MorphQPV: Exploiting Isomorphism in Quantum Programs to Facilitate Confident Verification

Siwei Tan*
Zhejiang University
Hangzhou, China
siweitan@zju.edu.cn

Debin Xiang*
Zhejiang University
Hangzhou, China
db.xiang@zju.edu.cn

Liqiang Lu†
Zhejiang University
Hangzhou, China
liqianglu@zju.edu.cn

Junlin Lu
Peking University
Beijing, China
ljl@pku.edu.cn

Qiuping Jiang
Ningbo University
Ningbo, China
jiangqiuping@nbu.edu.cn

Mingshuai Chen
Zhejiang University
Hangzhou, China
m.chen@zju.edu.cn

Jianwei Yin†
Zhejiang University
Hangzhou, China
zjuyjw@zju.edu.cn

Abstract

Unlike classical computing, quantum program verification (QPV) is much more challenging due to the non-duplicability of quantum states that collapse after measurement. Prior approaches rely on deductive verification that shows poor scalability. Or they require exhaustive assertions that cannot ensure the program is correct for all inputs. In this paper, we propose MorphQPV, a confident assertion-based verification methodology. Our key insight is to leverage the isomorphism in quantum programs, which implies a structure-preserve relation between the program runtime states. In the assertion statement, we define a tracepoint pragma to label the verified quantum state and an assume-guarantee primitive to specify the expected relation between states. Then, we characterize the ground-truth relation between states using an isomorphism-based approximation, which can effectively obtain the program states under various inputs while avoiding repeated executions. Finally, the verification is formulated as a constraint optimization problem with a confidence estimation model to enable rigorous analysis. Experiments suggest that MorphQPV reduces the number of program executions by 107.9× when verifying the 27-qubit quantum lock algorithm and improves the probability of success by 3.3×-9.9× when debugging five benchmarks.

CCS Concepts: • **Computer systems organization** → **Quantum computing**; • **Theory of computation** → **Program verification**.

*These authors contribute equally to this work.
†Corresponding author

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651360

Keywords: quantum computing, program verification

ACM Reference Format:
Siwei Tan, Debin Xiang, Liqiang Lu, Junlin Lu, Qiuping Jiang, Mingshuai Chen, and Jianwei Yin. 2024. MorphQPV: Exploiting Isomorphism in Quantum Programs to Facilitate Confident Verification. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620666.3651360

1 Introduction

Quantum computing is a promising technique that can achieve low power consumption and low computational complexity [12, 26, 32]. A quantum program consists of a sequence of quantum instructions that speed up computation using quantum superposition and entanglement. Similar to classical programs, quantum programs inevitably incur many computational defects, such as bugs. Currently, there is strong motivation to develop advanced verification tools to debug quantum programs. For example, only 21% of questions related to quantum programs are resolved on Stack Overflow [38], which is significantly lower (4.1 times less) than that of classical programs. Nevertheless, the verification of quantum programs faces obstacles due to unique properties introduced by the quantum mechanism. Specifically, the superposition state greatly expands the search space for bugs, while the non-duplicability requires massive program executions to probe the program states.

Quantum program verification (QPV) is mainly performed by deductive verification [52, 54, 55, 57] and runtime assertion [13, 20, 27–29]. Deductive verification relies on precise formulations of the program behaviors, e.g., quantum Hoare logic [57], where the correctness can be compiled into a set of verification conditions. However, reasoning about the correctness relies on (1) discharging the verification conditions via classical computers, incurring significant computational costs, and (2) human expertise to identify inductive invariants, thus restricting automation levels. On the other hand,

runtime assertion is a lightweight method that tests the program under with varying inputs. An assertion is defined as a predicate toward the properties of program states (e.g., purity [55] and expectation [27, 28]), which is expected to be true if there are no bugs. Validating the assertion entails three steps: (a) statement of the assertion; (b) characterization of the program; and (c) validation to check whether the characterization result satisfies the assertion predicates. The application of assertions has been primarily studied on a case-by-case basis [20]. Researchers like Liu et al. [28] and Li et al. [27] have expanded on this work to enable dynamic assertions on quantum hardware.

