Supporting Information for

A Formally Certified End-to-End Implementation of Shor's Factorization Algorithm

Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, Xiaodi Wu

Corresponding Author: Xiaodi Wu
Email: xiaodiwu@umd.edu

This PDF file includes:

Supporting text
Figs. S1 to S3
SI References

Supporting Information Text

1. Formal Methods and Quantum Programming

A. Proof Assistants. A *proof assistant* is a software tool for formalizing mathematical definitions and stating and proving properties about them. A proof assistant may produce proofs automatically or assist a human in doing so, interactively. Either way, the proof assistant confirms that a proof is correct by employing a *proof verifier*. Since a proof's correctness relies on the verifier being correct, a verifier should be small and simple and the logical rules it checks should be consistent (which is usually proved meta-theoretically).

Most modern proof assistants implement proof verification by leveraging the *Curry-Howard correspondence*, which embodies a surprising and powerful analogy between formal logic and programming language type systems (1, 2). In particular, logical propositions are analogous to programming language types, and proofs are analogous to programs. As an example, the logical implication in proof behaves like a function in programs: Given a proof (program expression a) of proposition (type) A, and a proof that A implies B (a function f of type $A \to B$), we can prove the proposition B (produce a program expression of type B, i.e., via the expression $f(a)$). We can thus represent a proof of a logical formula as a typed expression whose type corresponds to the formula. As a result, proof verification is tantamount to (and implemented as) program type checking.

Machine-aided proofs date back to the Automath project by de Bruijn (3), which was the first practical system exploiting the Curry-Howard correspondence. Inspired by Automath, interactive theorem provers (ITPs) emerged. Most modern proof assistants are ITPs. Milner proposed Stanford LCF (4), introducing *proof tactics*, which allow users to specify particular automated proof search procedures when constructing a proof. A tactic reduces the current proof goal to a list of new subgoals. The process of producing a machine-aided proof is to sequentially apply a list of tactics to transform a proof goal into predefined axioms. Users have direct access to the intermediate subgoals to decide which tactic to apply.

While ITPs were originally developed to formalize mathematics, the use of the Curry-Howard correspondence makes it straightforward to also support writing proved-correct, i.e., *verified*, computer programs. These programs can be *extracted* into runnable code from the notation used to formalize them in the proof assistant.

Modern ITPs are based on different variants of type theories. The ITP employed in this project, Coq (5), is based on the Calculus of Inductive Constructions (6). Coq features propositions as types, higher-order logic, dependent types, and reflections. A variety of proof tactics are included in Coq, like induction. These features have made Coq widely used by the formal methods community.

Coq is a particularly exciting tool that has been used both to verify complex programs and to prove hard mathematical theorems. The archetype of a verified program is the CompCert compiler (7). CompCert compiles code written in the widely used C programming language to instruction sets for ARM, x86, and other computer architectures. Importantly, CompCert's design precisely reflects the intended program behavior—the *semantics*—given in the C99 specification, and all of its optimizations are guaranteed to preserve that behavior. Coq has also been used to verify proofs of the notoriously hard-to-check Four Color Theorem, as well as the Feit–Thompson (or odd order) theorem. Coq's dual uses for both programming and mathematics make it an ideal tool for verifying quantum algorithms.

Coq isn't the only ITP with a number of success stories. The F* language is being used to certify a significant number of internet security protocols, including Transport Layer Security (TLS) (8) and the High Assurance Cryptographic Library, HACL* (9), which has been integrated into the Firefox web browser. Isabelle/HOL was used to verify the seL4 operating system kernel (10). The Lean proof assistant (also based on the Calculus of Inductive Constructions) has been used to verify essentially the entire undergraduate mathematics curriculum and large parts of a graduate curriculum (11). Indeed, Lean has reached the point where it can verify cutting-edge proofs, including a core theorem in Peter Scholze's theory of condensed mathematics, first proven in 2019 (12, 13). Our approach to certifying quantum programs could be implemented using these other tools as well.

B. Debugging Quantum Software by Testing. In the testing scheme, programmers will generate test cases according to the specifications of the desired quantum semantics of the target applications, and execute them on hardware for debugging purposes. Unfortunately, this method does not apply to quantum programs in general.

One approach is through runtime assertions on the intermediate program states during the execution. Intermediate quantum program states, however, will collapse when observed for intermediate values, which implies that assertions could disturb the quantum computation itself. Moreover, many quantum algorithms generate samples over an exponentially large output domain, whose statistical properties could require exponentially many samples to be verified information-theoretically. Together with the fact that quantum hardware is noisy and error-prone, interpreting the readout statistics of quantum hardware for testing purposes is extremely expensive and challenging. One can avoid the difficulty of working with quantum hardware by simulating quantum programs on classical machines, which, however, requires exponential resources in simulation and is not scalable at all. Finally, correctness is only guaranteed in test cases in this scheme.

In the formal methods approach, programmers will develop quantum programs, their desired specifications, and mechanized proofs that the two correspond. All these three components—programs, specifications, and proofs—will be validated statically by the compiler of a proof assistant with built-in support to handle quantum programs. Once everything passes the compiler's check, one has a certified implementation of the target quantum application, which is guaranteed to meet desired specifications on all possible inputs, even without running the program on any real machine.

C. Challenges in formally certifying quantum programs. We have demonstrated the advantages of employing formal verification in the realm of quantum programming. However, due to the distinct properties of quantum programs compared to classical ones, it is necessary to adapt formal techniques to address the unique challenges involved in certifying quantum programs.

First, as quantum mechanics relies on complex vectors to describe states, we need to create a certified complex linear algebra library and establish the semantics of quantum circuits based on it. Since these complex vectors are directly linked to output probability distributions when measuring quantum states, the second challenge involves developing foundations for probabilistic programs. Third, many quantum algorithms consist of both quantum and classical components, necessitating hybrid reasoning that considers both components and their integration. Finally, quantum algorithms consistently demand quantum oracles derived from classical circuit implementations. As a result, tools that facilitate the transformation from certified classical circuits to certified quantum circuits as oracles are crucial.

In the subsequent sections, we tackle these challenges using the tools we have developed in this paper and SQIR, a quantum programming language. We will provide a detailed explanation of how we mechanically certify Shor's algorithm.

D. SQIR. To facilitate proofs about quantum programs, we developed the *small quantum intermediate representation* (SQIR)(14, 15), a circuit-oriented programming language *embedded* in Coq, which means that a SQIR program is defined as a Coq data structure specified using a special syntax, and the semantics of a SQIR program is defined as a Coq function over that data structure (details below). We construct quantum circuits using SQIR, and then state and prove specifications using our Coq libraries for reasoning about quantum programs with complex linear algebra. SQIR programs can be extracted to OpenQASM 2.0 (16), a standard representation for quantum circuits executable on quantum machines.

A SQIR program is a sequence of gates applied to natural number arguments, referring to names (labels) of qubits in a global register. Using predefined gates `SKIP` (no-op), `H` (Hadamard), and `CNOT` (controlled *not*) in SQIR, a circuit that generates the Greenberger–Horne–Zeilinger (GHZ) state with three qubits in Coq is defined by

```
Definition GHZ3 : ucom base 3 := H 0; CNOT 0 1; CNOT 0 2.
```

The type `ucom base 3` says that the resulting circuit is a unitary program that uses our base gate set and three qubits. Inside this circuit, three gates are sequentially applied to the qubits. More generally, we could write a Coq function that produces a GHZ state generation circuit: Given a parameter n, function `GHZ` produces the n-qubit GHZ circuit.

```
Fixpoint GHZ (n : ℕ) : ucom base n :=
  match n with
  | 0 ⇒ SKIP
  | 1 ⇒ H 0
  | S (S n') ⇒ GHZ (S n'); CNOT n' (S n')
  end.
```

These codes define a recursive program `GHZ` on one natural number input `n` through the use of `match` statement. Specifically, `match` statement returns `SKIP` when `n=0`, `H 0` when `n=1`, and recursively calls on itself for `n-1` otherwise. One can observe that `GHZ 3` (calling `GHZ` with argument 3) will produce the same SQIR circuit as definition `GHZ3`, above.

The function `uc_eval` defines the semantics of a SQIR program, essentially by converting it to a unitary matrix of complex numbers. This matrix is expressed using axiomatized reals from the Coq Standard Library (17), complex numbers from Coquelicot (18), and the complex matrix library from QWIRE (19). Using `uc_eval`, we can state properties about the behavior of a circuit. For example, the specification for `GHZ` says that it produces the mathematical GHZ state when applied to the all-zero input.

```
Theorem GHZ_correct : ∀ n : ℕ, 0 < n →
    uc_eval (GHZ n) × |0⟩^⊗n = 1/√2 * |0⟩^⊗n + 1/√2 * |1⟩^⊗n.
```

This theorem can be proved in Coq by induction on n. By converting the quantum circuits programmed in SQIR into matrices via `uc_eval`, we can formally verify it generating the GHZ state when applying on the initial state $|0\rangle^{\otimes n}$. In this way, we can reason about unitary quantum circuits of arbitrary size on arbitrary input states. Compared to classical programs, quantum programs use complex vectors here to represent the states and complex matrices to represent the semantics, which is implemented using SQIR's linear algebra library.

