure arises, note that $s - p = u + v - p \in [c, p)$ and $\lfloor (s-p)/2^{n-\delta} \rfloor = \lfloor u/2^{n-\delta} \rfloor$. We obtain an upper bound by ignoring the condition $w_s = 0$. We count failure pairs (u, v) that satisfy $c \leq u + v - p < u < p$ by counting ordered pairs $(z = u+v-p, u)$ in $[c, p)$ that agree on their most significant $n-\delta$ bits, i.e., pairs that lie in an interval $I_j = [j2^{n-\delta}, (j+1)2^{n-\delta}) \cap [c, p)$. Let $|I_j| = c_j$. The I_j partition $[c, p)$, so $\sum_j c_j = p - c$ and each $c_j \leq 2^{n-\delta}$. The number of ordered pairs in I_j is $\binom{c_j}{2}$ and summing them yields

$$\sum_j \binom{c_j}{2} = \sum_j \frac{c_j(c_j-1)}{2} \leq (2^{n-\delta} - 1)\frac{p-c}{2}.$$

The total number of failure pairs covering all three cases is therefore bounded by

$$\begin{aligned} N_{\mathrm{add}} &\leq \frac{c(2p-c-1)}{2} + c(2^{n-\kappa}-1)\frac{p-2c-1}{2} \\ &\quad + (2^{n-\delta}-1)\frac{p-c}{2} \\ &= 2^{n-1}c\left(1 + \frac{p-2c-1}{2^\kappa}\right) + (2^{n-\delta}-1)\frac{p-c}{2}. \end{aligned}$$

For a uniform distribution of $(u, v) \in \mathbb{F}_p \times \mathbb{F}_p$, the failure probability due to simplified comparison, truncated carry propagation, or truncated phase correction is

$$\frac{N_{\mathrm{add}}}{p^2} < \frac{c}{p} + \frac{c}{2^{\kappa+1}} + \frac{1 + c/p}{2^{\delta+1}}.$$

Inputs during affine point addition. Inputs to the approximate addition in our elliptic curve point addition are not uniform random but intermediate results of our elliptic curve point addition.

Let (k, l) be a fixed pair of scalars and $0 \leq i \leq 2L - 1$. Assume no failures in the circuit up to the i-th point addition step. Let $A = (x_1, y_1)$ be the accumulator point and $T = (x_2, y_2)$ the i-th lookup point. Approximate addition is used three times in the point addition circuit: to compute $(x_1 - x_2) + R \bmod p$, $3x_2 + (x_1 - x_2 + R) \bmod p$, and $(R + x_2 - x_3) + (-R) \bmod p$, where R is a uniform random mask sampled fresh for each point addition step.

Note that we sample $a_0 \in \mathbb{Z}_r^*$ uniformly at random for each run of the double-scalar multiplication, and $\mu_i \in \mathbb{Z}_r \setminus \{-j2^{iw} : 0 \leq j < 2^w\}$ and $R \in \mathbb{F}_p$ are uniform random samples independent of a_0 and each other for the i-th windowed point addition step. The sample space of triples (a_0, μ_i, R) has size $(r-1)(r-2^w)p$.

Since we assume no failures up to this point, and with fixed previous random mask values, the map $(a_0, \mu_i) \mapsto (A, T)$ is injective. We can thus count the number of triples that lead to a given pair of inputs as follows.

For an input $(R, x_1 - x_2)$ to the first addition to be equal to a fixed pair $(u, v) \in \mathbb{F}_p \times \mathbb{F}_p$, it has to satisfy $R = u$ and $x_1 - x_2 = v$, which gives p possible triples. The two x-coordinates belong to at most two elliptic curve points each, meaning there are at most $4p$ pairs (A, T) corresponding to the given addition input pair. The same bound holds for the third addition. For the addition with inputs $3x_2$ and $x_1 - x_2 + R$, $3x_2 = u$ fixes x_2, yielding at most two curve points. There are at most $r - 1$ possible points A, each of which then fixes x_1 and thus R. This means that there are at most $2(r-1)$ points per addition input.

Adding these three counts and multiplying their sum by the number of addition failure pairs N_{add} gives a bound on the number of triples (a_0, μ_i, R) leading to one of the two approximation failures. Dividing by the total number of triples yields an upper bound on the addition failure probability p_{add} during a single elliptic curve point addition as

$$p_{\mathrm{add}} \leq \frac{(8p + 2(r-1))N_{\mathrm{add}}}{p(r-1)(r-2^w)} \leq \frac{10N_{\mathrm{add}}}{(r-1)(r-2^w)}$$

Note that we used $r - 1 < p$, which is not necessarily true for a general elliptic curve, but holds for `secp256k1`.

Lemma 2. *Let $n, w, p, c, r, \kappa, \delta$ be the parameters as described in this section. Let a_0, μ_i, and R be the random mask values as above. The probability p_{add} over random (a_0, μ_i, R) that either a simplified comparison failure or a carry propagation failure occurs during one of the three approximate additions in the i-th point addition step is bounded by*

$$p_{\mathrm{add}} < \frac{10 \cdot 2^{n-1}}{(r-1)(r-2^w)}\left(c(1 + \frac{p-2c}{2^\kappa}) + \frac{p-c}{2^\delta}\right).$$

For the concrete parameters of `secp256k1`, we have $n = 256$ and $c = 2^{32} + 977$. Our circuits use $\kappa = 65$, $w = 16$, and $\delta = 32$ such that

$$p_{\mathrm{add}} < 1.7463 \cdot 10^{-9} \approx 2^{-29.0931} < 2^{-29.093}.$$

Subtraction. Our modular subtraction circuit is an approximation of $|u\rangle\,|v\rangle \mapsto |u\rangle\,|(v-u) \bmod p\rangle$ for $u, v \in [0, p)$. It is computed using the approximate addition, braced by two modular complements that implement the operation $|x\rangle \mapsto |p - 1 - x\rangle$. More precisely, the circuit realizes

$$\begin{aligned} |u\rangle\,|v\rangle &\mapsto |u\rangle\,|p-1-v\rangle \\ &\mapsto |u\rangle\,|((p-1-v)+u) \bmod p\rangle \\ &\mapsto |u\rangle\,|p-1-((p-1-v+u) \bmod p)\rangle \\ &= |u\rangle\,|(v-u) \bmod p\rangle\,. \end{aligned}$$

Since we already obtained a bound on the failure probability due to approximate addition, it remains to bound the failure probability due to the approximate modular

complement. The operation $|x\rangle \mapsto |p-1-x\rangle$ is implemented by a bitwise complement followed by a truncated κ-bit addition of $2^\kappa - c$. The same result could be computed by a truncated addition of c to x followed by a bitwise complement. Without failures, this computes the modular complement because $p-1-x = 2^n - c - 1 - x = ((2^n-1)-x) - c = (2^n-1)-(x+c)$, again using the pseudo-Mersenne shape of p. The only additional source of failures is the truncated carry propagation leading to an incorrect value if $x \bmod 2^\kappa \geq 2^\kappa - c$. Counting failure inputs similarly to the analogous case for the modular addition, we obtain the number of inputs leading to a complement failure as

$$N_{\text{cmpl}} = c(2^{n-\kappa} - 1),$$

which means that for a uniform input, the failure probability due to truncated carry propagation in the modular complement is $N_{\text{cmpl}}/p < c/2^\kappa$.

Inputs during affine point addition. Like in the analysis of approximate addition, we fix scalars (k, l) and $0 \leq i \leq 2L-1$. Approximate subtraction occurs four times, namely to compute $x_1 - x_2$, $y_1 - y_2$, $(y_3 + y_2) - y_2$, and $(x_2 - x_3) - x_2$.

We argue very similarly to the addition case. Given x_1, x_2, there are at most 2 elliptic curve points with each x-coordinate, hence at most 4. For a pair of y-coordinates, there can be 3 points each due to the corresponding x-coordinate satisfying a cubic equation, at most 9 points in total. Again, due to bijections mapping the pairs of points before and after the point addition and pairs of values through the ideal subtraction, the same bounds apply to the last two calls as to the first two.

Hence, we can conclude that the probability p_{sub} of modular subtraction approximation failures during a single elliptic-curve point addition is

$$p_{\text{sub}} \leq \frac{(4+9+9+4)N_{\text{sub}}}{(r-1)(r-2^w)}.$$

Here, $N_{\text{sub}} \leq N_{\text{add}} + 2pN_{\text{cmpl}}$ is the number of input pairs that lead to a failure in the approximate subtraction circuit.

Lemma 3. *Let $n, w, p, c, r, \kappa, \delta$ be the parameters as described in this section. Let a_0 and μ_i be the random mask values as above. The probability p_{sub} over random (a_0, μ_i) that either a borrow propagation failure or a phase failure occurs during one of the four approximate subtractions in the i-th point addition step is bounded by*

$$p_{\text{sub}} \leq \frac{13}{5} p_{\text{add}} + \frac{52pc(2^{n-\kappa}-1)}{(r-1)(r-2^w)}.$$

Evaluating the expression with the concrete `secp256k1` parameters and implementation parameters $\kappa = 65$, $w = 16$, and $\delta = 32$ gives

$$p_{\text{sub}} < 1.0593796008 \times 10^{-8} \approx 2^{-26.4922} < 2^{-26.492}.$$

Negation. Finally, we analyze an approximation to $|u\rangle \mapsto |(-u) \bmod p\rangle$. Modular negation is computed by a bitwise complement, which gives $2^n - 1 - u$ and then we subtract $c-1$ to obtain $2^n - c - u = p - u$. The subtraction is approximate by truncating it to κ bits. This leads to an incorrect value if $u \bmod 2^\kappa \geq 2^\kappa - c + 1$. The integer value $p - u$ is the correct canonical representative of the modular negative of u except if $u = 0$. Therefore, the circuit swaps the values of 0 and p at the end, leaving all other values intact.

Again, we have two failure cases. The first is due to truncated subtraction of $c-1$ to κ bits, happening if $u \bmod 2^\kappa \geq 2^\kappa - c + 1$, and thus is induced by $(2^{n-\kappa} - 1)(c-1)$ of the canonical inputs.

The second failure is due to a shortened comparison window for the swap of 0 and p to the τ most-significant bits of the register. The swap is triggered when all these bits equal the least-significant bit. The circuit assumes that $\kappa \leq n - \tau$ and $\tau \geq 2$, otherwise it computes exact negation. Clearly, this comparison implementation has false positives, namely on non-zero even integers that are smaller than $2^{n-\tau}$ and on odd integers in $[2^n - 2^{n-\tau}, p)$. The total number of these failure cases is

$$\left(2^{n-\tau-1} - 1\right) + \frac{2^{n-\tau} - c - 1}{2} = 2^{n-\tau} - \frac{c+3}{2}.$$

However, we have already counted some of these as failure cases, namely those that have the form $u = j2^\kappa + 2^\kappa - c + h$, $0 \leq j < 2^{n-\kappa} - 1$, $1 \leq h < c$ which lead to an intended intermediate value $(2^{n-\kappa} - j - 1)2^\kappa - h$. Counting the blocks in these intervals, the overlap has size

$$(2^{n-\tau-\kappa+1} - 1)\frac{c-1}{2}.$$

Since a failure due to the truncated subtraction cannot be corrected by a short swap failure, the total number of failure inputs for approximate negation is

$$\begin{aligned} N_{\text{neg}} &= (2^{n-\kappa} - 1)(c-1) + 2^{n-\tau} - \frac{c+3}{2} \\ &\quad - (2^{n-\tau-\kappa+1} - 1)\frac{c-1}{2} \\ &= 2^{n-\tau} + (c-1)(2^{n-\kappa} - 2^{n-\tau-\kappa}) - c - 1, \end{aligned}$$

which means that the failure probability for uniform input satisfies

$$\frac{N_{\text{neg}}}{p} < \frac{c-1}{2^\kappa} + \frac{1 + c/p}{2^\tau}.$$

Inputs during affine point addition. Modular negation occurs once to compute $-x_3 \bmod p$. Fixing an input u therefore fixes x_3, so there are at most 2 elliptic curve points leading to this x-coordinate. The table lookup point still can assume any of its $r - 2^w$ possible values, leading to $2(r - 2^w)$ possible curve points leading to input x_3. With $(r-1)(r-2^w)$ possible point pairs, the probability p_{neg} of a failure due to approximate negation

during a single elliptic-curve point addition therefore is bounded by

$$p_{\text{neg}} \le \frac{2N_{\text{neg}}}{r-1}.$$

Lemma 4. *Let n, w, p, c, r, κ be the parameters as described in this section. Let a_0 and μ_i be the random mask values as above. The probability p_{neg} over random (a_0, μ_i) that either a truncated subtraction failure or a shortened comparison failure occurs during the approximate negation in the i-th point addition step is bounded by*

$$p_{\text{neg}} \le \frac{2(2^{n-\tau} + (c-1)(2^{n-\kappa} - 2^{n-\tau-\kappa}) - c - 1)}{r-1}.$$

Setting concrete parameters in our implementation for `secp256k1` with $\tau = 32$ results in

$$p_{\text{neg}} < 6.9850 \cdot 10^{-10} \approx 2^{-30.4150} < 2^{-30.415}.$$

d. Subtracting a square

Consider the subtraction of a square as described in Section VII. It acts on two registers and subtracts the square of the value in the first from that in the second modulo p. It maps $|x\rangle\,|y\rangle$ to $|x\rangle\,|(y - x^2) \bmod p\rangle$. We use a uniform random mask $R \in \mathbb{F}_p$ that is added to the target register, which also acts as the accumulator register in the algorithm. The square subtraction circuit then maps

$$|x\rangle\,|(y + R) \bmod p\rangle \mapsto |x\rangle\,|(y - x^2 + R) \bmod p\rangle$$

and a subtraction of R in the second register gives the desired result.