Confidence is the key metric to evaluate QPV, which is defined as the probability that the verification result holds true for all inputs. Existing assertion works [13, 27–29, 55] exhibit low confidence in verifying the overall correctness of the program. This limitation arises from the fact that they can only validate the assertion under a small subset of test inputs, without the capability to generalize the validation result to the entire input space. To improve confidence, some works [1, 46, 47] exhaustively test numerous inputs, which is insufficient when considering the continuous Hilbert space. For instance, Li et al. [27] and Liu et al. [29] only test a single input in each verification, while bugs may not be activated under this input. Quito [47] applies a grid search on the input space, necessitating 4.8×10^6 program executions to verify an 11-qubit QRAM program [14].

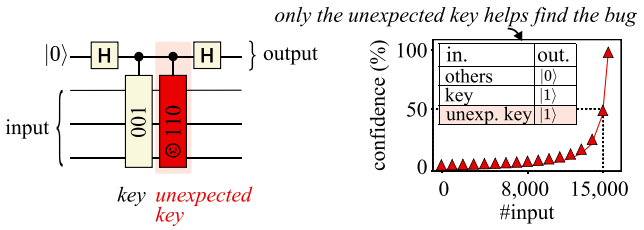
Fundamentally, the implementations of current assertion methods cannot support input-independent verification due to the following limitations:

a) Assertion statement. Current predicates of the assertion are limited to a single program state, rendering them unable to check the relations between states at different time points. For example, the assertions proposed by Li et al. [27] and Liu et al. [29] only check whether qubits are in a specified state set. For different inputs, this state set has to be re-declared.

b) Characterization. Current techniques cannot characterize the relations between states, leading to repetitive testing of programs for each input [27]. Besides, they can only probe limited features of the state due to the quantum collapse after the measurement. For example, Twist [55] only supports to validate the purity of the state, and Huang et al. [20] only probes the amplitudes of the states.

c) Assertion validation. Current methods fail to consolidate findings from different testing inputs to support comprehensive verification. Strategies employed by Quito [47] and Fuzz [46] apply an exhaustive search until going through the entire space or encountering a bug. Furthermore, there is an absence of an analytical framework to accurately predict verification confidence.

Figure 1 (a) showcases a quantum lock program to illustrate the above-mentioned limitations. Quantum lock is an important module in encrypting quantum program outputs by encoding a binary key, such as 001 shown in Figure 1 (a). The program is expected to output $|1\rangle$ only when the input matches the key, e.g., $001 \rightarrow |1\rangle$, others $\rightarrow |0\rangle$. A program bug occurs when an unexpected key, such as 110, is present in the program, making the program also output $|1\rangle$ for inputting this key ($110 \rightarrow |1\rangle$). This bug cannot be identified until this unexpected key is tested. In the assertion statement, recent studies [27, 29, 47] require distinct assertions (output= $|1\rangle$ and output= $|0\rangle$) for various inputs (key and others). These methods exhaustively test inputs to detect the unexpected key during the characterization phase. In the validation, they cannot generalize the execution results from some inputs to other inputs. For example, they cannot use the results of inputs 010 and 101 to validate input 110. Figure 1 (b) presents the confidence of the overall correctness from the method proposed by Liu et al. [29]. For a 15-qubit quantum lock program, the confidence level is a mere 0.006% after a single test and only reaches 50% confidence after testing 1.5×10^4 inputs.



(a) 4-qubit quantum lock with a bug caused by the unexpected key.

(b) Confidence of the verification result in the 15-qubit quantum lock.

Figure 1. Motivational example.