We also exhibit how quantum Fourier transformation, a key part of Shor's algorithm, is programmed and certified. We can formulate the quantum Fourier transformation (QFT) sub-procedure and its correctness. The QFT program in SQIR reads:

```
Fixpoint controlled_rotations n : base_ucom n :=
  match n with
  | 0 | 1 ⇒ SKIP
  | 2     ⇒ control 1 (Rz (2 * PI / 2 ^ n) 0)
  | S n'  ⇒ cast (controlled_rotations n') n ;
             control n' (Rz (2 * PI / 2 ^ n) 0)
  end.

Fixpoint QFT n : base_ucom n :=
  match n with
  | 0    ⇒ SKIP
  | 1    ⇒ H 0
  | S n' ⇒ H 0 ; controlled_rotations n ;
           cast (map_qubits S (QFT n')) n
  end.
```

Here `control n G` creates a gate `G` controlled by the `n`-th qubit in the circuit, and `map_qubits S c` relabels the qubit in circuit `c` by adding 1 to each label. Similar to the GHZ example above, to specify the correctness of `QFT`, we state that its corresponding unitary transforms an initial state $|\mathtt{f}\rangle$ for binary vector `f` to the Fourier state of `f`:

```
Lemma QFT_semantics : ∀ n f, n > 0 →
  uc_eval (QFT n) × (f_to_vec n f) = 1/√2^n ⨂_{i=0}^{n-1} (|0⟩ .+ Cexp (2π/2^{n-i} * (funbool_to_nats (n - i) (shift f i))) .* |1⟩).
```

E. Certifying Quantum Phase Estimation (QPE) in SQIR. To date, SQIR has been used to implement and verify a number of quantum algorithms (15), including quantum teleportation, GHZ state preparation, the Deutsch-Jozsa algorithm, Simon's algorithm, the quantum Fourier transform (QFT), Grover's algorithm, and quantum phase estimation (QPE). QPE is a key component of Shor's prime factoring algorithm (described in the next section), which finds the eigenvalue of a quantum program's eigenstates. Here we exhibit how we accommodate formal methods for quantum programs via the QPE program.

Then we can define QPE in SQIR as follows:

```
Fixpoint controlled_powers {n} f k kmax :=
  match k with
  | 0    ⇒ SKIP
  | 1    ⇒ control (kmax-1) (f 0)
  | S k' ⇒ controlled_powers f k' kmax ;
            control (kmax-k'-1) (f k')
  end.
Definition QPE k n (f : ℕ → base_ucom n) :=
  let f' := (fun x ⇒ map_qubits (fun q ⇒ k+q) (f x)) in
  npar k U_H ;
  controlled_powers f' k k ;
  invert (QFT k).
```

`QPE` takes as input the precision `k` of the resulting estimate, the number `n` of qubits used in the input program, and a circuit family `f`. `QPE` includes three parts: (1) `k` parallel applications of Hadamard gates; (2) exponentiation of the target unitary; (3) an inverse QFT procedure. (1) and (3) are implemented by recursive calls in SQIR via methods `npar` and `QFT`. Our implementation of (2) inputs a mapping from natural numbers representing which qubit is the control, to circuits implementing repetitions of the target unitary, since normally the exponentiation is decomposed into letting the x-th bit control 2^x repetition of the target unitary. Then `controlled_powers` recursively calls itself, to map the circuit family on the first `n` qubits to the exponentiation circuit. In Shor's algorithm, (2) is efficiently implemented by applying controlled in-place modular multiplications with pre-calculated multipliers, whose implementation and certification details are presented in the next section via a new intermediate representation dealing with classical reversible circuits and their transformation to quantum circuits.

To deal with measurements of quantum programs, we intuitively analyze the output probability distribution entry by entry, calculating the probability of obtaining each possible output. This effectively realizes a simplified reasoning system for probabilistic programs, since we do not want to formalize a complicated probability theory. The correctness of `QPE` is elaborated in (15) with the following statement proved in Coq:

```
Lemma QPE_semantics_full : ∀ (k n z : ℕ) (c : base_ucom n) (ψ : Vector (2 ^ n)) (δ : R),
  n > 0 → k > 1 → uc_well_typed c → Pure_State_Vector ψ →
  -1/2^{k+1} ≤ δ < 1/2^{k+1} → let θ := (z/2^k + δ) in (uc_eval c) × ψ = e^{2πiθ} .* ψ →
  probability_of_outcome (|z⟩_k ⊗ ψ) ((uc_eval (QPE k n c)) × (|0⟩^{⊗k} ⊗ ψ)) ≥ 4/π^2.
```

This lemma specifies the correctness of `QPE`: when the circuit `c` has an eigenvalue θ approximated the best by $2\pi z/2^k$ for a z selected from $\{0, ..., 2^k - 1\}$, the probability of obtaining $|z\rangle_k$ (the computational basis state where `z` is expressed in the binary form with length `k`) when measuring the result of executing `QPE k n c` is at least $\frac{4}{\pi^2}$.

The proof of QPE's correctness first analyzes the unitary matrices representing the semantics of sub-procedures and expresses the quantum state before measurement as a series of linear transformations applied to the input state. Then we algebraically calculate the probability of obtaining output `z` as $|(\langle z|_k \otimes \psi) \times (\texttt{uc\_eval (QPE k n c)}) \times (|0\rangle_k \otimes \psi)|^2 = |\sum_{j=0}^{2^k-1} e^{2\pi i j \delta}/2^k|^2$. Then we discuss whether $\delta = 0$ and algebraically express the probability of obtaining $|z\rangle_k$. The probability for $\delta = 0$ is 1. For $\delta \neq 0$, we use a series of inequalities on trigonometric functions to obtain a lower bound of $4/\pi^2$ with the help of the Coq-Interval library which deals with algebra expressions.

When QPE is connected to the classical post-processing of Shor's algorithm, to certify the correctness of the hybrid procedure, we enumerate all possible outputs, collect those leading to a successful run, and calculate their probability summation as the success probability.

2. Shor's Algorithm and Its Implementation

Shor's factorization algorithm consists of two parts. The first employs a hybrid classical-quantum algorithm to solve the order-finding problem; the second reduces factorization to order-finding. In this section, we present an overview of Shor's algorithm (see Figure 2 for a summary). In next sections, we discuss details about our implementation (see Figure 3) and certified correctness properties.

We give a brief introduction to the procedure of Shor's algorithm, whose details are presented later. The classical pre-processing will identify cases where N is prime, even, or a prime power, which can be efficiently tested for and solved by classical algorithms. Otherwise, one will proceed to the main part of Shor's algorithm (enclosed in the green frame) to solve the case where $N = p^k q$. One starts with a random integer sample a between 1 and N. When a is a co-prime of N, i.e., the greatest common divisor $\gcd(a, N) = 1$, the algorithm leverages a quantum computer and classical post-processing to find the order r of a modulo N (i.e., the smallest positive integer r such that $a^r \equiv 1(\mod N)$). The quantum part of order finding involves quantum phase estimation (QPE) on modular multipliers for (a, N). The classical post-processing finds the

continued fraction expansion (CFE) $[a_1, a_2, \cdots, a_{2m}]$ of the output $s/2^m \approx k/r$ of quantum phase estimation to recover the order r. Further classical post-processing will rule out cases where r is odd before outputting the non-trivial factor. To formally prove the correctness of the implementation, we first prove separately the correctness of the quantum component (i.e., QPE with in-place modular multiplier circuits for any (a, N) on n bits) and the classical component (i.e., the convergence and the correctness of the CFE procedure). We then integrate them to prove that with one randomly sampled a, the main part of Shor's algorithm, i.e., the quantum order-finding step sandwiched between the pre and post classical processing, will succeed in identifying a non-trivial factor of N with probability at least $1/\text{polylog}(N)$. By repeating this procedure $\text{polylog}(N)$ times, our certified implementation of Shor's algorithm is guaranteed to find a non-trivial factor with a success probability close to 1.

Our focus is on implementation over an ideal quantum computer, where there is no gate implementation error and no topology constraints. In practice, error correction codes become necessary to correct errors caused by system decoherence or state leakage. This involves using multiple physical qubits to represent a logical qubit, which can significantly increase the circuit size depending on the precision of the quantum computer. Incorporating error correction into a certified implementation of Shor's algorithm is left as a future research direction.

A. A Hybrid Algorithm for Order Finding. The multiplicative order of a modulo N, represented by $\text{ord}(a, N)$, is the least integer r larger than 1 such that $a^r \equiv 1 \pmod N$. Calculating $\text{ord}(a, N)$ is hard for classical computers, but can be efficiently solved with a quantum computer, for which Shor proposed a hybrid classical-quantum algorithm (20). This algorithm has three major components: (1) in-place modular multiplication on a quantum computer; (2) quantum phase estimation; (3) continued fraction expansion on a classical computer.

In-place Modular Multiplication An in-place modular multiplication operator $IMM(a, N)$ on n working qubits and s ancillary qubits satisfies the following property:

$$\forall x < N, \quad IMM(a, N)|x\rangle_n|0\rangle_s = |(a \cdot x) \bmod N\rangle_n|0\rangle_s,$$

where $0 < N < 2^{n-1}$. It is required that a and N are co-prime, otherwise the operator is irreversible. This requirement implies the existence of a multiplicative inverse a^{-1} modulo N such that $a \cdot a^{-1} \equiv 1 \pmod N$.

Quantum Phase Estimation Given a subroutine U and an eigenvector $|\psi\rangle$ with eigenvalue $e^{i\theta}$, quantum phase estimation (QPE) finds the closest integer to $\frac{\theta}{2\pi}2^m$ with high success probability, where m is a predefined precision parameter.