To understand the possible failure cases due to approximate arithmetic, we discuss the specific components as shown in Appendix A. The main operation is detailed for its integer-valued representation in Algorithm 7. It uses the components $\mathsf{AddBMultiple}^q_\pm$ (Algorithm 10), $\mathsf{AddCMultiple}^q_-$ (Algorithm 11), $\mathsf{AddSlice}^q_\sigma$ (Algorithm 8), and $\mathsf{PhaseGE}^q$ (Algorithm 9). Notation and parameter properties are introduced in Appendix A.

Approximation failures in AddSlice. The basic component used in the other algorithms is AddSlice (Algorithm 8). It operates on two quantum registers, a source register and a target register, holding values s and v. It updates the target by adding $\sigma 2^t s_{[o,o+w)}$. The parameter δ provides a way to trade efficiency for accuracy: it determines the number of bits used in the phase comparison, while the parameter $\rho \in \mathbb{Z}_{\ge 0}$ is the carry-padding width for local slice additions. The parameters must satisfy either $t+w+\rho < n$ or $t+w = n$ with $\delta \le w$. There are four possible sources of approximation failures in AddSlice.

A failure can occur during the complement computation of $(p-1) - v$ in the case of negative sign $\sigma = -1$. While the bitwise complement operation is exact, the preceding addition of c may drop a propagating carry. The corresponding failure preimage set is

$$\widetilde{E}_{\text{compl}} = \{v \in [0, 2^n) \mid v_{[0,\kappa)} \ge 2^\kappa - c\}$$

and has cardinality $c\,2^{n-\kappa}$. Complement computation only happens if $\sigma = -1$, but then it occurs twice, once when entering and once when leaving the complemented representation.

Another failure occurs for $t + w + \rho < n$ if in the addition of the source slice $s' = s_{[o,o+w)}$ into v, the carry propagates more than $w + \rho$ bits. The failure preimage set is

$$\widetilde{E}_{\text{carry}}(s, o, t, w) = \left\{v \in [0, 2^n) \mid v_{[t,t+w+\rho)} \ge 2^{w+\rho} - s'\right\}$$

with cardinality $s' 2^{n-w-\rho} \le (2^w - 1)2^{n-w-\rho} < 2^{n-\rho}$.

If $t + w = n$, a carry that propagates more than w bits means that the sum is at least 2^n. This carry is captured in an additional qubit, and an approximate reduction modulo p is applied by adding c with carry propagation truncated to κ bits. This failure case is captured by

$$\widetilde{E}_{\text{wrap}}(s, o, t, w) = \{v < 2^n \mid v + 2^t s' \ge 2^n, \\ ((v + 2^t s') \bmod 2^n)_{[0,\kappa)} \ge 2^\kappa - c\}.$$

Translation by $2^t s_{[o,o+w)}$ modulo 2^n is a bijection. Dropping the carry condition therefore upper bounds the cardinality of this set by the usual truncated carry-propagation bound $c2^{n-\kappa}$.

The fourth failure case in AddSlice is a failure due to an approximate phase correction in the case $t + w = n$. The extra carry bit can be uncomputed by an X-measurement and phase correction. Let $h = \mathbb{1}[v + 2^t s_{[o,o+w)} \ge 2^n]$ and

$$v^+ = ((v + 2^t s_{[o,o+w)}) \bmod 2^n) + hc.$$

If no failure occurs due to $\widetilde{E}_{\text{wrap}}(s, o, t, w)$, the addition of hc has no carry out of the low κ bits, so $v^+ < 2^n$. When $m = 1$ is measured, the exact carry correction contributes the phase $(-1)^q$. Together with a full-width call to PhaseGE, it gives

$$(-1)^q (-1)^{q\mathbb{1}[(v^+)_{[t,n)} \ge s_{[o,o+w)}]} = (-1)^{q\mathbb{1}[(v^+)_{[t,n)} < s_{[o,o+w)}]},$$

which is the required carry-dependent phase. The condition using $\ge$ implemented by PhaseGE is consistent with reconstructing the carry through the condition using $<$. The implementation supplies only the most-significant δ bits to PhaseGE, and the resulting reconstruction can fail for two reasons. First, when $h = 1$, the corresponding full-width comparison reconstructs h exactly if and only if $v < p$; hence a non-canonical input $v \ge p$ causes a failure. Second, the comparison on the most-significant δ bits can disagree with the full-width comparison. The failure preimage sets for those events are

$$\widetilde{E}_{\text{ph,rep}}(s, o, t, w) = \{v \in [0, 2^n) \mid v \ge p, \quad h = 1\},$$

with cardinality at most c and (with $s' = s_{[o,o+w)}$)

$$\widetilde{E}_{\mathrm{ph,trunc}}(s,o,t,w) = \{v < 2^n \mid \mathbb{1}[(v^+)_{[t,n)} \geq s'] \neq \mathbb{1}[(v^+)_{[n-\delta,n)} \geq s'_{[w-\delta,w)}]\},$$

and $|\widetilde{E}_{\mathrm{ph,trunc}}(s,o,t,w)| \leq 2^{n-\delta}$. The bound on the second cardinality follows because if the two comparisons disagree, the δ most significant bits of the values must be equal, fixing δ of the bits of v^+.

The operation AddBMultiple implements an addition or subtraction by a B-multiple of a register using a B-adic decomposition. For an integer s with $0 \leq s < 2^{n_s}$, where $n_s \in \{n, n+2\}$, write $s = s_L + Bs_H$ with $0 \leq s_L < B$. Then

$$Bs = Bs_L + B^2 s_H \equiv Bs_L + cs_H \pmod{p}.$$

The Bs_L term is applied by one wrapping call to AddSlice, and five local calls to compute cs_H using the fixed list T_c, which is a list containing a signed binary representation of c. For `secp256k1`, we have $T_c = ((32,+1),(10,+1),(6,-1),(4,+1),(0,+1))$, such that $c = \sum_{(d,\epsilon)\in T_c} \epsilon 2^d = 2^{32}+2^{10}-2^6+2^4+1$. There are no further target-dependent failure sites outside these six AddSlice calls.

The operation AddCMultiple adds or subtracts a multiple of c using the signed binary representation list T_c. For $0 \leq s < 2^n$, each element in T_c with $d > 0$ satisfies $2^d s = 2^d s_{[0,n-d)} + 2^n s_{[n-d,n)}$; if $d = 0$, there is no decomposition. The decomposition into low and high parts at the boundary n allows the addition of the high parts to be postponed and later corrected by combining them into $h_s = \sum_{(d,\epsilon)\in T_c} \epsilon s_{[n-d,n)}$. This contributes $2^n h_s \equiv ch_s \pmod{p}$. For `secp256k1`, where $n = 256$, this is

$$h_s = s_{[224,256)} + s_{[246,256)} - s_{[250,256)} + s_{[252,256)},$$

and $0 \leq h_s \leq c-3$. Hence $0 \leq ch_s < 2^{65}$, so the high correction fits in the fixed 65-bit auxiliary register. The implementation constructs ch_s by exact reversible signed additions modulo 2^{65}. Intermediate values may get reduced modulo 2^{65}, but the final residue is the ordinary integer ch_s. Reversing the same sequence therefore clears the auxiliary qubits. AddCMultiple makes five calls to AddSlice for the direct shifted terms and one call for the high correction ch_s. The exact construction and uncomputation of ch_s introduce no additional target-dependent failure sites, so all such sites occur in these six calls to AddSlice.

For `secp256k1`, the square-subtract trace contains three calls to AddBMultiple, each containing one call to AddSlice with three wrapping sites and five AddSlice calls with three occurrences of the local carry failure sites. There is one call to the negative version of AddCMultiple, which has five AddSlice calls that can wrap and one local call for the upper bits correction. The negative terms in the representation of c contribute ten complement failure sites and five phase corrections. Overall, there are 16 local AddSlice calls, 9 wrapping calls, and 13 negative calls, giving 26 complement sites.

Intermediate results in the algorithm can lie in $[0, 2^n)$ and thus be non-canonical representations of elements modulo p. However, under a failure-free execution on canonical inputs, the outputs lie again in $[0, p)$ and are thus canonical. This is clear if the control is $q = 0$ as the operation is the identity. For $q = 1$, the last operation of the algorithm that acts on the output register is a modular complement implemented by an addition of c followed by a bitwise complement in the last line of the AddSlice algorithm, which is called with negative sign $\sigma = -1$ from AddBMultiple. On a failure-free execution, the truncated addition of c into the v register right before the complement succeeds, which means that $v \bmod 2^\kappa < 2^\kappa - c$ because no carry has propagated out of the lowest κ bits. Since $v < 2^n$, we have $v+c < 2^n$, thus $v < p$ and the bitwise complement correctly computes $(2^n-1)-(v+c) = p-1-v \in [0,p)$. Therefore, any failure that would lead to a non-canonical output is already accounted for through previous failures.

Overall, under the helper conditions above, and assuming a uniformly masked target and exact integer squaring operations, residue projection and a union bound give bounds for the probabilities p_{val} of complement, wrapping and carry failures and p_{phase} of phase correction errors as

$$p_{\mathrm{val}} \leq \left(1+\frac{c}{p}\right)\left(\frac{16}{2^\rho}+\frac{35c}{2^\kappa}\right),$$
$$p_{\mathrm{phase}} \leq \left(1+\frac{c}{p}\right)\left(\frac{9}{2^\delta}+\frac{9c}{2^{256}}\right).$$

Consequently, the coherent failure probability p_{sq} for the square-subtract operation satisfies

$$p_{\mathrm{sq}} \leq \left(1+\frac{c}{p}\right)\left(\frac{16}{2^\rho}+\frac{35c}{2^\kappa}+\frac{9}{2^\delta}+\frac{9c}{2^{256}}\right).$$

For $\rho = \delta = 32$ and $\kappa = 65$, this gives $p_{\mathrm{sq}} < 2^{-26.59}$.

e. *Approximate in-place multiplication and division*

Another component needed in the point addition circuit is an in-place modular multiplication and its inverse, which functions as an in-place modular division, as described in Schrottenloher's recent work [2]. It is used to compute and uncompute the slope λ. While we have proved failure bounds for the modular squaring operation and the coordinate offset computations in the previous section, we obtain a bound on the failure probability for the gcd-based in-place multiplication by a Monte Carlo simulation instead. We sampled 1,000,000 uniformly random pairs $(u,v) \in \mathbb{F}_p^* \times \mathbb{F}_p$, simulated our approximate circuits to compute $|u\rangle|v\rangle \mapsto |u\rangle|(uv) \bmod p\rangle$ and $|u\rangle|v\rangle \mapsto |u\rangle|(u^{-1}v) \bmod p\rangle$, and counted the number of inputs for which the circuit failed (wrong output, residual phase, deallocation of a nonzero ancilla, etc.).

The simulation recorded $K = 21$ failures for both operations. The circuit parameters we used were mostly identical to the ones used by Schrottenloher [2]. We list the parameters in Table X.

TABLE X: The parameters used in the Monte Carlo simulations of the in-place multiplication circuit. Most of these parameters were described and chosen as in [2].

Parameter	Value
Iteration count	$1.413n + 2.4\sqrt{n}$
Register size at iteration i	$n - i/1.413 + 2.3\sqrt{n}$
Rounds with $x + y = p$	37
Comparison bits in gcd	$40 + 2.3\sqrt{n}$
Short modular corrections	+32 bits (65 total)
Phase-fix comparisons	32 bits

The simulation allows us to make the following statement using the Clopper-Pearson [85] bound: With confidence level at least $1 - 2^{-129}$, the probability that the in-place multiplication procedure using the binary gcd algorithm with the above parameters fails under uniform random inputs $(u, v) \in \mathbb{F}_p^* \times \mathbb{F}_p$ is bounded by 0.000149304. In turn, this allows us to conclude via union bound that with confidence level at least $1 - 2^{-128}$, both in-place multiplications succeed with probability at least $1-2\cdot0.000149304$ for uniform random inputs from $\mathbb{F}_p^* \times \mathbb{F}_p$.

Since the actual inputs to the in-place multiplication in the circuit are differences of elliptic curve point coordinates, their distribution is not equal to the uniform distribution used in the simulation. This introduces a factor of approximately 4 into the bound inequality, which we will show next.

The actual input to the first in-place modular division algorithm is the pair of differences $(x_1 - x_2, y_1 - y_2)$. Here $A_i = (x_1, y_1)$ is the current accumulator point after i point addition steps, i.e., $A_i = (x_1, y_1) = A_0 + S_i$, where S_i is the point accumulated into A_0 after i steps. The point (x_2, y_2) is the point looked up in the i-th window from the i-th lookup table T_i.

Because $a_0 \xleftarrow{\$} \mathbb{Z}_r^*$ is sampled uniformly at random and the map $A_0 \mapsto A_i$ as a shift by the point S_i is injective, and we only need to take into account first-time failures of trajectories that are correct up to this point, the point A_i can be one of $r - 1$ points, namely in the set $E(\mathbb{F}_p) \setminus \{\mathcal{O}\} + S_i$.

The point (x_2, y_2) is of the form $[j2^{iw} + \mu_i]P$ or $[j2^{(i-L)w} + \mu_i]Q$. Since μ_i is sampled uniformly at random from $\mathbb{Z}_r \setminus \{-2^{iw}j \mid j \in \{0, 1, \ldots, 2^w - 1\}\}$ or $\mathbb{Z}_r \setminus \{-2^{(i-L)w}j \mid j \in \{0, 1, \ldots, 2^w - 1\}\}$, the point can be any point in $E(\mathbb{F}_p)$ except for the set of points of size 2^w given by the exceptions above.