In this paper, we propose MorphQPV, a confident verification methodology that achieves confident verification with a minimal number of program executions. MorphQPV is advanced in its ability that only takes twice the effort of classical program verification, which is a significant advantage when considering the exponentially increased space of quantum states. Our key insight is to leverage the *isomorphism* of the program, which implies the structure-preserving relationship between the input and the runtime states of the program. This isomorphism facilitates a novel characterization technique, employing the program's behavior under specific inputs to infer behaviors under alternative inputs, which helps to extend the verification result to the entire input space.

MorphQPV introduces a verification flow consisting of three steps: First, MorphQPV defines a form of multi-state assertion to specify the expected relations between runtime states of the program, inspired by a classical assume-guarantee primitive. The multi-state assertion is a static statement that features an input-independent description of program behavior. Besides, the assertion is described as a classical function involving the density matrix of the state to

ensure maximum expressiveness. Second, MorphQPV characterizes the relation between states as classical functions using an isomorphism-based approximation. These functions can approximate the density matrix of the program states based on the input, without executing the program. The accuracy of the approximation is guaranteed by mathematical proof. Third, MorphQPV models the assertion and the characterized relations into a constraint optimization problem, which validates the assertion without specifying the test inputs. Furthermore, in addition to estimating the confidence when the program is correct, our approach can provide counter-examples when finding bugs.

The major contributions of the paper are summarized as follows:

- We propose a multi-state assertion that characterizes the relationships among a sequence of quantum states at different time points. This enhances both the efficiency and expressiveness of program verification.
- We propose an approximation technique to calculate the runtime states rather than repeatedly executing the program. The approximation effectively captures the program behavior at a relatively low computational cost.
- We propose an input-independent validation method for the assertion, capable of generating counter-examples and providing an estimation of the validation confidence.

The evaluation utilizes case studies and quantitative analysis to suggest that our work is more expressive in identifying errors. It also achieves an up to 107.9× reduction of program executions and 3.3×-9.9× improvement of the success probability when debugging five algorithms. The source code and dataset of MorphQPV are publicly available on (https://github.com/JanusQ/MorphQPV).

2 Background

2.1 Quantum Program

In a quantum program, information is encoded in the state of qubits. For a n-qubit system, its quantum state is described by a density matrix ρ, which can be categorized into the pure state and the mixed state. Generally, the density matrix of a mixed state is a linear combination of the density matrices of pure states, represented as $\rho = \sum_i p_i |\psi_i\rangle \langle\psi_i|$. Here, p_i represents the probability, and $|\psi_i\rangle \langle\psi_i|$ denotes the density matrix of the pure state. All properties of the quantum state can be analyzed using its density matrix. For example, when the state ρ is pure, $|\rho\rho^\dagger - \rho|$ equals 0. In this paper, the quantum states are also represented as the density matrices.

Inherently, quantum computing is an evolution of the quantum state, which is mathematically expressed as a unitary matrix (unitary) U. For a quantum state represented as a density matrix ρ, it evolves to a new density matrix ρ' after applying one unitary:

$$\rho' = U\rho U^\dagger. \tag{1}$$

Table 1. Notations used in the following sections.

Notation	Meaning
$\dagger$	Conjugate transpose that transposes a matrix and applies complex conjugation to its elements.
tr	Trace operator that sums the diagonal elements of a matrix.
$\|\| \cdot \|\|$	L2 norm that calculates the square root of the sum of the square of matrix elements.
ρ_{T_i}	Density matrix of qubit state at tracepoint T_i.
$\rho_{\mathrm{T}_i} = f_i(\rho_{\mathrm{in}})$	Classical function that approximates the relation between input ρ_{in} and tracepoint state ρ_{T_i}.
$\langle\sigma_{\mathrm{in},i}, \sigma_{\mathrm{T},i}\rangle$	i^{th} sampled input $\sigma_{\mathrm{in},i}$ and corresponding state $\sigma_{\mathrm{T},i}$ at tracepoint T.
N_{sample}	Number of sampled inputs.
N_{in}	Number of qubits of the input.
N_{T_i}	Number of qubits at tracepoint T_i.
shots	Number of times to repeatedly execute one quantum program.