Shor's algorithm picks a random a from $[1, N)$ first, and applies QPE on $IMM(a, N)$ on input state $|0\rangle_m |1\rangle_n |0\rangle_s$ where $m = \lfloor \log_2 2N^2 \rfloor, n = \lfloor \log_2 2N \rfloor$ and s is the number of ancillary qubits used in $IMM(a, N)$. Then a computational basis measurement is applied on the first m qubits, generating an output integer $0 \leq \texttt{out} < 2^m$. The distribution of the output has $\text{ord}(a, N)$ peaks, and these peaks are almost equally spaced. We can extract the order by the following procedure.

Continued Fraction Expansion The post-processing of Shor's algorithm invokes the continued fraction expansion (CFE) algorithm. A k-level continued fraction is defined recursively by

$$\langle\rangle = 0,$$
$$\langle a_1, a_2, ..., a_k\rangle = \frac{1}{a_1 + \langle a_2, a_3, ..., a_k\rangle}.$$

k-step CFE finds a k-level continued fraction to approximate a given real number. For a rational number $0 \leq \frac{a}{b} < 1$, the first term of the expansion is $\lfloor \frac{b}{a} \rfloor$ if $a \neq 0$, and we recursively expand $\frac{b \bmod a}{a}$ for at most k times to get an approximation of $\frac{a}{b}$ by a k-level continued fraction. In Coq, the CFE algorithm is implemented as

```
Fixpoint CFE_ite (k a b p1 q1 p2 q2 : ℕ) : ℕ × ℕ :=
  match k with
  | 0 ⇒ (p1, q1)
  | S k' ⇒ if a = 0 then (p1, q1)
            else let (c, d) := (⌊b/a⌋, b mod a) in
            CF_ite k' d a (c · p1 + p2) (c · q1 + q2) p1 q1
  end.
Definition CFE k a b := snd (CF_ite (k+1) a b 0 1 1 0).
```

Function `CFE_ite` takes in the number of iterations k, target fraction `a/b`, the fraction from the $(k-1)$-step expansion, and the $(k-2)$-step expansion. Function `CFE k a b` represents the denominator in the simplified fraction equal to the k-level continued fraction that is the closest to $\frac{a}{b}$.

The post-processing of Shor's algorithm expands $\frac{\texttt{out}}{2^m}$ using CFE, where `out` is the measurement result and `m` is the precision for QPE defined above. It finds the minimal step `k` such that $a^{\texttt{CFE k out } 2^m} \equiv 1 \pmod N$ and $k \leq 2m + 1$. With probability no less than $1/\text{polylog}(N)$, there exists k such that `CFE k out` 2^m is the multiplicative order of a modulo N. We can repeat the QPE and post-processing for $\text{polylog}(N)$ times. Then the probability that the order exists in one of the results can be arbitrarily close to 1. The minimal valid post-processing result is highly likely to be the order.

B. Reduction from Factorization to Order Finding. To completely factorize composite number N, we only need to find one non-trivial factor of N (i.e., a factor that is not 1 nor N). If a non-trivial factor d of N can be found, we can recursively solve the problem by factorizing d and $\frac{N}{d}$ separately. Because there are at most $\log_2(N)$ prime factors of N, this procedure repeats for at most polylog(N) times. A classical computer can efficiently find a non-trivial factor in the case where N is even or $N = p^k$ for prime p. However, Shor's algorithm is the only known (classical or quantum) algorithm to efficiently factor numbers for which neither of these is true.

Shor's algorithm randomly picks an integer $1 \leq a < N$. If the greatest common divisor $\gcd(a, N)$ of a and N is a non-trivial factor of N, then we are done. Otherwise we invoke the hybrid order finding procedure to find $\mathrm{ord}(a, N)$. With probability no less than one half, one of $\gcd\left(a^{\lfloor \frac{\mathrm{ord}(a,N)}{2} \rfloor} \pm 1, N\right)$ is a non-trivial factor of N. Note that $\gcd\left(a^{\lfloor \frac{\mathrm{ord}(a,N)}{2} \rfloor} \pm 1, N\right)$ can be efficiently computed by a classical computer(21). By repeating the random selection of a and the above procedure for constant times, the success probability to find a non-trivial factor of N is close to 1.

C. Implementation of Modular Multiplication. One of the pivoting components of Shor's order finding procedure is a quantum circuit for in-place modular multiplication (IMM). We initially tried to define this operation in SQIR but found that for purely classical operations (that take basis states to basis states), SQIR's general quantum semantics makes proofs unnecessarily complicated. In response, we developed the *reversible circuit intermediate representation* (RCIR) to express classical functions and prove their correctness. RCIR programs can be translated into SQIR, and we prove this translation correct.

RCIR RCIR contains a universal set of constructs on classical bits labeled by natural numbers. The syntax is:

$$R \;:=\; \texttt{skip} \;\mid\; \texttt{X}\ n \;\mid\; \texttt{ctrl}\ n\ R \;\mid\; \texttt{swap}\ m\ n \;\mid\; R_1;\ R_2.$$

Here `skip` is a unit operation with no effect, `X` n flips the n-th bit, `ctrl` n R executes subprogram R if the n-th bit is 1 and otherwise has no effect, `swap` m n swaps the m-th and n-th bits, and $R_1; R_2$ executes subprograms R_1 and R_2 sequentially. We remark that `swap` is not necessary for the expressiveness of the language, since it can be decomposed into a sequence of three `ctrl` and `X` operations. We include it here to facilitate `swap`-specific optimizations of the circuit.

As an example, we show the RCIR code for the MAJ (majority) operation (22), which is an essential component of the ripple-carry adder.

```
Definition MAJ a b c :=
  ctrl c (X b) ; ctrl c (X a) ; ctrl a (ctrl b (X c)).
```

It takes in three bits labeled by a, b, c whose initial values are v_a, v_b, v_c correspondingly, and stores v_a `xor` v_c in a, v_b `xor` v_c in b, and $MAJ(v_a, v_b, v_c)$ in c. Here $MAJ(v_a, v_b, v_c)$ is the majority of v_a, v_b and v_c, the value that appears at least twice.

To reverse a program written in this syntax, we define a reverse operator by $\texttt{skip}^{\texttt{rev}} = \texttt{skip}$, $(\texttt{X}\ n)^{\texttt{rev}} = \texttt{X}\ n$, $(\texttt{ctrl}\ n\ R)^{\texttt{rev}} = \texttt{ctrl}\ n\ R^{\texttt{rev}}$, $(\texttt{swap}\ m\ n)^{\texttt{rev}} = \texttt{swap}\ m\ n$, $(R_1;\ R_2)^{\texttt{rev}} = R_2^{\texttt{rev}};\ R_1^{\texttt{rev}}$. We prove that the reversed circuit will cancel the behavior of the original circuit.

We can express the semantics of a RCIR program as a function between Boolean registers. We use notation $[k]_n$ to represent an n-bit register storing natural number $k < 2^n$ in binary representation. Consecutive registers are labeled sequentially by natural numbers. If $n = 1$, we simplify the notation to [0] or [1].

The translation from RCIR to SQIR is natural since every RCIR construct has a direct correspondence in SQIR. The correctness of this translation states that the behavior of a well-typed classical circuit in RCIR is preserved by the generated quantum circuit in the context of SQIR. That is, the translated quantum circuit turns a state on the computational basis into another one corresponding to the classical state after the execution of the classical reversible circuit.

Details of IMM Then the goal is to construct a reversible circuit $IMM_c(a, N)$ in RCIR satisfying

$$\forall x < N, \quad [x]_n[0]_s \xrightarrow{IMM_c(a,N)} [a \cdot x \bmod N]_n[0]_s. \qquad \text{[C.1]}$$

so that we can translate it into a quantum circuit in SQIR. The overview of our implementation of such a circuit is presented in Figure S1. We present it in detail below.

Since we will encounter bit-level logic frequently, we define $x \odot_? y$ for binary logic operator $\odot$ as a binary expression over x and y whose value is 1 if $x \odot y$ is true and 0 otherwise. For example, $3 \geq_? 2 = 1$ and $3 \leq_? 2 = 0$ since $3 \geq 2$. This notation also carries over into our Coq code, e.g. `x<?y` is the expression $x <_? y$.

Adapting the standard practice (23), we implement modular multiplication based on repeated modular additions. For addition, we use Cuccaro et al.'s ripple-carry adder (RCA) (22). RCA realizes the transformation

$$[c][x]_n[y]_n \xrightarrow{RCA} [c][x]_n[(x + y + c) \bmod 2^n]_n,$$

for ancillary bit $c \in \{0, 1\}$ and inputs $x, y < 2^{n-1}$. We use Cucarro et al.'s RCA-based definitions of subtractor (SUB) and comparator (CMP), and we additionally provide a n-qubit register swapper (SWP) and shifter (SFT) built using swap gates.