The set of possible input pairs $((x_1, y_1), (x_2, y_2))$ therefore has size $(r - 1)(r - 2^w)$, and we need to relate the distribution of input pairs $(x_1 - x_2, y_1 - y_2)$ to the uniform random distribution of pairs $(u, v) \in \mathbb{F}_p^* \times \mathbb{F}_p$ that was used in our Monte Carlo simulation.

The $u = 0$ case is already covered by our exceptional point analysis, so we may assume $u \neq 0$. Fix a specific pair (u, v) with $u \neq 0$. A given difference of curve point coordinates is equal to (u, v) if and only if $x_1 = x_2 + u$ and $y_1 = y_2 + v$.

Using that both points satisfy the curve equation, we obtain

$$\begin{aligned} y_2^2 + 2y_2v + v^2 = y_1^2 &= x_1^3 + b \\ &= x_2^3 + 3x_2^2u + 3x_2u^2 + u^3 + b, \end{aligned}$$

which, together with $y_2^2 = x_2^3 + b$, gives

$$2vy_2 + v^2 = 3x_2^2u + 3x_2u^2 + u^3.$$

If $v \neq 0$, y_2 can be expressed by a term quadratic in x_2, and substituting this expression into the curve equation shows that x_2 satisfies a polynomial equation of degree 4. Hence, there are at most 4 possible values for x_2, yielding at most 4 solutions for pairs (x_2, y_2).

If $v = 0$, then $y_1 = y_2$ and x_2 satisfies a quadratic polynomial equation, yielding at most 2 solutions for x_2 and again at most 4 pairs in total.

The same bound of at most 4 possible curve point pairs yielding a given pair of differences applies to the second call of the binary-gcd-based circuit, the in-place modular multiplication $(u, v) \to (u, uv \pmod p)$. After an execution of an ideal in-place multiplication, the outputs are $x_2 - x_3$ and $y_2 + y_3 = y_2 - (-y_3)$. These are again coordinate differences, where $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$ are the coordinates of the (negative of the) accumulator point A_{i+1} for the next iteration and the lookup point P_i, and so the factor-of-four bound holds at the point in the circuit. Now, the mapping (A_i, P_i) to $(P_i, -A_{i+1})$ is a bijection, as is the ideal, non-approximate in-place multiplication for non-zero u ($u = 0$ corresponds to an exceptional case already accounted for). As a result, there is a one-to-one correspondence between the multiplier inputs that fail and the corresponding (ideal) outputs, each of which corresponds to at most four curve point pairs.

Let q_{div} and q_{mul} be the failure probabilities of the in-place division and multiplication circuits for pairs (u, v) sampled uniformly at random. Hence there are at most $p(p - 1)q_{\text{div}}$ and $p(p - 1)q_{\text{mul}}$ failure pairs in $\mathbb{F}_p^* \times \mathbb{F}_p$. Each can be reached by at most 4 pairs of points under the difference map. The probability $p_{gcd,\text{div}}$ over uniform random samples of a_0 and μ_i that an input point pair leads to a failure in the division algorithm is at most

$$p_{gcd,\text{div}} \leq \frac{4p(p-1)}{(r-1)(r-2^w)} q_{\text{div}},$$

and analogously for the in-place multiplication

$$p_{gcd,\text{mul}} \leq \frac{4p(p-1)}{(r-1)(r-2^w)} q_{\text{mul}}.$$

Overall, the probability that a failure occurs in one of the in-place binary gcd circuits over the random uniform

choices of a_0 and μ_i is bounded by $\approx 4(q_{\text{div}} + q_{\text{mul}})$:

$$p_{gcd,\text{div}} + p_{gcd,\text{mul}} \leq \frac{4p(p-1)}{(r-1)(r-2^w)}(q_{\text{div}} + q_{\text{mul}}).$$

3. Failure probability bound

Using the above failure probabilities for all the components in the computation of $\widetilde{f}_\omega(k,l)$, we obtain the overall failure bound p_f as follows.

We start by bounding the failure probability p_{ecadd} for a single affine elliptic curve point addition using a union bound over all possible failure types for the separate components. We thus find that with probability $1 - 2^{-128}$,

$$\begin{aligned} p_{\text{ecadd}} &\leq p_{\text{sub}} + p_{gcd,\text{div}} + p_{\text{add}} + p_{\text{sq}} + p_{gcd,\text{mul}} + p_{\text{neg}} \\ &\leq 0.001195. \end{aligned}$$

Then, via union bound on the 28 point additions, we have that

$$p_f \leq 28 p_{\text{ecadd}} + \frac{6L}{r-1} \leq 0.03346.$$

4. Algorithm success probability with approximate arithmetic

In this section, we derive a bound on the success probability of Shor's algorithm for solving the ECDLP when using approximate double scalar multiplication with failure bound p_f. Our bound takes into account wrong outputs as well as phase errors due to approximate phase-fix predicates.

Recall the failure set

$$\begin{aligned} B_\omega = \{(k,l) \in &[0,2^n) \times [0,2^n) \mid \\ &f_\omega(k,l) \neq \widetilde{f}_\omega(k,l) \vee s_\omega(k,l) = -1\} \end{aligned}$$

for the masked approximate implementation $\tilde{f}_\omega(k,l)$, i.e., the set of inputs (k,l) for which the masked approximate implementation does not agree with the ideal masked value $f_\omega(k,l)$ or it agrees but the implementation produces a residual (-1)-phase.

For the analysis, we assume that we perform the known translation from $f_\omega(k,l)$ to $f(k,l)$ right after applying the oracle, i.e., we assume that the random point offsets of the masked implementation are removed. Because the projector onto successful outcomes of the algorithm (see the definition of M below) only acts on the input register, this shift does not affect outcome probabilities, and thus does not have to be implemented in practice. However, this assumption simplifies the analysis significantly. For brevity, we will write $x \in X := [0,2^n) \times [0,2^n)$ instead of (k,l) and $f(x)$ instead of $f(k,l)$.

Let $N = 2^{2n} = |X|$ and the ideal state before the QFT-measurement

$$|\Psi\rangle = \frac{1}{\sqrt{N}} \sum_{x \in X} |x\rangle |f(x)\rangle .$$

The corresponding approximate state is

$$|\tilde{\Psi}_\omega\rangle = \frac{1}{\sqrt{N}} \sum_{x \in X} s_\omega(x) |x\rangle |\tilde{f}_\omega(x)\rangle ,$$

where $s_\omega(x) \in \{-1, 1\}$ tracks the phase error for label x. We thus have $s_\omega(x) = 1$ and $\tilde{f}_\omega(x) = f(x)$ for all $x \notin B_\omega$.

Let the projector onto the successful outcome set S after the inverse QFT be $\sum_{y \in S} |y\rangle \langle y|$. Before the inverse QFT, this corresponds to the projector

$$M = \left(\text{QFT} \sum_{y \in S} |y\rangle \langle y| \, \text{QFT}^\dagger \right) \otimes \mathbb{1},$$

and we have that the success probabilities

$$P = \langle \Psi | M | \Psi \rangle \quad \text{and} \quad \tilde{P} = \mathbb{E}_\omega[\langle \tilde{\Psi}_\omega | M | \tilde{\Psi}_\omega \rangle].$$

We now write the difference between the ideal and the expected state $\mathbb{E}_\omega[|\tilde{\Psi}_\omega\rangle]$ as

$$|\Delta\rangle := \mathbb{E}_\omega[|\tilde{\Psi}_\omega\rangle] - |\Psi\rangle = \frac{1}{\sqrt{N}} \sum_{x \in X} |x\rangle |d_x\rangle ,$$

with $|d_x\rangle := \mathbb{E}_\omega[1_{x \in B_\omega}(s_\omega(x) |\tilde{f}_\omega(x)\rangle - |f(x)\rangle)]$.

Recall that p_f is such that for all $x \in X$,

$$\Pr_\omega[x \in B_\omega] \leq p_f.$$

Therefore, $\| |d_x\rangle \| \leq 2p_f$ from the triangle inequality for expectations and

$$\| \mathbb{E}_\omega[|\tilde{\Psi}_\omega\rangle] - |\Psi\rangle \|^2 = \frac{1}{N} \sum_{x \in X} \| |d_x\rangle \|^2 \leq \frac{4}{N} \sum_{x \in X} p_f^2 = 4p_f^2.$$

Using Jensen's inequality and the fact that M is a projector independent of ω, we have

$$\begin{aligned} \tilde{P} = \mathbb{E}_\omega[\| M |\tilde{\Psi}_\omega\rangle \|^2] &\geq \left\| M \mathbb{E}_\omega[|\tilde{\Psi}_\omega\rangle] \right\|^2 \\ &\geq (\max\{0, \| M |\Psi\rangle \| - \| M |\Delta\rangle \|\})^2 \\ &\geq \left(\max\{0, \sqrt{P} - 2p_f\} \right)^2 . \end{aligned}$$

Given a lower bound P_0 with $P_0 \leq P$, we thus have

$$\tilde{P} \geq \left(\max\{0, \sqrt{P_0} - 2p_f\} \right)^2 .$$

Appendix C: End-to-end integrated routing

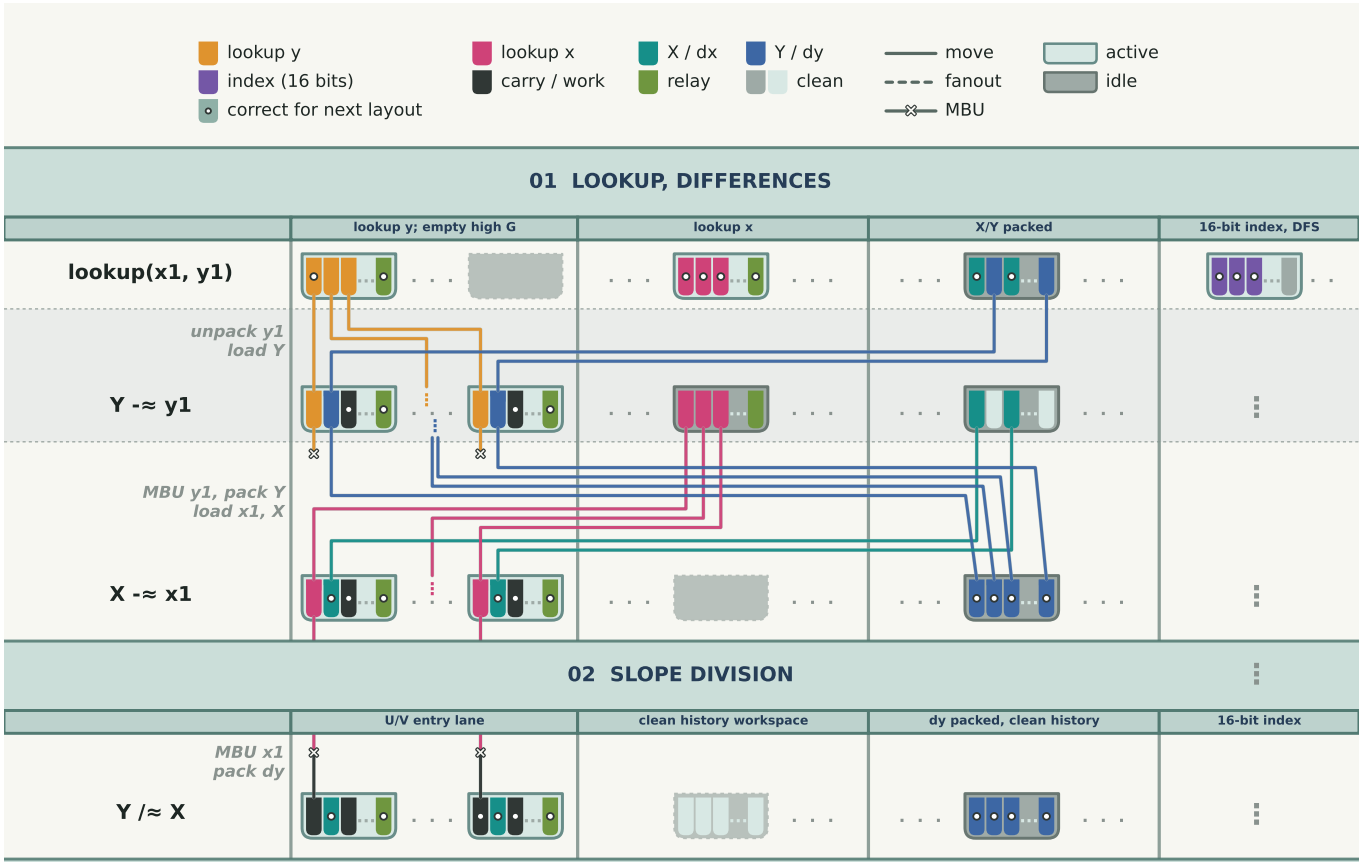


FIG. 21: Routing of the point-addition algorithm, stages 1–2.