These components realize the following transformations:

$$[0][x]_n[y]_n \xrightarrow{SUB} [0][x]_n[(y-x) \bmod 2^n]_n$$
$$[0][x]_n[y]_n \xrightarrow{CMP} [x \geq_? y][x]_n[y]_n$$
$$[x]_n[y]_n \xrightarrow{SWP} [y]_n[x]_n$$
$$[x]_n \xrightarrow{SFT} [2x]_n$$

We remark here that SFT is correct only when $x < 2^{n-1}$. With these components, we can build a modular adder (`ModAdd`) and modular shifter (`ModSft`) using two ancillary bits at positions 0 and 1.

```
Definition ModAdd n :=
    SWP02 n; RCA n; SWP02 n; CMP n;
    ctrl 1 (SUB n); SWP02 n; (CMP n)^rev; SWP02 n.
Definition ModSft n := SFT n; CMP n; ctrl 1 (SUB n).
```

$\texttt{SWP}_{02}$ is the register swapper applied to the first and third n-bit registers. These functions realize the following transformations:

$$[0][0][N]_n[x]_n[y]_n \xrightarrow{\texttt{ModAdd}} [0][0][N]_n[x]_n[(x+y) \bmod N]_n$$
$$[0][0][N]_n[x]_n \xrightarrow{\texttt{ModSft}} [0][N \leq_? 2x][N]_n[2x \bmod N]_n$$

Note that $(a \cdot x) \bmod N$ can be decomposed into

$$(a \cdot x) \bmod N = \left(\sum_{i=0}^{n-1} (1 \leq_? a_i) \cdot 2^i \cdot x \right) \bmod N,$$

where a_i is the i-th bit in the little-endian binary representation of a. By repeating `ModSft`s and `ModAdd`s, we can perform $(a \cdot x) \bmod N$ according to this decomposition, eventually generating a circuit for modular multiplication on two registers ($MM(a, N)$), which implements

$$[x]_n[0]_n[0]_s \xrightarrow{MM(a,N)} [x]_n[a \cdot x \bmod N]_n[0]_s.$$

Here s is the number of additional ancillary qubits, which is linear to n. Finally, to make the operation in-place, we exploit the modular inverse a^{-1} modulo N:

```
Definition IMM a N n :=
  MM a N n; SWP01 n; (MM a^-1 N n)^rev.
```

There is much space left for optimization in this implementation. Other approaches in the literature (24–28) may have a lower depth or fewer ancillary qubits. We chose this approach because its structure is cleaner to express in our language, and its asymptotic complexity is feasible for efficient factorization, which makes it great for mechanized proofs.

D. Implementation of Shor's algorithm. Our final definition of Shor's algorithm in Coq uses the `IMM` operation along with a SQIR implementation of `QPE` described in the previous sections. The quantum circuit to find the multiplicative order $\text{ord}(a, N)$ is then

```
Definition shor_circuit a N :=
  let m := log2 (2*N^2) in
  let n := log2 (2*N) in
  let f i := IMM (modexp a (2^i) N) N n in
  X (m + n - 1); QPE m f.
```

We can extract the distribution of the result of the random procedure of Shor's factorization algorithm

```
Definition factor (a N r : ℕ) :=
  let cand1 := Nat.gcd (a ^ (r / 2) - 1) N in
  let cand2 := Nat.gcd (a ^ (r / 2) + 1) N in
  if (1 <? cand1) && (cand1 <? N) then Some cand1
  else if (1 <? cand2) && (cand2 <? N) then Some cand2
  else None.
Definition shor_body N rnd :=
  let m := log2 (2*N^2) in
  let k := 4*log2 (2*N)+11 in
  let distr := join (uniform 1 N)
               (fun a ⇒ run (to_base_ucom (m+k)
                         (shor_circuit a N))) in
  let out := sample distr rnd in
  let a := out / 2^(m+k)) in
  let x := (out mod (2^(m+k))) / 2^k in
  if Nat.gcd a N =? 1%ℕ
  then factor a N (OF_post a N x n)
  else Some (Nat.gcd a N).
Definition end_to_end_shor N rnds :=
  iterate rnds (shor_body N).
```

Here `factor` is the reduction finding non-trivial factors from multiplicative order, `shor_body` generates the distribution and sampling from it, and `end_to_end_shor` iterates `shor_body` for multiple times and returns a non-trivial factor if any of them succeeds.

3. Certification of the Implementation

In this section, we summarize the facts we have proved in Coq to fully verify Shor's algorithm, as presented in the previous section.

A. Certifying Order Finding. For the hybrid order finding procedure in Appendix A, we verify that the success probability is at least 1/polylog(N). Recall that the quantum part of order finding uses in-place modular multiplication ($IMM(a, N)$) and quantum phase estimation (QPE). The classical part applies continued fraction expansion to the outcome of quantum measurements. Our statement of order finding correctness says:

```
Lemma Shor_OF_correct :
  ∀ (a N : ℕ),
    (1 < a < N) → (gcd a N = 1) →
    ℙ[Shor_OF a N = ord a N] ≥ β / ⌊log2(N)⌋^4.
```

where $\beta = \frac{4e^{-2}}{\pi^2}$. The probability sums over possible outputs of the quantum circuit and tests if post-processing finds `ord a N`.

Certifying IMM We have proved that our RCIR implementation of IMM satisfies (C.1). Therefore, because we have a proved-correct translator from RCIR to SQIR, our SQIR translation of IMM also satisfies this property. In particular, the in-place modular multiplication circuit $IMM(a, N)$ with n qubits to represent the register and s ancillary qubits, translated from RCIR to SQIR, has the following property for any $0 \leq N < 2^n$ and $a \in \mathbb{Z}_N$:

```
Definition IMMBehavior a N n s c :=
  ∀ x : ℕ, x < N →
    (uc_eval c) × (|x⟩_n ⊗ |0⟩_s) = |a · x mod N⟩_n ⊗ |0⟩_s.
Lemma IMM_correct a N :=
  let n := log2 (2*N) in
  let s := 3*n + 11 in
  IMMBehavior a N n s (IMM a n).
```

Here `IMMBehavior` depicts the desired behavior of an in-place modular multiplier, and we have proved the constructed $IMM(a, N)$ satisfies this property.

Certifying QPE over IMM We certify that QPE outputs the closest estimate of the eigenvalue's phase corresponding to the input eigenvector with probability no less than $\frac{4}{\pi^2}$:

```
Lemma QPE_semantics :
  ∀ m n z δ (f : ℕ → base_ucom n) (|ψ⟩ : Vector 2^n),
    n > 0 → m > 1 → -1/2^(m+1) ≤ δ < 1/2^(m+1) →
    Pure_State_Vector |ψ⟩ →
    (∀ k, k < m →
      uc_WT (f k) ∧ (uc_eval (f k)) |ψ⟩ = e^(2^(k+1) π i (z/2^m + δ)) |ψ⟩) →
    ‖⟨z, ψ| (uc_eval (QPE k n f)) |0, ψ⟩‖^2 ≥ 4/π^2.
```

To utilize this lemma with $IMM(a, N)$, we first analyze the eigenpairs of $IMM(a, N)$. Let $r = \text{ord}(a, N)$ be the multiplicative order of a modulo N. We define

$$|\psi_j\rangle_n = \frac{1}{\sqrt{r}} \sum_{l<r} \omega_r^{-j \cdot l} |a^l \mod N\rangle_n$$

in SQIR and prove that it is an eigenvector of any circuit satisfying `IMMBehavior`, including $IMM(a^{2^k}, N)$, with eigenvalue $\omega_r^{j \cdot 2^k}$ for any natural number k, where $\omega_r = e^{\frac{2\pi i}{r}}$ is the r-th primitive root in the complex plane.

```
Lemma IMMBehavior_eigenpair :
  ∀ (a r N j n s k : ℕ) (c : base_ucom (n+s)),
    Order a r N → N < 2^n →
    IMMBehavior a^(2^k) N n s c →
    (uc_eval (f k)) |ψ_j⟩_n ⊗ |0⟩_s = e^(2^(k+1) π i j/r) |ψ_j⟩_n ⊗ |0⟩_s.
```

Here `Order a r N` is a proposition specifying that r is the order of a modulo N. Because we cannot directly prepare $|\psi_j\rangle$, we actually set the eigenvector register in QPE to the state $|1\rangle_n \otimes |0\rangle_s$ using the identity:

```
Lemma sum_of_ψ_is_one :
  ∀ a r N n : ℕ,
    Order a r N → N < 2^n → 1/√r ∑_{k<r} |ψ_j⟩_n = |1⟩_n.
```

By applying `QPE_semantics`, we prove that for any $0 \leq k < r$, with probability no less than $\frac{4}{\pi^2 r}$, the result of measuring QPE applied to $|0\rangle_m \otimes |1\rangle_n \otimes |0\rangle_s$ is the closest integer to $\frac{k}{r} 2^m$.

Certifying Post-processing Our certification of post-processing is based on two mathematical results (also formally certified in Coq): the lower bound of Euler's totient function and the Legendre's theorem for continued fraction expansion. Let $\mathbb{Z}_n^*$ be the integers smaller than n and coprime to n. For a positive integer n, Euler's totient function $\varphi(n)$ is the size of $\mathbb{Z}_n^*$. They are formulated in Coq as follows.

```
Theorem Euler_totient_lb : ∀ n, n ≥ 2 → φ(n)/n ≥ e^-2 / ⌊log2 n⌋^4.
Lemma Legendre_CFE :
  ∀ a b p q : ℕ,
    a < b → gcd p q = 1 → 0 < q → |a/b - p/q| < 1/(2q^2) →
    ∃ s, s ≤ 2 log2(b) + 1 ∧ CFE s a b = q.
```

The verification of these theorems is discussed later.

By Legendre's theorem for CFE, there exists a $s \leq 2m+1$ such that `CFE s out` $2^m = r$, where `out` is the closest integer to $\frac{k}{r}2^m$ for any $k \in \mathbb{Z}_r^*$. Hence the probability of obtaining the order (r) is the sum $\sum_{k\in\mathbb{Z}_r^*} \frac{4}{\pi^2 r}$. Note that $r \leq \varphi(N) < N$. With the lower bound on Euler's totient function, we obtain a lower bound of $1/\text{polylog}(N)$ of successfully obtaining the order $r = \text{ord}(a, N)$ through the hybrid algorithm, finishing the proof of `Shor_OF_correct`.

Lower Bound of Euler's Totient Function We build our proof on the formalization of Euler's product formula and Euler's theorem by de Rauglaudre (29). By rewriting Euler's product formula into exponents, we can scale the formula into exponents of Harmonic sequence $\sum_{0<i\leq n} \frac{1}{i}$. Then an upper bound for the Harmonic sequence suffices for the result.

In fact, a tighter lower bound of Euler's totient function exists (30), but obtaining it involves evolved mathematical techniques which are hard to formalize in Coq since they involved analytic number theory. Fortunately, the formula certified above is sufficient to obtain a success probability of at least $1/\text{polylog}(N)$ for factorizing N.

Legendre's Theorem for Continued Fraction Expansion The proof of Legendre's theorem consists of facts: (1) `CFE s a b` monotonically increases, and reaches b within $2\log_2(b)+1$ steps, and (2) for `CFE s a b` $\leq$ `q` $<$ `CFE (s+1) a b` satisfying $\left|\frac{a}{b} - \frac{p}{q}\right| < \frac{1}{2q^2}$, the only possible value for `q` is `CFE s a b`. These are certified following basic analysis to the continued fraction expansion(31).

B. Certifying Shor's Reduction. We formally certify that for half of the possible choices of a, `ord a n` can be used to find a nontrivial factor of N:

```
Lemma reduction_fact_OF :
  ∀ (p k q N : ℕ),
    k > 0 → prime p → 2 < p → 2 < q →
    gcd p q = 1 → N = p^k * q →
    |ℤ_N| ≤ 2 · Σ_{a∈ℤ_N} [1 < gcd (a^⌊ord a N / 2⌋ ± 1) N < N].
```

The expression `[1 < (gcd (a^⌊ord a N / 2⌋ ± 1) N) < N]` equals to 1 if at least one of `gcd(a^⌊ord a N / 2⌋ + 1, N)` or `gcd(a^⌊ord a N / 2⌋ - 1, N)` is a nontrivial factor of N, otherwise it equals to 0. In the following we illustrate how we achieve this lemma.

From 2-adic Order to Non-Trivial Factors The proof proceeds as follows: Let $d(x)$ be the largest integer i such that 2^i is a factor of x, which is also known as the 2-adic order. We first certify that $d(\text{ord}(a, p^k)) \neq d(\text{ord}(a, q))$ indicates $a^{\lfloor \frac{\text{ord}(a,N)}{2} \rfloor} \not\equiv \pm 1 \pmod N$

```
Lemma d_neq_sufficient :
  ∀ a p q N,
    2 < p → 2 < q → gcd p q = 1 → N = pq →
    d (ord a p) ≠ d (ord a q) →
    a^⌊ord a N / 2⌋ ≢ ±1 (mod N).
```

This condition is sufficient to get a nontrivial factor of N by Euler's theorem and the following lemma

```
Lemma sqr1_not_pm1 :
  ∀ x N,
    1 < N → x^2 ≡ 1 (mod N) → x ≢ ±1 (mod N) →
    1 < gcd (x - 1) N < N ∨ 1 < gcd (x + 1) N < N.
```

By the Chinese remainder theorem, randomly picking a in $\mathbb{Z}_N$ is equivalent to randomly picking b in $\mathbb{Z}_{p^k}$ and randomly picking c in $\mathbb{Z}_q$. $a \equiv b \mod p^k$ and $a \equiv c \mod q$, so $\text{ord}(a, p^k) = \text{ord}(b, p^k)$ and $\text{ord}(a, q) = \text{ord}(c, q)$. Because the random pick of b is independent from the random pick of c, it suffices to show that for any integer i, at least half of the elements in $\mathbb{Z}_{p^k}$ satisfy $d(\text{ord}(x, p^k)) \neq i$.

Detouring to Quadratic Residue Shor's original proof(20) of this property made use of the existence of a group generator of $\mathbb{Z}_{p^k}$, also known as primitive roots, for odd prime p. But the existence of primitive roots is non-constructive, hence hard to present in Coq. We manage to detour from primitive roots to quadratic residues in modulus p^k in order to avoid non-constructive proofs.

A quadratic residue modulo p^k is a natural number $a \in \mathbb{Z}_{p^k}$ such that there exists an integer x with $x^2 \equiv a \mod p^k$. We observe that a quadratic residue $a \in \mathbb{Z}_{p^k}$ will have $d(\text{ord}(x, p^k)) < d(\varphi(p^k))$, where φ is the Euler's totient function. Conversely, a quadratic non-residue $a \in \mathbb{Z}_{p^k}$ will have $d(\text{ord}(x, p^k)) = d(\varphi(p^k))$:

```
Lemma qr_d_lt :
  ∀ a p k,
    k ≠ 0 → prime p → 2 < p →
    (∃ x, x^2 ≡ a mod p^k) →
    d (ord a p^k) < d (φ (p^k)).
Lemma qnr_d_eq :
  ∀ a p k,
```

```
    k ≠ 0 → prime p → 2 < p →
    (∀ x, x^2 ≢ a  mod p^k) →
    d (ord a p^k) = d (φ (p^k)).
```

These lemmas are obtained via Euler's Criterion, which describes the difference between multiplicative orders of quadratic residues and quadratic non-residues. The detailed discussion is put later.

We claim that the number of quadratic residues in $\mathbb{Z}_{p^k}$ equals to the number of quadratic non-residues in $\mathbb{Z}_{p^k}$, whose detailed verification is left later. Then no matter what i is, at least half of the elements in $\mathbb{Z}_{p^k}$ satisfy $d(\mathrm{ord}(x, p^k)) \neq i$. This makes the probability of finding an $a \in \mathbb{Z}_{p^k q}$ satisfying $d(\mathrm{ord}(a, p^k)) \neq d(\mathrm{ord}(a, q))$ at least one half, in which case one of $\texttt{gcd}\ \left(\texttt{a}^{\lfloor \frac{\texttt{ord a N}}{2} \rfloor} \pm 1\right)\ \texttt{N}$ is a nontrivial factor of N.

Euler's Criterion We formalize a generalized version of Euler's criterion: for odd prime p and $k > 0$, whether an integer $a \in \mathbb{Z}_{p^k}$ is a quadratic residue modulo p^k is determined by the value of $a^{\frac{\varphi(p^k)}{2}} \mod p^k$.

```
Lemma Euler_criterion_qr :
  ∀ a p k,
    k ≠ 0 → prime p → 2 < p → gcd a p = 1 →
    (∃ x, x^2 ≡ a  mod p^k) →
    a^(φ(p^k)/2)  mod p^k = 1.
Lemma Euler_criterion_qnr :
  ∀ a p k,
    k ≠ 0 → prime p → 2 < p → gcd a p = 1 →
    (∀ x, x^2 ≢ a  mod p^k) →
    a^(φ(p^k)/2)  mod p^k = p^k - 1.
```

These formulae can be proved by a pairing function over $\mathbb{Z}_{p^k}$:

$$x \mapsto (a \cdot x^{-1}) \mod p^k,$$

where x^{-1} is the multiplicative inverse of x modulo p^k. For a quadratic residue a, only the two solutions of $x^2 \equiv a \mod p^k$ do not form pairing: each of them maps to itself. For each pair (x, y) there is $x \cdot y \equiv a \mod p^k$, so reordering the product $\prod_{x \in \mathbb{Z}_{p^k}} x$ with this pairing proves the Euler's criterion.

With Euler's criterion, we can reason about the 2-adic order of multiplicative orders for quadratic residues and quadratic non-residues, due to the definition of multiplicative order and $ord(a, p^k) | \varphi(p^k)$.

Counting Quadratic Residues Modulo p^k For odd prime p and $k > 0$, there are exactly $\varphi(p^k)/2$ quadratic residues modulo p^k in $\mathbb{Z}_{p^k}$, and exactly $\varphi(p^k)/2$ quadratic non-residues.

```
Lemma qr_half :
  ∀ p k,
    k ≠ 0 → prime p → 2 < p →
    |ℤ_(p^k)| = 2 · Σ_(a∈ℤ_(p^k)) [∃x, x^2 ≡ a  mod p^k].
Lemma qnr_half :
  ∀ p k,
    k ≠ 0 → prime p → 2 < p →
    |ℤ_(p^k)| = 2 · Σ_(a∈ℤ_(p^k)) [∀x, x^2 ≢ a  mod p^k].
```

Here $[\exists x, x^2 \equiv a \mod p^k]$ equals to 1 if a is a quadratic residue modulo $\texttt{p}^\texttt{k}$, otherwise it equals to 0. Similarly, $[\forall x, x^2 \not\equiv a \mod \texttt{p}^\texttt{k}]$ represents whether a is a quadratic non-residue modulo $\texttt{p}^\texttt{k}$. These lemmas are proved by the fact that a quadratic residue a has exactly two solutions in $\mathbb{Z}_{p^k}$ to the equation $x^2 \equiv a \mod p^k$. Thus for the two-to-one self-map over $\mathbb{Z}_{p^k}$

$$x \mapsto x^2 \bmod p^k,$$

the size of its image is exactly half of the size of $\mathbb{Z}_{p^k}$. To prove this result in Coq, we generalize two-to-one functions with mask functions of type $\mathbb{N} \to \mathbb{B}$ to encode the available positions, then reason by induction.