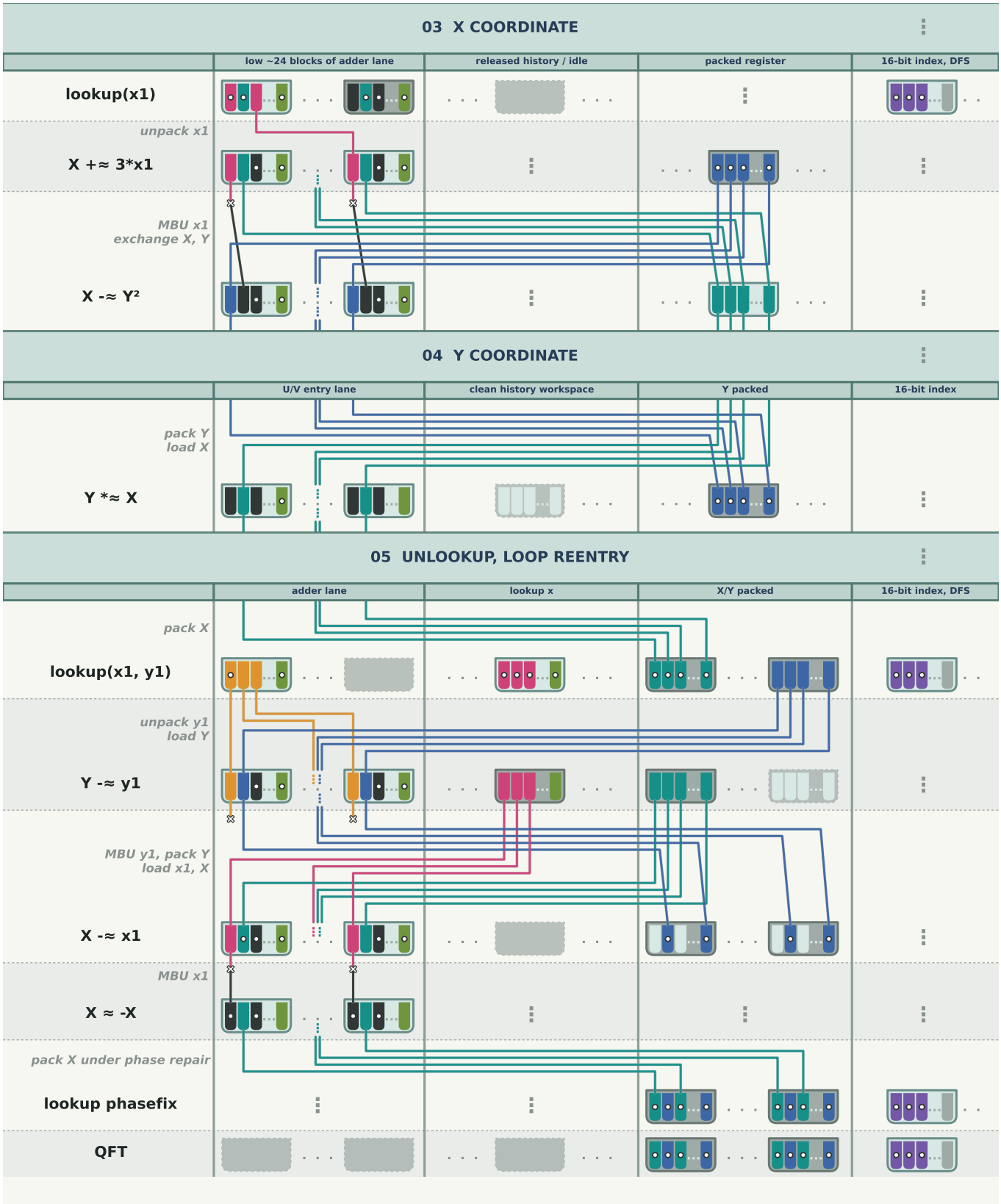


FIG. 21: Routing of the point-addition algorithm, continued: stages 3–5. The windowed point add ends in the same configuration as it begins, allowing for the routing to loop cleanly.

Appendix D: Circuit synthesis

Circuit synthesis turns the verified hierarchy of specs and defs into an architecture-legal, executable schedule. The entry point takes a root spec and its parameters, a target architecture, an optional layout plan, and synthesis options; it returns the measurement schedule together with its metrics and a serialized schedule artifact from which all downstream reporting is derived.

The architecture object declares a group of memory blocks and one or more groups of factory blocks. The memory type fixes block size and shape, which operations are directly legal within one block or between two blocks, and how many concurrent operations and logical measurements a block supports. Each factory group names a resource-state kind, a block count, the number of states per block, and the preparation and cleanup recipes for that factory type—for the ECC pipeline, CCZ factories. Legality checks, placement, and resource allocation all key off this object.

1. Synthesis tree and frontier lowering

Synthesis expands the root into a synthesis tree that records the decomposing hierarchy, the def selected at each node, qubit and classical-bit bindings, and classical conditions on components. The frontier lowering loop repeatedly selects a frontier node, chooses a def if the node is not directly legal on the architecture, allocates and binds the def's ancillae, and expands the node—until every leaf is architecture-legal. Routing is handled by placement and by local measurement-based gadgets rather than a separate global SWAP network: an operation that is not legal in the current geometry is decomposed into components such as logical measurements, controlled loads, CNOT fanout, or PcP gadgets (though we managed to entirely avoid naively decomposing PcPs in this work).

Schedule-dependent resource binding is deferred during this refinement: factory ancillae are virtual qubits in the tree, so defs and legality checks remain coherent while the physical factory assignment is chosen later, at scheduling time. Run-local facts—qubit ownership, ordering dependencies—are recorded in synthesis-side state and never leak into the reusable specs and defs. Two further mechanisms keep synthesis tractable at application scale: a compact representation of repeated structure, and memoized replay of previously lowered subtrees.

2. Compact repetition and deferred parameters

Repeated algorithmic structure is kept as a compact repeated subcircuit rather than expanded into a separate copy for every iteration. Its compile-time bounds determine how many times the body executes, while an iteration variable may select the operand qubit, register slice, constant, or implementation variant used on each pass. Repetitions may be nested or traversed in reverse, and inverting a repeated computation reverses both the iteration order and the enclosed operations. This representation is used for repeated shifts, multiplier digit sweeps, lookup traversal, modular-inversion rounds, and the repeated work units above point addition.

The repeated body remains compact while the compiler selects implementations, checks architectural legality, estimates costs, and records the synthesis state. When different iterations genuinely require different circuit shapes, the alternatives are selected by the iteration variable; changing register widths are a common example. Only when the final operation stream is emitted does the compiler substitute concrete operands and iteration-local classical results for every pass. This late expansion reduces compilation work and memory even though the executable schedule ultimately contains every operation.

3. Memoized lowering and subtree replay

The second mechanism is a dynamic-programming-style replay cache. After a subtree has been lowered completely, the compiler records its expansion under a structural key containing the source spec type and normalized parameters, the chosen def, and compatible ancilla bindings. These immutable tuples are hashable, so a repeated semantic component can locate an earlier lowering with a dictionary lookup rather than repeating recursive def selection, child-support tests, allotment, and layout decisions. Separate run-local caches remember preferred defs and architecture-support results.

On a replay hit, the compiler reproduces the cached child structure under the new occurrence and remaps occurrence-local classical identities. The replayed state includes the selected decomposition, ancilla bindings, sibling topology, factory-loan annotations, and qubit-ownership classes; it is not merely a copy of a flattened gate string. A candidate is rejected when its completed subtree or resource context is not safe to reuse, in which case ordinary expansion is performed. Rewinding or editing the live tree invalidates the affected cached state. These restrictions make hashing an optimization only: enabling or disabling replay must not change the emitted program.

This reuse happens during hierarchical synthesis, before measurement scheduling: the scheduler receives the bound serial stream and constructs concrete layers for every occurrence, so the replay speedup is a lowering speedup. Scheduler-side scalability comes instead from the packed layer representation and the optional late resolution of classical conditions described next.

4. Measurement scheduling

Beyond the general routine described in the main text, the scheduler applies a resource-pool model to state injections. Each CCZ factory use is split into an injection (three ZZ logical measurements, each coupling one CCZ state qubit to its memory target) and factory cleanup measurements. The scheduler tracks the number of available resource states per factory kind, decrementing on injection and incrementing on cleanup, and when a cleanup is delayed the propagating phase fix is tracked as a conditional Pauli byproduct. Only after scheduling are virtual factory indices assigned to physical factory blocks, pairing each injection with its matching cleanup. Internally, layers are stored as block-local Pauli bitmasks and flat instruction buffers rather than per-gate objects—the representation change that makes scheduling application-scale circuits tractable.

5. Layout plans

A layout plan is an authored, scoped placement policy for one spec implementation. It can force the def choice for its spec (and, through nested child plans, for the children) and binds semantic ports and ancillae to placement rules: serial strips through memory blocks, interleaved cycles, or direct block-and-slot coordinates. It can also reserve pools of helper qubits for Pauli-controlled-Pauli (PcP) gadgets. The plan is where design-level decisions enter: which registers are colocated, which resource pools exist, and which defs are forced. Below the planned shape the compiler still applies its local policies, but changing the plan changes the compiled tree, the schedule pressure, and the resulting metrics.

Appendix E: Architecture resource totals

The compiled-circuit resource counts used throughout this section are summarized in Table XI.

We note that the total number of Toffoli injections is different from the reported estimate in Part III, where we used the accounting from previous work.

This is for three reasons: (1) as discussed in Sec. X, we always execute conditional phase comparators in our compiled circuits as opposed to only counting operations if they are executed (as in previous work [2, 28]); (2) our phase-fix lookup is implemented suboptimally, i.e., we have not implemented the phaseup operation from [22]; and (3) the counts in Table XI include the initial lookup replacing the first point addition.

1. Footprint

Assuming an independent per-ion loss rate of $p_{\text{loss}} = 10^{-7}$ per POC, a code block containing its n data ions and n syndrome ancillas requires an average reloading rate

$$R_{\text{reload}} = (2n)p_{\text{loss}}. \tag{E1}$$

The resulting per-block rates for the three code types used in the architecture are given in Table XII.

TABLE XI: Compiled-circuit resource counts, assuming conditional operations are always executed.

Resource	Variable	Total
Measurement-layer depth	D_{meas}	75,251,329
Toffoli injections	N_{Tof}	40,420,330
Non-Toffoli logical measurements	$N_{\text{non-Tof}}$	369,882,166
Peak measurement parallelism	$k_{\text{cat}}^{(\text{peak})}$	24

TABLE XII: Per-block ion-reloading rates at $p_{\text{loss}} = 10^{-7}$ per ion per POC.

Code	n	Physical qubits ($2n$)	Expected losses per POC
Q102	102	204	2.04×10^{-5}
Q66	66	132	1.32×10^{-5}
C6	6	12	1.20×10^{-6}

Algorithm-wide exhaustion budget. For a reservoir of capacity r, we call the period between consecutive reservoir refills a *refill interval*, and let X be the number of replacement ions demanded during that interval. We define *exhaustion* as the event $X > r$, in which the replacement demand exceeds the available stock before the next refill. We assume that exhaustion of any local or global reservoir counts as a failed run and choose the reservoir capacities to bound the total run-wide exhaustion probability below the target

$$\varepsilon_{\text{res}} = 10^{-3}. \tag{E2}$$

Meeting this target ensures that reservoir exhaustion contributes at most 0.1 percentage points to the failure probability of an attempt.

For the runtime bound derived in appendix E 2, the device operates for at most $D_{\text{meas}} = 75{,}251{,}329$ measurement layers, as listed in Table XI. Thus,

$$\begin{aligned} T_{\text{run}}^{(\text{POC})} &= D_{\text{meas}} \times T_{\text{inj}} \\ &= 75{,}251{,}329 \times 147.48 \\ &= 1.10980 \times 10^{10} \text{ POCs}. \end{aligned} \tag{E3}$$

The device contains $69 + 4 = 73$ Q102 blocks, where the four additional blocks are the logical CliNR blocks. The four magic factories contain four Q66 blocks and eight C6 blocks, with two dedicated C6 blocks per Eastinthillation factory. Using these component allocations gives the following total numbers of refill intervals across

all reservoirs:

$$N_{\mathrm{Q102}} = 73T_{\mathrm{run}}^{(\mathrm{POC})}/27 = 3.00057 \times 10^{10}, \tag{E4}$$

$$N_{\mathrm{Q66}} = 4T_{\mathrm{run}}^{(\mathrm{POC})}/17 = 2.61129 \times 10^{9}, \tag{E5}$$

$$N_{\mathrm{C6}} = 8T_{\mathrm{run}}^{(\mathrm{POC})}/5.4 = 1.64415 \times 10^{10}, \tag{E6}$$

$$N_{\mathrm{global}} = T_{\mathrm{run}}^{(\mathrm{POC})}/5000 = 2.21960 \times 10^{6}. \tag{E7}$$

If q_i is the exhaustion probability in one refill interval for reservoir class i, the union bound gives

$$p_{\mathrm{res}} \leq \sum_i N_i q_i. \tag{E8}$$

We choose the smallest local and global reservoir capacities for which this bound remains below $\varepsilon_{\mathrm{res}}$. The corresponding sizing calculations follow.

To size each local reservoir, we assume that it can be refilled between SECs and model the number of losses in one SEC as a Poisson random variable with mean

$$\lambda = (2n)p_{\mathrm{loss}}T_{\mathrm{SEC}}. \tag{E9}$$

We choose the smallest number of ions r in each code block's local reservoir that is consistent with the run-wide exhaustion budget, using $q_i(r) = \Pr[\mathrm{Poisson}(\lambda) > r]$. The resulting values are shown in Table XIII. With one fewer ion, the Q102, Q66, and C6 contributions to the run-wide union bound would be 0.835, 0.00492, and 0.345, respectively—all above the 10^{-3} exhaustion budget.

TABLE XIII: Local ion-reservoir sizes, assuming only one refill per SEC.