C. End-to-end Certification. We present the final statement of the correctness of the end-to-end implementation of Shor's algorithm.

```
Theorem end_to_end_shor_fails_with_low_probability :
  ∀ N niter,
    ¬ (prime N) → Odd N →
    (∀ p k, prime p → N ≠ p^k) →
    ℙ_(rnds∈Uniform([0,1]^niter)) [end_to_end_shor N rnds = None]
      ≤ (1 - (1/2) * (β / (log2 N)^4))^niter.
```

Then r can be less than an arbitrarily small positive constant ϵ by enlarging `niter` to $\frac{2}{\beta} \ln \frac{1}{\epsilon} \log_2^4 N$, which is $O(\log^4 N)$.

This theorem can be proved by combining the success probability of finding the multiplicative order and the success probability of choosing proper a in the reduction from factorization to order finding. We build an ad-hoc framework for reasoning about discrete probability procedures to express the probability here.

D. Certifying Resource Bounds. We provide a concrete polynomial upper bound on resource consumption in our implementation of Shor's algorithm. The aspects of resource consumption considered here are the number of qubits and the number of primitive gates supported by OpenQASM 2.0 (16). The number of qubits is easily bounded by the maximal index used in the SQIR program, which is linear to the length of the input. For gate count bounds, we reason about the structure of our circuits. We first generate the gate count bound for the RCIR program, then we transfer this bound to the bound for the SQIR program. Eventually, the resource bound is given by

```
Lemma ugcount_shor_circuit :
  ∀ a N,
    0 < N →
    let m := Nat.log2 (2*(N^2)) in
    let n := Nat.log2 (2*N) in
    ugcount (shor_circuit a N) ≤
    (212*n*n + 975*n + 1031)*m + 4*m + m*m.
```

Here `ugcount` counts how many gates are in the circuit. Note $m, n = O(\log N)$. This gives the gate count bound for one iteration as $(212n^2 + 975n + 1031)m + 4m + m^2 = O(\log^3 N)$, which is asymptotically the same as the original paper (20), and similar to other implementations of Shor's algorithm (27, 28) (up to $O(\log \log N)$ multiplicative difference because of the different gate sets).

Using the certified bound, we may estimate the upper bound of gates in large factorization circuits in our implementation. To factorize a number of 1024-bit, our implementation will generate circuits with at most 4.58×10^{11} gates. This upper bound is several orders of magnitudes larger than the estimation in (27, 28) (orders of magnitudes around 10^9) because our implementation is not optimized to reduce the total gate count but for a structured certification procedure.

4. Running Certified Code

The codes are certified in Coq, which is a language designed for formal verification. To run the codes realistically and efficiently, extractions to other languages are necessary. Our certification contains the quantum part and the classical part. The quantum part is implemented in SQIR embedded in Coq, and we extract the quantum circuit into OpenQASM 2.0 (16) format. The classical part is extracted into OCaml code following Coq's extraction mechanism (32). Then the OpenQASM codes can be sent to a quantum computer (in our case, a classical simulation of a quantum computer), and OCaml codes are executed on a classical computer.

With a certification of Shor's algorithm implemented inside Coq, the guarantees of correctness on the extracted codes are strong. However, although our Coq implementation of Shor's algorithm is fully certified, extraction introduces some trusted code outside the scope of our proofs. In particular, we trust that extraction produces OCaml code consistent with our Coq definitions and that we do not introduce errors in our conversion from SQIR to OpenQASM. We "tested" our extraction process by generating order-finding circuits for various sizes and confirming that they produce the expected results in a simulator.

A. Extraction. For the quantum part, we extract the Coq program generating SQIR circuits into the OCaml program generating the corresponding OpenQASM 2.0 assembly file. We substitute the OpenQASM 2.0 gate set for the basic gate set in SQIR, which is extended with: $X, H, U_1, U_2, U_3, CU_1$, $SWAP, CSWAP, CX, CCX, C3X, C4X$. Here X, H are the Pauli X gate and Hadamard gate. U_1, U_2, U_3 are single-qubit rotation gates with different parametrization (16). CU_1 is the controlled version of the U_1 gate. $SWAP$ and $CSWAP$ are the swap gate and its controlled version. $CX, CCX, C3X$, and $C4X$ are the controlled versions of the X gate, with a different number of control qubits. Specifically, CX is the CNOT gate. The proofs are adapted with this gate set. The translation from SQIR to OpenQASM then is direct.

For the classical part, we follow Coq's extraction mechanism. We extract the integer types in Coq's proof to OCaml's `Z` type, and several number theory functions to their correspondence in OCaml with the same behavior but better efficiency. Since our proofs are for programs with classical probabilistic procedures and quantum procedures, we extract the sampling procedures with OCaml's built-in randomization library.

One potential gap in our extraction of Coq to OCaml is the assumption that OCaml floats satisfy the same properties as Coq Real numbers. It is actually not the case, but we did not observe any error introduced by this assumption in our testing. In our development, we use Coq's axiomatized representation of reals (17), which cannot be directly extracted to OCaml. We chose to extract it to the most similar native data type in OCaml–floating-point numbers. An alternative would be to prove Shor's algorithm correct with gate parameters represented using some Coq formalism for floating-point numbers (33), which we leave for future work.

B. Experiments. We test the extracted codes by running small examples on them. Since nowadays quantum computers are still not capable of running quantum circuits as large as generated Shor's factorization circuits (∼30 qubits, $\sim 10^4$ gates for small cases), we run the circuits with the DDSIM simulator (34) on a laptop with an Intel Core i7-8705G CPU. The experiment results are included in Figure 4 (b) (c).

As a simple illustration, we showcase the order finding for $a = 3$ and $N = 7$ on the left of Figure 4 (b). The extracted OpenQASM file makes use of 29 qubits and contains around 11000 gates. DDSIM simulator executes the file and generates simulated outcomes for 10^5 shots. The measurement results of QPE are interpreted in binary representation as estimated $2^m \cdot k/r$. In this case, the outcome ranges from 0 to 63, with different frequencies. We apply OCaml post-processing codes for order finding on each outcome to find the order. Those measurement outcomes reporting the correct order (which is 6) are

marked green in Figure 4 (b). The frequency summation of these measurement outcomes over the total is 28.40%, above the proven lower bound of the success probability of order finding which is 0.17% for this input.

We are also able to simulate the factorization algorithm for $N = 15$. For any a coprime to 15, the extracted OpenQASM codes contain around 35 qubits and 22000 gates. Fortunately, DDSIM still works efficiently on these cases due to the well-structured states of these cases, taking around 10 seconds for each simulation. We take 7×10^5 shots in total. When $N = 15$, the measurement outcomes from QPE in order finding are limited to $0, 64, 128, 192$ because the order of any a coprime to 15 is either 2 or 4, so $2^m \cdot k/r$ can be precisely expressed as one of them without approximation. The frequency of the simulation outcomes for $N = 15$ is displayed on the right of Figure 4 (b). We then apply the extracted OCaml post-processing codes for factorization to obtain a non-trivial factor of N. The overall empirical success probability is 43.77%, above our certified lower bound of 0.17%.

We have also tested larger cases on DDSIM simulator (34) for input size ranging from 2 bits to 10 bits (correspondingly, N from 3 to 1023), as in Figure 4 (c). Since the circuits generated are large, most of the circuits cannot be simulated in a reasonable amount of time (we set the termination threshold 1 hour). We exhibit selected cases that DDSIM is capable of simulating: $N = 15, 21, 51, 55, 63, 77, 105, 255$ for factorization, and $(a, N) = (2, 3), (3, 7), (7, 15), (4, 21), (18, 41), (39, 61), (99, 170), (101, 384), (97, 1020)$ for order finding. These empirically investigated cases are drawn as red circles in Figure 4 (c). Most larger circuits that are simulated by DDSIM have the multiplicative order a power of 2 so that the simulated state is efficiently expressible. For each input size, we also calculate the success probability for each possible input combination by using the analytical formulae of the success probability with concrete inputs. Shor shows the probability of obtaining a specific output for order finding is(20)

$$\mathbb{P}[\texttt{out} = u] = \frac{1}{2^{2m}} \sum_{0 \le k < r} \left| \sum_{\substack{0 \le v < r \\ v \equiv k \pmod r}} e^{2\pi i u v / 2^m} \right|^2 .$$

Here r is the order and m is the precision used in QPE. The success probability of order finding then is a summation of us for which the post-processing gives correct r. For most output u, the probability is negligible. The output tends to be around $2^m k/r$, so the sum is taken over integers whose distance to the closest $2^m k/r$ (for some k) is less than a threshold, and the overall probability of getting these integers is at least 95%. Hence the additive error is less than 0.05. These empirical results are drawn as blue intervals (i.e., minimal to maximal success probability) in Figure 4 for each input size, which is called the empirical range of success probability. The certified probability lower bounds are drawn as red curves in Figure 4 as well. The empirical bounds are significantly larger than the certified bounds for small input sizes because of loose scaling in proofs, and non-optimality in our certification of Euler's totient function's lower bounds. Nevertheless, asymptotically our certified lower bound is sufficient for showing that Shor's algorithm succeeds in polynomial time with large probability.