Code	T_{SEC} (POCs)	λ	r	$q_i(r)$
Q102	27.0	5.508×10^{-4}	3	3.83×10^{-15}
Q66	17.0	2.244×10^{-4}	3	1.06×10^{-16}
C6	5.4	6.480×10^{-6}	2	4.53×10^{-17}

Unlike the loss model of [1], the swap-loss model produces no cascading losses, so the global reservoir only needs to buffer the independent steady-state loss stream from the device components, including the cat-state bundles and Bell-state resources. Based on the loading-rate comparison below, we assume that the reservoir is replenished once per second, or every $T_{\mathrm{refill}} = 5000$ POCs. For a candidate global capacity R_{global}, the mean number of losses in this interval is

$$\lambda_{\mathrm{global}} = \big(N_{\mathrm{components}} + R_{\mathrm{global}}\big)p_{\mathrm{loss}}T_{\mathrm{refill}}. \tag{E10}$$

The peak component allocations contain 19,363 qubits before the global reservoir is added. Applying our run-wide reservoir-exhaustion budget gives a 34-ion global reservoir, as summarized in Table XIV. Its exhaustion probability per refill interval is 2.77×10^{-10}. With a 33-ion buffer, the run-wide union bound would instead be 2.36×10^{-3}, so 34 is the smallest capacity consistent with eq. (E2). The runtime conversion in appendix E 2 gives $\tau_{\mathrm{POC}} = 200\,\mu$s. The mean reloading rate needed to maintain the global reservoir is therefore

$$R_{\mathrm{load}}^{(\mathrm{min})} = \frac{1.9397 \times 10^{-3}}{\tau_{\mathrm{POC}}} \approx 9.70 \text{ ions/s}. \tag{E11}$$

This rate is modest compared with demonstrated loading capabilities across the Walking Cat architecture's targeted qubit modalities. A trapped-ion experiment loaded about ten barium ions per ablation shot at its highest tested power. At the required mean rate, this yield corresponds to about 0.970 ablation shots per second [86]. In a neutral-atom system, continuous reloading has produced 30,000 initialized qubits per second [87].

TABLE XIV: Global reservoir sizing for independent steady-state ion loss. Here ρ_{loss} is the expected number of losses per POC and q_{global} is the exhaustion probability per refill interval.

Device	Total qubits	ρ_{loss}	R_{global}	q_{global}
`secp256k1`	19,397	1.9397×10^{-3}	34	2.77×10^{-10}

Substitution into eq. (E8) gives

$$p_{\mathrm{res}} \leq 7.32 \times 10^{-4}. \tag{E12}$$

Therefore, even if every exhaustion event aborts the computation, the probability of completing a run without reservoir exhaustion is at least 99.927%.

Memory and logical CliNR blocks. Both the memory blocks and the logical CliNR blocks use Q102. Each block contains $n = 102$ data ions, the same number of syndrome ancillas, and the three-ion local reservoir from Table XIII, giving the per-block footprint

$$n_{\mathrm{phys}}^{\mathrm{Q102}} = 2n + r_{\mathrm{Q102}} = 2(102) + 3 = 207. \tag{E13}$$

So, the 69 memory blocks contribute $69 \times 207 = 14{,}283$ physical qubits. The four logical CliNR blocks contribute a further $4 \times 207 = 828$ physical qubits, so the Q102 blocks occupy 15,111 physical qubits in total.

Cat-state register counts. We first determine the number of cat-state verification rounds used by both code types. For a target verification error $\varepsilon = 10^{-11}$ and physical error rate $p = 10^{-4}$, the Walking Cat architecture's verification-round formula [1] gives

$$m = \left\lceil \frac{\log \varepsilon}{2\log(2p)} \right\rceil = \lceil 1.49 \rceil = 2,$$
$$(2p)^2 = 4 \times 10^{-8} > \varepsilon, \qquad (2p)^4 = 1.6 \times 10^{-15} < \varepsilon. \tag{E14}$$

Thus, two verification rounds are sufficient and minimal under this bound. We use the Walking Cat transport timing $\eta = 1/20$ below.

For Q102, $n = 102$, $T_{\mathrm{SEC}} = 27$ POCs from Table IV, and $w_{\mathrm{cat}} = 30$. One long edge accommodates

$s(102, 30) = \lfloor 102/30 \rfloor = 3$ factories. The three parallel logical measurements therefore use three adjacent factories along the top edge, occupying 90 sites; no factories are required along the bottom or either short edge. Every factory has circulation distance $P_{\text{loop}}(102) = 212$ transport steps, and

$$\begin{aligned} T_{\text{prep}}(30, 2) &= 14.8 \text{ POCs}, \\ T_{\text{prep}}(30, 2) + \eta P_{\text{loop}}(102) &= 25.4 \text{ POCs}. \end{aligned} \tag{E15}$$

The total time is below the 27-POC SEC, so proposition 2 gives $r = 1$ for all three measurements.

For Q66, $n = 66$, $T_{\text{SEC}} = 17$ POCs, and $w_{\text{cat}} = 16$. One long edge accommodates $s(66, 16) = \lfloor 66/16 \rfloor = 4$ factories, so the three parallel logical measurements use three adjacent factories along one long edge, occupying 48 sites. Every factory has circulation distance $P_{\text{loop}}(66) = 140$ transport steps, and

$$\begin{aligned} T_{\text{prep}}(16, 2) &= 13.45 \text{ POCs}, \\ T_{\text{prep}}(16, 2) + \eta P_{\text{loop}}(66) &= 20.45 \text{ POCs}. \end{aligned} \tag{E16}$$

The total time exceeds one 17-POC SEC but fits within two, so proposition 2 gives $r = 2$ registers for each measurement, or six weight-16 cat-state registers in total.

Cat-state bundle pairs and Bell-state resources. Let k_{cat} denote the maximum number of logical measurements that the device must support in parallel. The reuse schedule of Sec. XII C 2 assigns two mobile cat-state bundles to each measurement stream. For Q102, each bundle contains one weight-30 cat-state register and one weight-30 verification bank; the weight 30 is set by the widest logical measurement made in a Q102 block. Together, the two bundles allocated to one measurement stream form a cat-state bundle pair. Each bundle occupies $2 \times 30 = 60$ physical qubits, so each bundle pair occupies 120 physical qubits. The resulting footprint is therefore

$$n^{\text{cat}}_{\text{phys}}(k_{\text{cat}}) = 2k_{\text{cat}}(2 \times 30) = 120k_{\text{cat}}. \tag{E17}$$

The peak compiled-circuit requirement is $k^{(\text{peak})}_{\text{cat}} = 24$ simultaneous logical measurements, as listed in Table XI. The cat-state allocation is therefore

$$n^{\text{cat}}_{\text{phys}} = 120k^{(\text{peak})}_{\text{cat}} = 120(24) = 2{,}880. \tag{E18}$$

In each cat-state production round, every LM2 measurement involving memory blocks consumes k_{Bell} Bell pairs to stitch the two cat states used in that round, where k_{Bell} is the number of pairs required to verify the stitched cat state with undetected-error probability at most ε. Following the Walking Cat stitching protocol [1], each Bell pair is prepared with a small unitary circuit. One qubit from each pair is routed to each of the two target cat-state factories. At the two factories, we measure a joint ZZ-stabilizer parity check between each Bell-pair qubit and one qubit of the local cat state. This procedure stitches the two cat states into a larger, joint cat state. Each Bell pair supplies one stabilizer measurement of the joint cat state, so the k_{Bell} pairs provide k_{Bell} redundant stitch checks. Writing $p' \le 4p$ for the error rate of one stitch check, including Bell-pair and transport faults, an incorrect parity can induce a high-weight X error on one half of the stitched cat state, but the stitch is rejected unless all k_{Bell} stitch checks return the same wrong value. At $p = 10^{-4}$, choosing $k_{\text{Bell}} = 4$ gives $(p')^{k_{\text{Bell}}} \le (4p)^4 = 2.56 \times 10^{-14}$, below the target precision $\varepsilon = 10^{-11}$. We provision one Bell-state bundle for each of the 12 simultaneous LM2 measurements at peak demand. Each bundle therefore contains $2k_{\text{Bell}} = 8$ physical qubits, including while those qubits are in flight. The four independent pairs can be prepared in parallel in two POCs, well within the 13.45–14.8-POC cat-state production interval; we assume that pre-positioning the bundles similarly hides their transport latency, so no additional Bell pairs are required for pipelining. Therefore, our bell-state allocation is:

$$n^{\text{Bell,LM2}}_{\text{phys}} = 12 \times k_{\text{Bell}} \times 2 = 12 \times 4 \times 2 = 96. \tag{E19}$$

Magic factories. The throughput analysis in Sec. XII E specifies a total of four Eastinthillation factories. Each factory contains one Q66 block, two dedicated C6 blocks, a local reservoir provisioned with the three Q66 replacement ions and two replacement ions for each C6 block from Table XIII, and three weight-16 cat-state factories. Including syndrome ancillas, each Q66 block and its two C6 blocks occupy 2(66) and $2 \times 2(6)$ physical qubits, respectively, while each cat-state factory contains two weight-16 registers and one bank of 16 verification ancillas, occupying 3×16 physical qubits. Finally, we count one additional six-qubit C6 Bell-state source for each C6 block, as required by the measurement construction in Sec. XII E. Thus, the footprint of one magic factory is

$$\begin{aligned} n^{\text{magic}}_{\text{phys}} &= 2(66) + 2 \times 2(6) + (3 + 2 \times 2) \\ &\quad + 3(3 \times 16) + 2(6) \\ &= 319, \end{aligned} \tag{E20}$$

and the four factories occupy

$$N^{\text{magic}}_{\text{phys}} = 4n^{\text{magic}}_{\text{phys}} = 4 \times 319 = 1{,}276 \tag{E21}$$

physical qubits in total.

Combining the peak allocation of each component type with the global reservoir gives the device footprint

$$\begin{aligned} N_{\text{phys}} &= 69(207) + 24(120) + 12(8) \\ &\quad + 4(207) + 4(319) + 34 = 19{,}397. \end{aligned} \tag{E22}$$

2. Runtime

The compiled algorithm has measurement-layer depth $D_{\text{meas}} = 75{,}251{,}329$, as listed in Table XI. To obtain an upper bound on the runtime, we assign every measurement layer the mean duration of a CCZ injection.

Among the logical measurements used in the compiled elliptic-curve point-addition circuits, a CCZ injection has the longest mean duration. It also determines the critical path whenever it appears in a parallel layer, because its outcome is needed to resolve subsequent Clifford corrections on the data blocks. The three independent Viterbi processes in a CCZ injection have a joint average stopping time of 5.46 Q102 SECs, as derived in eq. (26). Since one Q102 SEC takes 27 POCs, this gives $T_{\text{inj}} = 147.48$ POCs, or approximately 0.02950 seconds, when evaluated using the unrounded duration. The resulting expected-runtime upper bound per attempt is therefore

$$\begin{aligned} T_{\text{run}} &\lesssim D_{\text{meas}} \times 0.0294958 \text{ s} \\ &= 75{,}251{,}329 \times 0.0294958 \text{ s} \\ &= 2{,}219{,}600 \text{ s} \approx 25.7 \text{ days}. \end{aligned} \tag{E23}$$

3. Logical failure probability

We include five contributions to the algorithm-wide logical failure probability: memory errors in the Q102 blocks, errors in logical measurements, errors in the CCZ states consumed by Toffoli gates, errors in the T states consumed by the iQFT, and reservoir exhaustion.

Memory errors. As an upper bound, we assume that all $D_{\text{meas}} = 75{,}251{,}329$ measurement layers occur on all 69 Q102 memory blocks in the device. But, in actuality, only a small subset of qubits needs to hold quantum information continuously for this whole time in the application, so our upper bound is quite conservative. We find

$$\begin{aligned} N_{\text{SEC}} &= D_{\text{meas}} \times \tau_{\text{CCZ}}^{\text{avg}} \times 69 \\ &= 75{,}251{,}329 \times \tau_{\text{CCZ}}^{\text{avg}} \times 69 \\ &= 2.83615 \times 10^{10}. \end{aligned} \tag{E24}$$

At the Q102 logical error rate $p_{\text{L,SEC}}^{\text{Q102}} = 9.34 \times 10^{-12}$ per SEC from Table IV, this corresponds to a memory-failure probability of

$$p_{\text{mem}} = 1 - \left(1 - p_{\text{L,SEC}}^{\text{Q102}}\right)^{N_{\text{SEC}}} = 0.2327. \tag{E25}$$

The four Q102 logical CliNR blocks hold encoded states only for the duration of frame clearing, rather than for the full algorithm. Their exposure is therefore included in the non-Toffoli logical-measurement count below, not in the long-lived memory-failure total.

Although the calculated memory-failure probability is relatively high, our estimate is substantially inflated by the assumption that all 69 Q102 blocks continuously hold quantum information throughout the algorithm. The probability can also be readily reduced by increasing the code distance or tuning the decoder, as discussed for the memory-error bottleneck in the magic factory.

Logical measurement errors. For a weight-30 Viterbi measurement at physical error rate $p = 10^{-4}$, the single-round bit-flip model gives $p_{\text{flip},30} = 2.1 \times 30 \times 10^{-4} = 6.3 \times 10^{-3}$. A vote margin of five therefore has logical error probability

$$p_{\text{Vit},30} = \left[1 + \left(\frac{1 - p_{\text{flip},30}}{p_{\text{flip},30}}\right)^5\right]^{-1} = 1.02 \times 10^{-11}. \tag{E26}$$

For $N_{\text{non-Tof}} = 369{,}882{,}166$ such non-Toffoli logical measurements, as listed in Table XI, their combined failure probability is $1 - (1 - p_{\text{Vit},30})^{N_{\text{non-Tof}}} = 0.00378$.