We also exhibit the empirical gate count and certified gate count for order finding and factorization circuits. The circuits for order finding are exactly the factorization circuits after a is picked, so we do not distinguish these two problems for gate count. On the right of Figure 4 (c), we exhibit these data for input sizes ranging from 2 to 10. We enumerate all the inputs for these cases and calculate the maximal, minimal, and average gate count and draw them as blue curves and intervals. The certified gate count only depends on the input size, which is drawn in red. One can see the empirical results satisfy the certified bounds on gate count. Due to some scaling factors in the analytical gate count analysis, the certified bounds are relatively loose. Asymptotically, our certified gate count is the same as the original paper's analysis.

References

1. HB Curry, Functionality in combinatory logic. *Proc. Natl. Acad. Sci. United States Am.* **20**, 584 (1934).
2. WA Howard, The formulæ-as-types notion of construction in *The Curry-Howard Isomorphism*, ed. PD Groote. (Academia), (1995).
3. NG De Bruijn, The mathematical language AUTOMATH, its usage, and some of its extensions in *Symposium on automatic demonstration*. (Springer), pp. 29–61 (1970).
4. R Milner, Implementation and applications of Scott's logic for computable functions. *ACM sigplan notices* **7**, 1–6 (1972).
5. T Coquand, G Huet, Constructions: A higher order proof system for mechanizing mathematics in *EUROCAL '85*, ed. B Buchberger. (Springer Berlin Heidelberg, Berlin, Heidelberg), pp. 151–184 (1985).
6. Inria, CNRS and contributors, Calculus of inductive constructions (2021) Accessed: 2021-12-10.
7. X Leroy, Formal verification of a realistic compiler. *Commun. ACM* **52**, 107–115 (2009).
8. K Bhargavan, et al., Everest: Towards a verified, drop-in replacement of HTTPS in *2nd Summit on Advances in Programming Languages*. (2017).
9. JK Zinzindohoué, K Bhargavan, J Protzenko, B Beurdouche, Hacl*: A verified modern cryptographic library in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17. (Association for Computing Machinery, New York, NY, USA), p. 1789–1806 (2017).
10. G Klein, et al., SeL4: Formal verification of an OS kernel in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09. (Association for Computing Machinery, New York, NY, USA), p. 207–220 (2009).
11. T mathlib Community, The lean mathematical library. *Proc. 9th ACM SIGPLAN Int. Conf. on Certif. Programs Proofs* (2020).

12. D Castelvecchi, Mathematicians welcome computer-assisted proof in 'grand unification' theory (https://www.nature.com/articles/d41586-021-01627-2) (2021).
13. K Hartnett, Proof assistant makes jump to big-league math (https://www.quantamagazine.org/lean-computer-program-confirms-peter-scholze-proof-20210728/) (2021).
14. K Hietala, R Rand, SH Hung, X Wu, M Hicks, A verified optimizer for quantum circuits. *Proc. ACM Program. Lang.* **5** (2021).
15. K Hietala, R Rand, SH Hung, L Li, M Hicks, Proving quantum programs correct in *Proceedings of the Conference on Interative Theorem Proving (ITP)*. (2021).
16. AW Cross, LS Bishop, JA Smolin, JM Gambetta, Open quantum assembly language. *arXiv preprint arXiv:1707.03429* (2017).
17. Library Coq.Reals.Reals (https://coq.inria.fr/library/Coq.Reals.Reals.html) (year?) Accessed: 2021-09-24.
18. S Boldo, C Lelay, G Melquiond, Coquelicot: A user-friendly library of real analysis for coq. *Math. Comput. Sci.* **9**, 41–62 (2015).
19. J Paykin, R Rand, S Zdancewic, QWIRE: A core language for quantum circuits. *SIGPLAN Not.* **52**, 846–858 (2017).
20. PW Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.* **26**, 1484–1509 (1997).
21. DE Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. (Addison Wesley Longman Publishing Co., Inc., USA), (1997).
22. SA Cuccaro, TG Draper, SA Kutin, DP Moulton, A new quantum ripple-carry addition circuit. *arXiv preprint quant-ph/0410184* (2004).
23. AL Ruiz, EC Morales, LP Roure, AG Ríos, *Algebraic circuits*. (Springer), (2014).
24. TG Draper, Addition on a quantum computer. *arXiv preprint quant-ph/0008033* (2000).
25. TG Draper, SA Kutin, EM Rains, KM Svore, A logarithmic-depth quantum carry-lookahead adder. *arXiv preprint quant-ph/0406142* (2004).
26. R Van Meter, KM Itoh, Fast quantum modular exponentiation. *Phys. Rev. A* **71**, 052320 (2005).
27. A Pavlidis, D Gizopoulos, Fast quantum modular exponentiation architecture for shor's factorization algorithm. *arXiv preprint arXiv:1207.0511* (2012).
28. C Gidney, M Ekerå, How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* **5**, 433 (2021).
29. D De Rauglaudre, Coq proof of the euler product formula for the riemann zeta function (https://github.com/roglo/coq_euler_prod_form) (2020).
30. JB Rosser, L Schoenfeld, Approximate formulas for some functions of prime numbers. *Ill. J. Math.* **6**, 64–94 (1962).
31. GH Hardy, EM Wright, *An Introduction to the Theory of Numbers*. (Oxford), Fourth edition, (1975).
32. Program extraction (https://coq.inria.fr/refman/addendum/extraction.html) (2021) Accessed: 2021-09-24.
33. S Boldo, G Melquiond, Flocq: A unified library for proving floating-point algorithms in coq in *2011 IEEE 20th Symposium on Computer Arithmetic*. pp. 243–252 (2011).
34. JKQ DDSIM – a quantum circuit simulator based on decision diagrams written in C++ (https://github.com/iic-jku/ddsim) (2021).

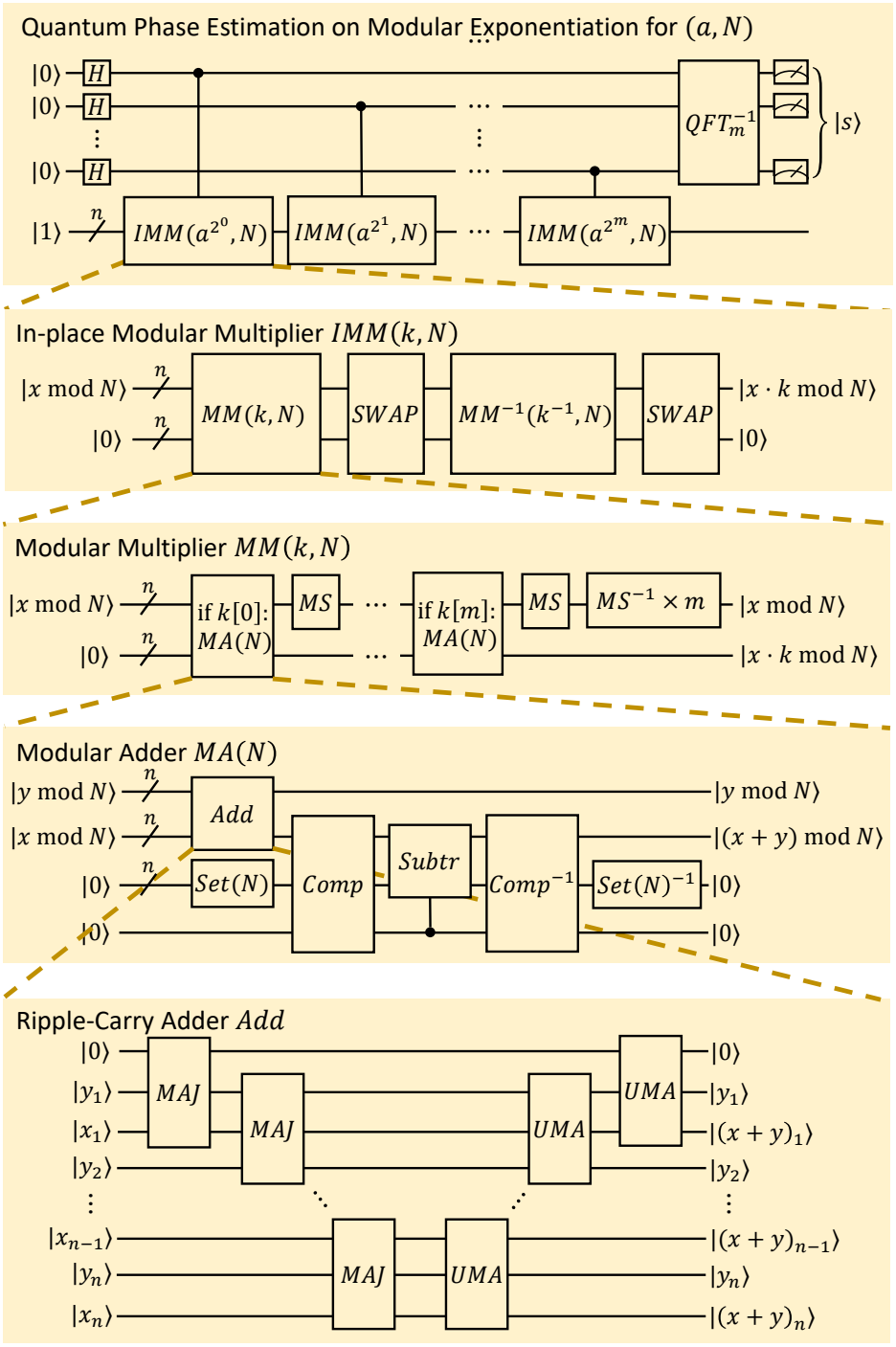

Fig. S1. An overview of our certified implementation of modular exponentiation circuit.