Eastinthillation factory errors. The Eastinthillation factory has logical error rate $p_{\text{L}}^{\text{CCZ}} \lesssim 9.36 \times 10^{-10}$ per accepted state from eq. (60). Using one accepted state for each of $N_{\text{Tof}} = 40{,}420{,}330$ Toffoli injections from Table XI, the cumulative failure probability of the Eastinthillation factories is

$$p_{\text{CCZ}} = 1 - \left(1 - p_{\text{L}}^{\text{CCZ}}\right)^{N_{\text{Tof}}} \lesssim 0.0371. \tag{E27}$$

iQFT-stage errors. The 5,684 accepted T-state pairs and 464 synthesized rotations used for the iQFT contribute

$$p_{\text{iQFT}} \leq 7.64 \times 10^{-5}, \tag{E28}$$

as calculated in eq. (61).

Reservoir exhaustion. The device's reservoir-exhaustion probability is at most 7.32×10^{-4}.

Assuming these five failure mechanisms are statistically independent, their contributions combine to give

$$\begin{aligned} p_{\text{fail,alg}} &\leq 1 - (1 - 0.2327)(1 - 0.00378) \\ &\quad \times (1 - 0.0371)(1 - 7.64 \times 10^{-5}) \\ &\quad \times (1 - 7.32 \times 10^{-4}) \\ &= 0.2646 \approx 26\%. \end{aligned} \tag{E29}$$

Appendix F: Swap-loss sensitivity

We choose $p_{\text{swap}} = 0.5$ as a conservative order-one swap probability. The precise choice has little effect on the reported logical error rate: with the swap-resilient LDU, the simulations show no discernible dependence on p_{swap} over the tested range $[0, 0.75]$, within statistical uncertainty, as shown in Fig. 22.

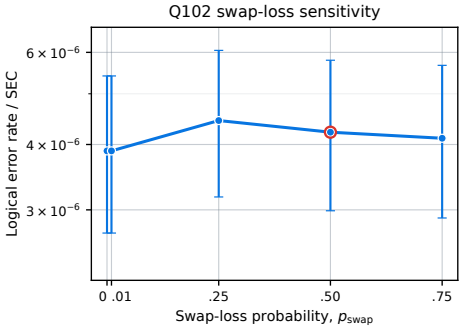


FIG. 22: Swap-loss sensitivity of the $[[102, 22, 9]]$ code. The logical error rate per SEC is shown as a function of p_{swap} at $p = 10^{-3}$, $p_{\mathrm{leak}} = 10^{-4}$, and $p_{\mathrm{loss}} = 10^{-6}$. Each point uses 10^6 shots of nine SECs, and the error bars are exact 95% binomial confidence intervals converted to per-SEC rates. The red ring marks the default simulation setting $p_{\mathrm{swap}} = 0.5$.

Appendix G: Q66 representatives for the Eastinthillation factory

This appendix gives physical Pauli representatives on the Q66 factory block for every measurement layer in Fig. 19.

TABLE XV: Q66 representatives for the RY0–RY3 layers.

Class	Target	Logical	w_{66}	Physical Pauli on qubits $0, \ldots, 65$
0	L3/Y0	YIII	16	IYIYYYIZIIIIIIIIYIIIIIIIXXIIIIIIIIIZYIIIIIYIIIIZZXZIIIIIIIZIIIIIIYI
	L2/Y2	IIYI	15	XIIIIYIIIIXXIIIIIIIIIZYZIIIIIXIIIIXIIZZYIIIIIIYIIIIIYIIIIIIIIIIIZII

TABLE XVI: Q66 representatives for the Acceptance layer.

Class	Target	Logical	w_{66}	Physical Pauli on qubits $0, \ldots, 65$
0	acceptance	ZIII	11	IIIIIIIIIIIIIIIIIIIZIIIIZIIIIIZIIIIIZIIIIIIIZZIIIIIIIIZIIZIZIIIZIZII
1	acceptance	XIXI	10	IIIIIIIIIIIIIIIIIIIXIIXXIIXXIIIXIIIIIIIIIIIIIIIIIIIIIXIIIIXXIIIXIIII
2	acceptance	YIYI	14	IIIIIIIIIIIXIIIIYZIIIIIIZIIYYIIZIIIIIIIIIIIIIIXIIYYIIXIIIIIIIXYIIIIZI
3	acceptance	IIZI	10	IIIIIIIIIIIIIIIIIIIIIZIIIIIIIIIZIIIIIIIZIIIIIIZZIIIIZZIZIIIIIIZZII
4	acceptance	ZZII	11	IIIIIIIIIIIIIIIIIIZIZIIIIZIIIZIIZIIZIIIIIIIZZIIIIIIIIIIIIIIIIZIZIIIZ
5	acceptance	XZXI	15	IXIIIIIIZYIIZIYIIIIIIIIIIIIIIIIIYIIIXYYIIYIIIIXZIIIIIIIIZIIIIIIIZXI
6	acceptance	YZYI	16	IIIIIIIIXIIIIXIIIIIIZIIIYYIIIIIIZIIIIZIIIXIIYYIIXIIIIZIYYIZIIIIIIZ
7	acceptance	IZZI	11	IIIIIIIIIIIIIIIIIIIIZIIIIZIIIIIZIIIIIZIIIIIIZZIIIIIIIIZIIZIZIIIZIZ
8	acceptance	ZIIZ	10	IIIIIIIIIIIIZIIIIIZIIIIIIZIIIIZIIIIZIIIIIIZIIIIIIIIIZZIIIIZIIIIZII
9	acceptance	XIXZ	15	IIIIXIIIIIIIZYYXIIIIIIIIYZIIIIIIIIYIIXXZIXIIIZIIIIIIIIIIYIIIIIIIIIIY
10	acceptance	YIYZ	16	IYIZIIIIZXIIIIYIIIIIIIIIZIIIIIIIYIIZXYXIIXZIIIYZIIIIIIIIIIIIIIIIIYI
11	acceptance	IIZZ	10	IIIIIIIIIIIIIIIIIIIIZIIIIIIIIIZIIIIIIIZIIIIIIZZIIIIZZIZIIIIIIZZIII
12	acceptance	ZZIZ	10	IIIIIIIIIIIIIZIIIIIZIIIIIIZIIIIZIIIIZIIIIIIZIIIIIIIIIZZIIIIZIIIIZI
13	acceptance	XZXZ	14	IIIIIIIIIIIIIIIIIIXIYIIIIIYIZZZIYIIIIIIIIIIIIIIIIIIIYIXXXIYIIIIIYIZ
14	acceptance	YZYZ	14	IIIIIIIIIIIIIIIIIIIIIIIIXZYIYYIYXIIIIIIIIIIIIIIIIIIIIIIIIIZYIYYIYXZI
15	acceptance	IZZZ	11	IIIIIIIIIIIIIIIIZIZIIIIZIIIZIIZIIZIIIIIIIZZIIIIIIIIIIIIIIIIZIZIIIZII

TABLE XVII: Q66 representatives for the CCZ injection layer.

Class	Target	Logical	w_{66}	Physical Pauli on qubits $0, \ldots, 65$
0	MZ0	IIIZ	15	IIZZIIZIIIIIIIIIIZIIIIIZZIZZIIZIIIZIIIIZZIIIIIIZIIIIIIIZIIIIIIIIZI
	MZ1	IZII	11	IZIIZIIZIIZIIZIIZIIZIIZIIZIIZIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
	MZ2	ZIII	12	ZIIIIIIIZIIZIIIIIIIIZIIIIIIIIIIIIIIIIIIIZIIZZIIZIIIIZIIZIIIZIIIIIIIZ
1	MZ0	IIIZ	10	IIIIZIZZIIIIIIIZIZZZIIIIIIIIIIIZIIIIIIIIIIIZIIIIIIIIIIZIIIIIIIIIIII
	MZ1	IZII	10	IIIIIIIIIIIIIIZIIIIIZIIIIIIIZIIIIZIIIIZIIIIIIZIIIIIIIIIIZZIIIIZIIIIZ
	MZ2	XIXI	10	IIIIIIIIIIXIIXIIIIIIIXIIIIXIIIIIIIIIIIIIIIIXIIIXXXIIXIIIIXIIIIIIII
2	MZ0	IIIZ	10	IIIIIIIIIIIIIIZIIIIIIZIIIIIZIIIZIIIIIIIZZIIIIIIIZIIIIIIIIIIZZIZIIIII
	MZ1	IZII	15	IIIIIZIIZIIIZIIIIZZIZIZIZZIIZIZIIIIIIZIIIIIIZIIIIIZIIIIIIIIIIIIIZI
	MZ2	YIYI	15	XIYIYIXXIIIZIZIYIIIIIIIIIIIIIIIIIIYIYXIIIIZIIIZYIZIIIIIIIIIIIIIIII
3	MZ0	IIIZ	10	IIIIIIZIZZZIIIIIIIIIZIIIIIIZIZZIZIIIIIIIIIIIZIIIIIIIIIIIIIIIIIIII
	MZ1	IZII	10	IIIIIZIIIIZIIIIIIZIIIIZIIIIIIIIIIIZIIIIIIIIIZZIIIIZIIIIZIIIIZIIII
	MZ2	IIZI	10	ZZIIIIIIIIIIIIIIZIIIIIIIIIIIIIIIZIIIIIIIIIZIIIIIZZZIIIIIIZIIIIIZII
4	MZ0	IIIZ	10	ZIIIZIIIIIIIIIIIIIIIZIIIIIIZIIIIIIZIIIIIIIIIIZZIIIIIIIZIIIIIIIIIIZZ
	MZ1	IZII	10	IIZIIIIIIZIIIIZIIIIIIIIIIIIIIIZIIIIIIZZIIIIZIIIIZIIIIZIIIIIIIZIIIIII
	MZ2	ZZII	11	IIIIIZIZIIZIIIIIIIIIIIIIIIZIIIIIIZIIIIIIIIZIZIZIIIIZZIIIIIIIZIIIIIIII
5	MZ0	IIIZ	10	IIIIIIIIIZIIIIIIZIZZIIIIIIIZIZZZIIIIIIIIIIIIIIIIIIIIIIIZIIIIIIIIIIZ
	MZ1	IZII	10	ZIIIIZIIIIIIIIIIIIIZIIIIIZIIIIIIZIIIIZIIIIZIIIIIIZIIIIIIIIIZZIIIII
	MZ2	XZXI	15	IIZYIIZIYIIIIIIIIIIIIIIIIYIIXIIIIIYIIIIXZIIIIIIIIZIIIIIIIZXIIIXYYI
6	MZ0	IIIZ	10	IIIIIIIIIZIIIIIIZIZZIIIIIIIZIZZZIIIIIIIIIIIIIIIIIIIIIIIZIIIIIIIIIIZ
	MZ1	IZII	10	ZIIIIZIIIIIIIIIIIIIZIIIIIZIIIIIIZIIIIZIIIIZIIIIIIZIIIIIIIIIZZIIIII
	MZ2	YZYI	16	IIIIIIXZYIYIYYZIIIIIIXIIIIIIIIIIIIIIIIZIIYIYXIIIIIIIIIXIIZIIIIIIZZII
7	MZ0	IIIZ	13	IIIZIIZIIIIIIIIIIIIIZZIZIIZIIIIIZZIIIIIZIIIZIIIIIIIZIZIZIIIIIIIIIII
	MZ1	IZII	12	IZIIIIIIZIZZZIIZIZIIIIIIIIIZIIIIIIZZIIIIIZIIIIIIIZIIIIIIIIIIIIIIII
	MZ2	IZZI	13	IIIIZIIIIIIIIIIIIIZZIIZIIZIIZIIZIIIIIIIIIIIZIZZIIIIIZIIIIIZIIIIIZI
8	MZ0	IIIZ	11	ZIIIIIIIIIIIIIIIIZIIIIIIIIZIIIIIIZZIIIIIIIZIIIIIZZZIIIIIIIIIIIIZZ
	MZ1	IZII	11	IZIIZIIZIIZIIZIIZIIZIIZIIZIIZIIZIIZIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
	MZ2	ZIIZ	11	IIZIIZIIZIIZIIZIIZIIZIIZIIZIIZIIZIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
9	MZ0	IIIZ	10	IIIIIIIIIIIIIIIIZIIIIIIIIIZIIIIIIIZIIIIIIIZZIIIIZZIZIIIIIIIZZIIIIIII
	MZ1	IZII	12	IIIIIIIIIIIIIIZZIIIIIIIZZZZIIZIZIIIIIIIZIIIIIIIIIIIIIIZIIZIIIIIIIZIIII
	MZ2	XIXZ	15	IYIXYIIIIIIIYIIIIIIIIIIIIIIIIIIIXIXYIIIIIIIIIIYZZIIIIIIZIIIIIIZZIIXIY

Continued on next page

Class	Target	Logical	w_{66}	Physical Pauli on qubits $0, \ldots, 65$
10	MZ0	IIIZ	10	IIIIIIIIIIIIIIIIZIIIIIIIIIZIIIIIIIZIIIIIIZZIIIIZZIZIIIIIIZZIIIIIII
	MZ1	IZII	10	IIIIIZIIIIIZIIIIIIZIIIIZIIIIIIIIIIIZIIIIIIIIIZZIIIIZIIIIZIIIIZIIII
	MZ2	YIYZ	16	YZIIYIIIIIIIZZYIIIIIIIYIZIIIIIZIZXIIXIXIIIIYXIIIIIIIIYIIIIIIIIIII
11	MZ0	IIIZ	15	IIIIIZIIIIIZZIIZIIZIIZIIIIIZIZIIZIIZIIIIIIIZIZIIIIIIIZIZIIIIIIIIZ
	MZ1	IZII	11	IZIIZIIZIIZIIZIIZIIZIIZIIZIIZIIZIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
	MZ2	IIZZ	12	IIIIIIZIZIIIIIIIIIIIIIIIZIIIIIZIIIIIIZZZIIIIIIIIZZIIIIIIIZIIIZZIII
12	MZ0	IIIZ	10	IIIIIIZIZZZIIIIIIIIIZIIIIIIZIZZIZIIIIIIIIIIZIIIIIIIIIIIIIIIIIIII
	MZ1	IZII	10	IIIIIZIIIIIZIIIIIIZIIIIZIIIIIIIIIIZIIIIIIIIIZZIIIIZIIIIZIIIIZIIII
	MZ2	ZZIZ	10	IZIIIIIZIIIIIZIIIIZIIIIIIIIIIIIIIIIIIIIZZIIIIZIIIIZIIIIZIIIIIZI
13	MZ0	IIIZ	10	IIIIIIZIIIIIZIIIZIIIIIIIIIIIIIIIZIIIIIIIIIIZZIZIIIIIIIIIZZIIIIIIIZ
	MZ1	IZII	10	IIZIIIIIIZIIIIZIIIIIIIIIIIIIIIZIIIIIIZZIIIIZIIIIZIIIIZIIIIIIZIIIIII
	MZ2	XZXZ	15	XIIIIIIIIIIIIIIIIYIIIIYIXYZYIIIZIIIIIIIIIIIIIIIIIZIXIIYXIIIXIIZIIYI
14	MZ0	IIIZ	10	IIIIIIIIIZIIIIIIZIZZIIIIIIIZIZZZIIIIIIIIIIIIIIIIIIIIIIZIIIIIIIIIIZ
	MZ1	IZII	10	ZIIIIZIIIIIIIIIIIIIZIIIIIZIIIIIIZIIIIZIIIIZIIIIIIZIIIIIIIIIZZIIIII
	MZ2	YZYZ	14	IIIIIIYIYIYYIXXZIIIIIIIIIIIIIIIIIIIIIIZIYIYYIXYIIZIIIIIIIIIIIIIIII
15	MZ0	IIIZ	10	IIIIIIZIZZZIIIIIIIIIZIIIIIIZIZZIZIIIIIIIIIIZIIIIIIIIIIIIIIIIIIII
	MZ1	IZII	10	IIIIIZIIIIIZIIIIZIIIIIIIIIIIIIIZIIIIIIZZIIIIZIIIIZIIIIIZIIIIZIII
	MZ2	IZZZ	11	IIIIIIIIIIIIIZIZIIIIZIIIZIIZIIIIIIIIIZZIIIIIIIIIIIIIIIIZIZIIIZIIZII

TABLE XVIII: Q66 representatives for the protected phase-fixup singleton layer.

Class	Target	Logical	w_{66}	Physical Pauli on qubits $0, \ldots, 65$
0	S	IIXZ	15	IIIIIIIXIIIIIYIIXXIIIIIYIIIIIIIXIIIIIIIZYIXIIYYIZIIIIIIZYIIIIIIIIZI
1	S	IXIZ	14	IIIIIIIIIZIXIIIIIXIIIIIIYYYIIIIIXIZIIIIIYYYIIIIIIZIIIIIZIXIIIIIIII
2	S	IZXI	15	YIIIIYIIIIXXIIIIIIIIIXIIIIZZYIIIIYIIIIYIIIIZIXIIIIZYIIIIIIZIIIIIII
3	S	IZXZ	15	IXZIIIIYIIYYZIIIZXIIIIIIIXIIIIIIIYIXIIYXIIIIIIZIIYIIIIIIIIIIIIIII
4	S	XIYY	15	IIIIIIIIXIIIIIYIIXXIIIIIYIIIIIIIXIIIIIIZYIXIIYYIZIIIIIIZYIIIIIIIIZ
5	S	XIZI	14	IIIIIIIIZIXIIIIIXIIIIIIYYYIIIIIXIZIIIIIYYYIIIIIIZIIIIIZIXIIIIIIIII
6	S	XIZZ	16	IYYIIIIXIIYIYZZIIIIIIIIIIYIIIIIIZXIIIZIIXIXXIIIIYXIIIIIIIIIIIIIIII
7	S	XYYI	16	XIYIYIIZIZIIIIIZIZZYIIIIIIIXYIIIIXIIYIYIIIIIIYIIIIIIYIIIIIIIIIIII
8	S	XYYZ	16	IIXZIIYIXIIIIIIIIIIIIIIIXYIZIIIIYZIYIIXZIIYIIIIXIYZIIIIIIIIIIIIII
9	S	XZYY	15	IIIYIZIZYYIIIIIIIIZXIIIIIIYIIIIXIZIIYIIIIIIYIIIIIYIZIIIIIIIIIIIYI
10	S	XZZI	16	IXXIZIIIIYIYXIZIZYYIIIIIIXIIZIIIIYIIZIIIIZIYIIIIIIIIIIIIIIIIIIII
11	S	XZZZ	16	IYIXXIIIIIZIIIIYIIIIIIIYXIIIIIZIZIIYIIIIIXIZIIZIYIIIIIIIIIIIIIIYZ
12	S	YIXY	15	IIIIYIIIIYIIIIIIIXIIIIIIYXIIIIZIXIIIIYIIIIYIIIIYXXIIIIZIIIIIIIIIZZ
13	S	YYXI	16	XIYIYIIZIIIIIIIIIIIIXIZIIIIIXYIIIIXIIYIYIZIIIIYZIIIIIXIZIIIIIIIIII
14	S	YYXZ	15	IIIIIYIIXIYIIIIIIIIIIIIIXIYIIIIIYIYXIXIYIIIIZIIIIYIIIIIIZXIIIIIIIIY
15	S	YZXY	14	IIIIIYIIIIYIIIIXIIIIXXIIIZIIIIIIYIZIIIIZIIIIYIIIIYIIIIIIYIIIIIXIIIZ
16	S	ZXZZ	16	ZIYIXXIIIIIZIIIIYIIIIIIIYXIIIIIZIZIIYIIIIIXIZIIZIYIIIIIIIIIIIIIIIY
17	S	ZZZX	16	IIYIIIXIIIIIZXIZIIYIIYIIIIIIXIIIIIIIIIIYIIYIIXIZXIIIIIZIIIYIIIIIIZ

TABLE XIX: Q66 representatives for the terminal phase-fixup DMG triple.

Class	Target	Logical	w_{66}	Physical Pauli on qubits $0, \ldots, 65$
0	MX0	IIIX	14	IXIIIIIIIIXIXXXIIXIIXXIIXIIIXIIIIXIIIIIIIIIIXIIXIIIIIIIIIIIIIIIIIX
	MX1	IXII	15	IIIIXXIIXIIIIIIIIIIIIIIXXIXXIIXXIXIIIIIXXIIIIIXIIIIIIIIIIIXIIIIXIII
	MX2	IIXI	11	IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIXIIXIIXIIXIIXIIXIIXIIXIIXIIXI
1	MX0	IIIX	12	IIIIIXXIIIXXIXIIIIIIIIIIXIIXIIIIXXIXIIIIIIIIIIIIIIIIXIIIIIXIIIIIIII
	MX1	YYXI	18	YZZIXIIIIIIIIIIIIXIZYYIIIYXIXIIZIIIIYIXIIIIIZXIIIIYIIIIIZIIIIIIIII
	MX2	IIXI	11	IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIXIIXIIXIIXIIXIIXIIXIIXIIXIIXI
2	MX0	IIIX	11	XXIIIIIIXIIIIIXIXIIIXIXIIXIIIIIIIIIIIIIIIIIIIIIXIIIIIXIIIIXIIIIIII
	MX1	IXII	12	IIIIIIXXIIIIIIIXIIXIIXIIIIIXIIIIIIIXIIXIIIIIIXXIIIIXIIIIIIIXIIIIII
	MX2	XIZI	14	IIIIXIIIIIIYYYIIIIIXIIIIIIIIIZIXIIIIZIIIIIZIXIIIIIIIIIZIIIIIYYYIII
3	MX0	IIIX	11	XIIXIIIIXIIIIIXXIIIIIIIIIIIIIIXIIIXIIIIIIIIXIXIIIIIIIXXIIIIIIIIIII
	MX1	XYYI	17	IIIIXIIIIIIIXIIIIYZIYIZZYIIIIIIIXIIIIXIZZIIIIIYIIYIYIIIIXIIIIIIIIZ
	MX2	XZZI	16	IIIIIIYIIXXZIZIIIIIIIIIIIIYYIIIXIIIZIIIYIIYIIIZIYZIIIIIIIIYIIIIIYII
4	MX0	YIXY	15	IIIYIIIIXXIIIIIIIIIXIIIIZZYIIIIYIIIIYIIIIZIXIIIIZYIIIIIIZIIIIIIIYI
	MX1	IXII	10	IIIIIXIIIIIIXXXIIIIIXIIIIIIIIIIIIXIIIIIIIIIIIIXIIIIIIIIIIIIIIIXXXII
	MX2	IIXI	10	IXIIIIXIIIIXIIIIXXIIIIIIIIIXIIIIIIIIIIIXIIIIXIIIIIIXIIIIIXIIIIIIII
5	MX0	YZXY	14	IZIIIIIYIIIIIIYIIIIYIIIIXIIIIXXIIIIIIXIIIZZIIIIZIIIIYIIIIYIIIIIIYI
	MX1	YYXZ	15	IIXIYIIIIIYIIIIIIYIIXIYIIIIIIIIIIIIZXIIIIIIIIYYXIXIYIIIIZIIIIYIIIII
	MX2	IIXI	10	IIIIIIIIIXIIIIIIXIIIIXIIIIXIIIIXXXIIIIIXIIIIIIIIIIIIIIIXIIIIXIIIIII
6	MX0	XIYY	15	IIXIIIIIYIIXXIIIIIYIIIIIIIXIIIIIIZYIXIIYYIZIIIIIIZYIIIIIIIIZIIIIIII
	MX1	IXII	10	XIIIIIIXIXXIIIIXXIIIIIIXIIIIIIIIIXIIIIIIIIXIIIIIIIIIIIIIIIIIIIIIXI
	MX2	XIZZ	16	IIIIXIIIIIIIIYXIIIIXIYIIIIIIIIZXIIIIIIZIIIIIZYZIIIIIYZIIIIIIXYYIII
7	MX0	XZYY	15	IIIYIZIZYYIIIIIIIIZXIIIIIIYIIIIXIZIIYIIIIIIYIIIIIYIZIIIIIIIIIIIIYI
	MX1	XYYZ	16	IIIIIIIIIIIIXYIZIIIIYIIXZIIYIXIIIIIXIYZIIIIIIIIIIIIIIIIZIYIIXZIIYII
	MX2	XZZZ	18	IIIIIIYIIIIZIIZIIYIIIZIIIIIIIIIIIZIIIIIIZYXYIYZIZIIZIYYIYIIIIIIXIII

TABLE XX: Q66 representatives for the terminal phase-fixup DMG pair.