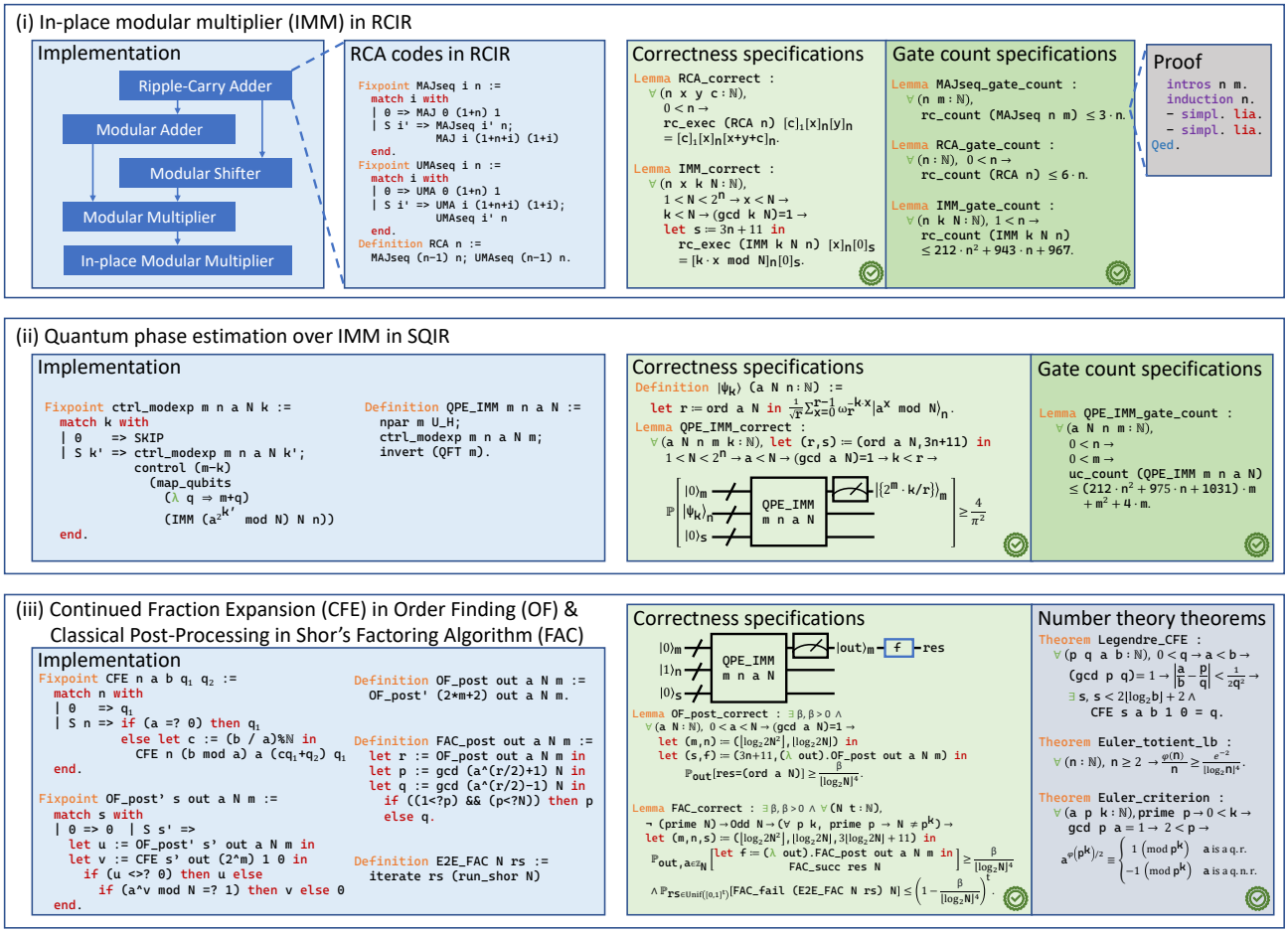

Fig. S2. Showcases of major components of our end-to-end implementation and corresponding proofs. Codes are adjusted for pretty-printing. (i) The implementation of IMM. We use the example of the Ripple-Carry Adder (`RCA`) to illustrate the specifications and proofs. (ii) The implementation of quantum phase estimation over IMM in SQIR (`QPE_IMM`). The correctness specification states that, under some premises, the probability of measuring the closest integer to $2^m k/r$, where r is the order of a modulo N, is larger than a positive constant $4/\pi^2$. We also certify the gate count of the implementation of `QPE_IMM`. (iii) The implementation of classical post-processing for order finding and factorization. Continued fraction expansion `CFE` is applied to the outcome of `QPE_IMM` to recover the order with a certified success probability at least 1/polylog(N). The success probability of factorization is also certified to be at least 1/polylog(N), which can be boosted to 1 minus some exponentially decaying error term after repetitions. These analyses critically rely on number theoretical statements like Legendre's theorem, lower bounds for Euler's totient function, and Euler's criterion for quadratic residues, which have been proven constructively in Coq in our implementation.

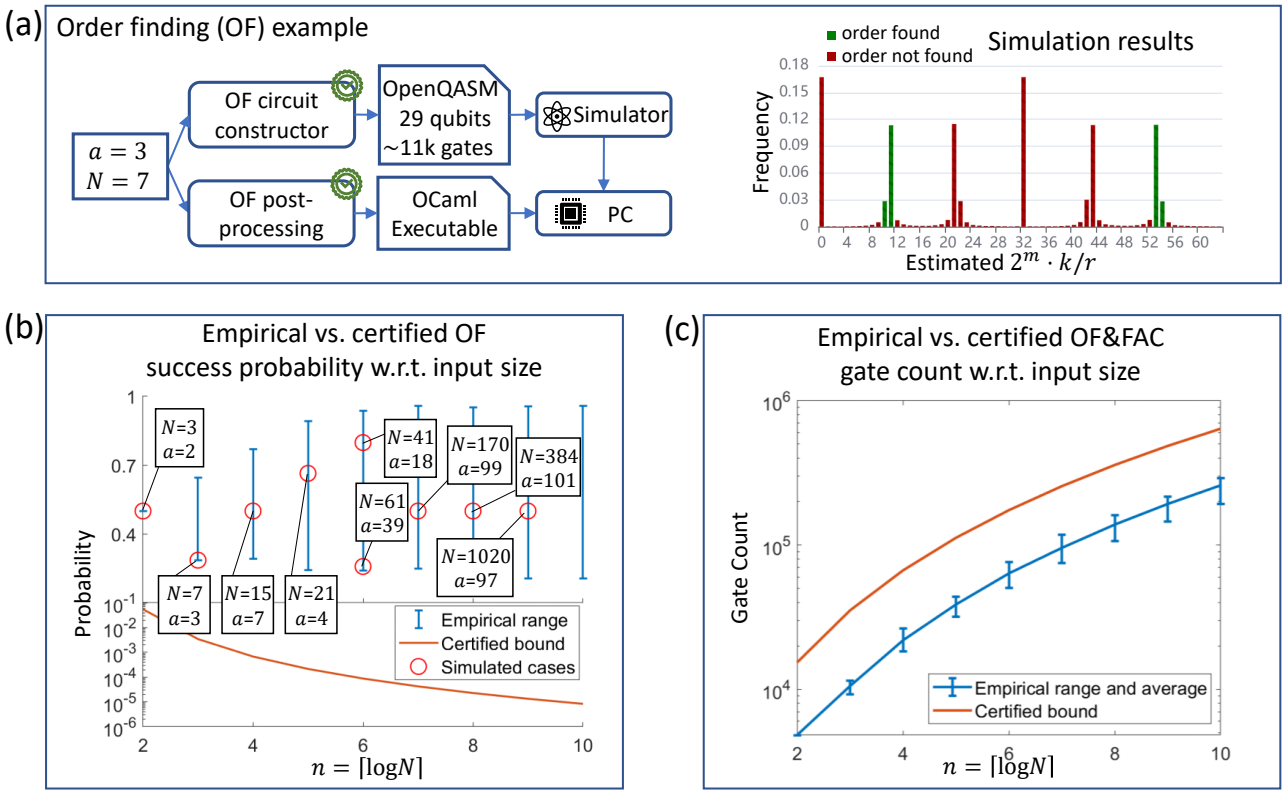

Fig. S3. End-to-end execution of the order-finding algorithm. (a) Examples of end-to-end executions of order finding (OF). The left example finds the order for a=3 and N=7. The generated OpenQASM file uses 29 qubits and contains around 11k gates. We employed JKQ DDSIM (34) to simulate the circuit for 100k shots, and the frequency distribution is presented. The trials with post-processing leading to the correct order r=6 are marked green. The empirical success probability is 28.40%, whereas the proved success probability lower bound is 0.34%. (b, c) Empirical statistics of the gate count and success probability of order finding for every valid input N with respect to input size n from 2 to 10 bits. We draw the bounds certified in Coq as red curves. Whenever the simulation is possible with DDSIM, we draw the empirical bounds as red circles. Otherwise, we compute the corresponding bounds using analytical formulas with concrete inputs. These bounds are drawn as blue intervals called empirical ranges (i.e., minimal to maximal success probability) for each input size.