Class	Target	Logical	w_{66}	Physical Pauli on qubits $0, \ldots, 65$
0	MX0	IIIX	11	XXIIIIIIXIIIIIXIXIIIXIXIIXIIIIIIIIIIIIIIIIIIIIIIIXIIIIIXIIIIXIIIIIII
	MX2	XIZI	14	IIIIXIIIIIIYYYIIIIIXIIIIIIIIIZIXIIIIZIIIIIZIXIIIIIIIIIZIIIIIYYYIII
1	MX0	IIIX	11	XIIXIIIIXIIIIIXXIIIIIIIIIIIIIIXIIIXIIIIIIIIXIXIIIIIIIXXIIIIIIIIIIII
	MX1	XYYI	17	IIIIXIIIIIIIXIIIIYZIYIZZYIIIIIIIXIIIIXIZZIIIIIYIIYIYIIIIXIIIIIIIIZ
2	P0	IIIX	14	IXIIIIIIIIXIXXXIIXIIXXIIXIIIXIIIIXIIIIIIIIIIXIIXIIIIIIIIIIIIIIIIIX
	P1	ZXZI	17	IIIZIYIIIZIYIIIZXIIIIIIIIIIXIIIIIIYXYIIIZIIZIZIIZXIIIIIIIXXIIIIIII
3	P0	IIXI	12	IIIXXIIXIIIXIIIIIXXIIIIIIXIIIIIXIXIIIIXIXIIIIIIIIIIIIIIIIIIIIIIXI
	P1	IZIX	20	IXXIIXYIIIIIIYIIIIIIYIIZZIIZIIZIIIXIXIIIIIIXIIZZIIIIIIIIIIIIZYYYIX
4	MX1	YYXI	18	YZZIXIIIIIIIIIIIIIXIZYYIIIYXIXIIZIIIIYIXIIIIIZXIIIIYIIIIIZIIIIIIIII
	MX2	IIXI	11	IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIXIIXIIXIIXIIXIIXIIXIIXIIXIXI
5	MX0	YZXY	14	IZIIIIIYIIIIIIYIIIIYIIIIXIIIIXXIIIIIIXIIIZZIIIIZIIIIYIIIIYIIIIIIYI
	MX2	IIXI	10	IIIIIIIIIXIIIIIIXIIIIXIIIIXIIIIXXXIIIIIXIIIIIIIIIIIIIIIXIIIIXIIIIII
6	P0	IIXZ	19	IIZIIIIIIIIIIIIIXIZIIIIIIIZIIIIXIZZXXXIIXIIYXIIIIYYYIIIIIYYIIIIIXI
	P1	IXIZ	16	ZIIIIXIIZIIIIIXXIIIIIYYYIIIIIIXIYIIIIIIIIZIIIIYIIIIIZIXIIIIIIIXYII
7	P0	IIXZ	18	ZIIIIIIIIZIIIIIZIIIIIIIIIIIIIIIIIIZXIIXIIXIIXIIYZIYZIXIIXIIYIIXIIYZ
	P1	YYXI	18	IIIXIZIYIIIZZIIIIZIIYIIXYIIIXXIYIIIIIIIYIXZIIIIIXIIIIYIIIZIIIIIIII
8	P0	IIXZ	17	IIIIIIIXIIXIYXXIIIXYIYZIIIYIIXYIYYYYIIIIIIIIIIIIIIIIIIIIIIIIIIIIYI
	P1	YYXZ	18	IIXIZYIIIYIYIIIIIIIIIIIIIIIIXYIIIIIIIIIXZIXIZZXIIIIXZXIIIIIZIIIIIIZ
9	MX0	XIYY	15	IIXIIIIIYIIXXIIIIIYIIIIIIIXIIIIIIZYIXIIYYIZIIIIIIZYIIIIIIIIZIIIIIII
	MX1	IXII	10	XIIIIIIXIXXIIIIXXIIIIIIXIIIIIIIIXIIIIIIIIXIIIIIIIIIIIIIIIIIIIIIIXI
10	MX1	IXII	12	IIIIIIXXIIIIIIIXIIXIIXIIIIIXIIIIIIIXIIXIIIIIIXXIIIIXIIIIIIIXIIIIIII
	MX2	XIZI	14	IIIIXIIIIIIYYYIIIIIXIIIIIIIIIZIXIIIIZIIIIIZIXIIIIIIIIIZIIIIIYYYIII
11	P0	IXII	11	XXXIIIIIIIIIXXIIIIXXIIIIIIXIIIIIIXIIIIIIIXIIXIIIIIIIIIIIIIIIIIIIII
	P1	ZIZX	18	IIIXIIIYIZIZIIYIIIIIIIXIIIIXIZYIIIIYIIIIYIIYIXIYIIIIIXXIIIIIYZIIII
12	B1	IXIZ	14	IIIIIIZIXIIIIIXIIIIIIYYYIIIIIXIIIIIIIYYYIIIIIIZIIIIIZIXIIIIIIIIIZI
	B2	XIZI	14	IIZIXIIIIIXIIIIIIYYYIIIIIXIIIIIIIYYYIIIIIIZIIIIIZIXIIIIIIIIIZIIIII
13	P0	IXIZ	18	XIIXIXYIZZZIXIIIIIIIXZIIIIIIYXYZXZIIIIIIIIIIYIIIIIIIIIIIIIIIIIIXI
	P1	XIZZ	16	IIXIIIIZIIIIIIIIIIIIIIZZYIIIIIIIIIIIIIXYZYIIIIIIYIIIIIZIIIIXXXIXIIZ
14	P0	IZIX	18	IIIIIIIIIIIXIIIIIIZIYXIIIIIIIXIIIZIIIXXYIYIIIXYZZIIYIYIIIIZIIIIIIZ
	P1	IZXI	18	IXIXIIIIIIIIZZIIIIIYIIIZXIIIXIIXIIIYYIIIIIYYIIIIIZIIZIIIZXIIIYIIII
15	P0	IZIX	16	XIIYIIIIXZIIIIYXIIIIIIIIIIIIZYIIIYIZIZIIIIYIXIIZIIIIYXIIIIIIIIIIII
	P1	XIZI	14	IIIIIYYYIIIIIXIIIIIIIIIZIXIIIIIXIIIIZIXIIIIIIIIIZIIIIIYYYIIIIIIZII
16	P0	IZIX	20	XIIZZZZXIYXZIIIXIIIXIIXIIIIIZYIIIYIIYIIXIIIYIIIIIIXIYIIIIIIIIIIIII
	P1	XZZI	17	IIIIIIIIIIIIIIIIIIIIIIYIIIIZIIIIIYIXXIXZIIIIIXZZIIIIIIIZXXYIYIIIZIZ
17	P0	IZXI	22	XZXXZIIZIIYIIZIIZIXYXXYXXYIXZXIZIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIXI

Continued on next page

Class	Target	Logical	w_{66}	Physical Pauli on qubits $0, \ldots, 65$
	P1	YIXY	22	IIIIIZIIIIIIIIIIIIIIIIIIIIIIIIIIZZXZIXIZXZIYZZYZZYZIXIIXIIYIIXIIX
18	P0	IZXI	17	XIIYZZYXIZIZIIXIIIIIXXIIIIIIYZIIIYIIYIIIIIIYIIIIIIIIZIIIIIIIIIIII
	P1	YZXY	19	IYIIIIIIZIIIIIIZIIIIIIXIIIIXIIYIIIYIIIXXIXIIIIIIYZIIIZIXZZIIYIIYIX
19	MX0	XIYY	15	IIXIIIIIYIIXXIIIIIYIIIIIIIXIIIIIIZYIXIIYYIZIIIIIIZYIIIIIIIIZIIIIII
	MX2	XIZZ	16	IIIIXIIIIIIIIYXIIIIXIYIIIIIIIIZXIIIIIIZIIIIIZYZIIIIIYZIIIIIXYYIII
20	P0	XIYY	15	IIIYIIIIIIIXIIIIIIIXIIIIIYIIXXIIIZYIIIIIIIIZIIIIIIZYIXIIYYIZIIIII
	P1	ZXZI	15	IIIIIIXYYIIIIIIZIIYYIIZIXXIIIIIIXIIIIIZYIIIIIIIXIIYIIIIZIIIIIIIII
21	P0	XIZX	15	XIIIIIYIIXXIIIIIYIIIIIIIXIIIIIIIIIXIIYYIZIIIIIIZYIIIIIIIIZIIIIIIZY
	P1	ZXZZ	16	IIIXIXIIIIIYIIIIIIZXIYYIIZIYYIIIIIIYYIIIIIIIZIIIIIIIZYIIIIIIZIIIII
22	P0	XIZZ	16	IZIIIIIIXIIIZIIIIIIIZIIIIIIIZIXIIYIYZIIIIIIYYIXIIIIZIYIIIIIIIIIIXY
	P1	XZYY	15	IIIIXIIIIYIZIZYYIIIIIIIIZXIIIIIIYIIIIYIZIIYIIIIIIYIIIIIYIZIIIIIIII
23	MX1	XYYI	17	IIIIXIIIIIIIXIIIIYZIYIZZYIIIIIIIXIIIIXIZZIIIIIYIIYIYIIIIXIIIIIIIIZ
	MX2	XZZI	16	IIIIIIYIIXXZIZIIIIIIIIIIIYYIIIXIIIZIIIYIIYIIIZIYZIIIIIIIYIIIIIYII
24	P0	XYYI	16	XIYIYIIZIZIIIIIZIZZYIIIIIIIXYIIIIXIIYIYIIIIIIYIIIIIIYIIIIIIIIIIII
	P1	XZZX	16	IYIIIIXIXIYIIIIIIIIIIIIZYIIIIXIIZIXZIIIIYIIZIIIYXIIIIIIIIIIIIZYIII
25	P0	XYYI	16	XIYIYIIZIZIIIIIZIZZYIIIIIIIXYIIIIXIIYIYIIIIIIYIIIIIIYIIIIIIIIIIII
	P1	XZZZ	16	IZIIIIZIIIIIIIZIIIIIIYIZIZIIIIIIIXIYXIYIZIIIIXIIIIIIIIIIYYXIYIIIIII
26	P0	XYYI	16	ZYIIIIIIIXYIIIIXIYIYIIZIZIIIIIZIZIYIIIIIIIIIIIIIIXIIYIYIIIIIIYIIIII
	P1	ZIZX	17	IIIIIIIIIIIIYIIIIIZIXXIIIIIYIIIIIZIYIIYYYZYIIYIIIYIIIIIIIZIYIIIIZI
27	MX0	XZYY	15	IIIYIZIZYYIIIIIIIIZXIIIIIIYIIIIXIZIIYIIIIIIYIIIIIYIZIIIIIIIIIIIIYI
	MX1	XYYZ	16	IIIIIIIIIIIIXYIZIIIIYIIXZIIYIXIIIIIXIYZIIIIIIIIIIIIIIIZIYIIXZIIYII
28	P0	XYYZ	16	YIYIXZIIIIIIIIIIIIIYIIIIIZXXIIIIXZIYIYIIIIZZXIIIIIIYIIIIIIIIIIII
	P1	XZZI	16	IIIIIIIXXIIIIIIXIYYIZIZYYIIIIIIXIIIIIIIYZIIIIIZIIYIZIIIIIIIIIIIIIZ
29	MX1	XYYZ	16	IIIIIIIIIIIIXYIZIIIIYIIXZIIYIXIIIIIXIYZIIIIIIIIIIIIIIIZIYIIXZIIYII
	MX2	XZZZ	18	IIIIIIYIIIIZIIZIIYIIIZIIIIIIIIIIIZIIIIIIZYXYIYZIZIIZIYYIYIIIIIIXIII
30	P0	XYYZ	16	IIIYIYIIZIIIIIIIIIIIIXYIIIIIIYIIXIIXIZIYIZIIXIYZIIIIIIIZIIIIIIIIIX
	P1	ZIYY	16	YIXIZIIZIIIIIIIIIXIIIIIIIXYIZIIXIIYIYIIIZIIYIIIIIIYZIIIIIIIIIIIIXI
31	MX0	XZYY	15	IIIYIZIZYYIIIIIIIIZXIIIIIIYIIIIXIZIIYIIIIIIYIIIIIYIZIIIIIIIIIIIIYI
	MX2	XZZZ	18	IIIIIIYIIIIZIIZIIYIIIZIIIIIIIIIIIZIIIIIIZYXYIYZIZIIZIYYIYIIIIIIXIII
32	P0	XZYY	18	IIIZIIYIIYIIIIIIIIXIIIIIIIIXIIZIIZIYIIYIXIYXXYIIIIIXYZIIIIIYIIIIII
	P1	YYXZ	19	IIXIIIIIYIXIIXIZIZIIIIYIIIYIXZIYIIIIIIIIIXIIIIXIIZIIIIXIIYYIIIXZII
33	P0	YIIY	20	IIYIXYIIZIIZXIZIIZIIYXXYXXYXIYIXZXIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
	P1	ZXZZ	18	IZIIIIIYIIZIIZIIIIIXIIIIIIIIIIIIIIIIIIIIZIZIXYXIXZIIZIIIXXIYZIIIIIYII
34	P0	YIXY	15	IIIYIIIIIIIXIIIIIIYXIIIIZIXIIIIYIIIIYIIIIYXXIIIIZIIIIIIIIIIZZIIIIYI
	P1	ZXZI	15	YYIIZIXXIIIIIIXIIIIIIXYYIIIIIIZIIIIIIZIIIIIIIIIIIIIIIIIZYIIIIIIIXIIY
35	P0	YYXI	17	IYIIIIIIIZIIIIYIIIIIIIIIZIIIIIZIIXXYIYIIYZIZYIIXIIIIIIYIIIIIIIIIYX
	P1	ZIYY	16	IIIIXYIZIIIIYIIXZIIYIXIIIIIIIIIIIIIIIIIIIIIIIIIIZIYIIXZIIYIIIIXIYZII
36	P0	YYXI	16	IIYIZXIIIZIIIIIYIIIIIIIIXYIIIIIIYIIYIZIXIIYIIIYYIXIIIIIIIIIIIIIIIZ
	P1	ZIZX	16	IIIZIIYYIIZIIIXIIIIXIIIIIIXIYYIXIIIIIIZIIIIZIIIIIIIIXIIIIIIYYIIIXI
37	P0	YYXZ	15	IIIIIYIIXIYIIIIIIIIIIIIXIYIIIIIYIYXIXIYIIIIZIIIIYIIIIIIZXIIIIIIIIY
	P1	YZIY	15	IYXIIIIZIXIIIIYIIIIYIIIIIIIXIIIIIIIIIIIIIIZZIIIIYIIIIYIIIIYXXIIIIZI
38	MX0	YZXY	14	IZIIIIIYIIIIIIYIIIIYIIIIXIIIIXXIIIIIIXIIIZZIIIIIZIIIIYIIIIYIIIIIIYI
	MX1	YYXZ	15	IIXIYIIIIIYIIIIIIYIIXIYIIIIIIIIIIIIZXIIIIIIIIYYXIXIYIIIIZIIIIYIIIII
39	P0	ZIYY	16	YIXZIIIIIIIIIIIIIYIIIIIIZXXIIIIYIIYIYIIIIZZXIIIIIIYIIIIIIIIIIIIIXZ
	P1	ZXZZ	16	IIIIIYIIIIIIZXIYYIIZIYYIIIIIIIXIXIIIIIZIIIIIIIZYIIIIIIZIIIIIIIYYII
40	P0	ZIZX	20	IIIZXIIIIIIZZXIXXYIIYIIXYIIYIIIYIIIYYIIIIZIIIIIXIIYIIZIIIIIIIIIIZI
	P1	ZXZI	18	IIIIIIIIXIIIIIYIIIIIIYIIIYIIZYIIXIIIIXYYZIIIIIIIZIIIIIYXXIIIZIIYIZ