Here, $\dagger$ is the conjugate transpose operation that transposes the matrix and applies complex conjugation to its elements.

Quantum measurement is the only operation that reads information from qubits to classical computers. The projective measurement is performed on a collection of operators $\{O\}$ satisfying $\sum_{O\in\{O\}} O^\dagger O = I$, where I is the identity matrix. The expectation of measuring quantum state ρ on operator O is

$$\mathbb{E}_O[\rho] = tr(O\rho), \tag{2}$$

where tr is the trace operation. The measurement process evolves state ρ into a new state ρ':

$$\rho' = \frac{O\rho O^\dagger}{\mathbb{E}_O[\rho]}. \tag{3}$$

In this work, we assume that a program execution consists of one input and multiple shots. The notations and terminologies used in this paper are listed in Table 1.

2.2 Isomorphism

Mathematically, an *isomorphism* is defined as a structure-preserving mapping between two spaces of the same type that can be retraced by an inverse mapping [25]. For example, the mapping $\mathbb{R}_x \rightarrow \mathbb{R}_y$ that satisfies $x + 1 = y$ is isomorphic since we can formulate its inverse mapping $x = y - 1$. Extended from isomorphism, a *linear isomorphism* is defined as a mapping specified by reversible matrices. **A quantum evolution is isomorphic, as the quantum operator is reversible ($U^{-1} = U^\dagger$ or $O^{-1} = O^\dagger$).** A linear isomorphism f satisfies the additivity and homogeneity:

$$\text{additivity} : f(u + v) = f(u) + f(v),$$
$$\text{homogeneity} : f(c\ u) = c\ f(u),$$

where c is a constant. Elements u and v are in the input space of isomorphism f. Based on the additivity and homogeneity,

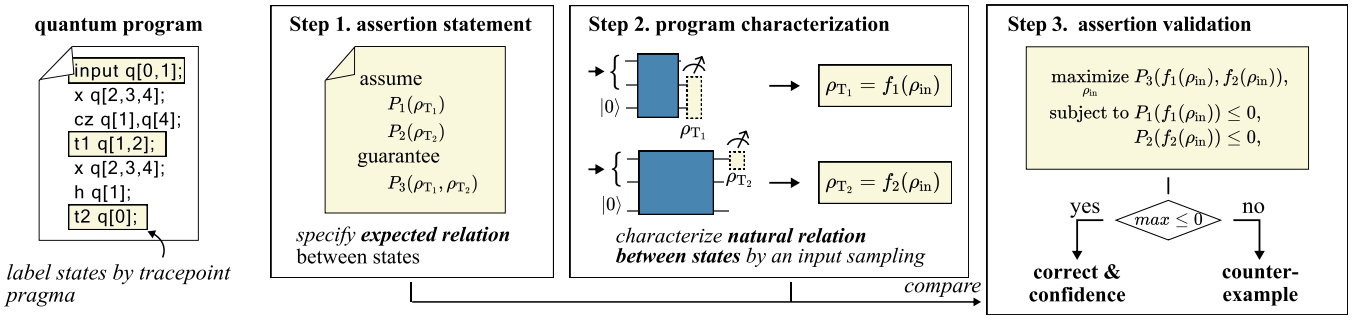


Figure 2. Overview of MorphQPV.

the equality holds in:

$$f(\sum_i c_i\, u_i) = \sum_i c_i\, f(u_i). \tag{4}$$

3 MorphQPV Overview

Figure 2 presents the verification workflow of MorphQPV to verify quantum program, consisting of three steps:

Step 1. assertion statement. We label the states in the program via tracepoint pragma. Subsequently, each tracepoint records the time and the associated qubits, which is used to describe the expected program behavior with these states. We define an assume-guarantee assertion with predicates to specify: (a) the ranges of these states, represented as objective functions for each state, e.g., $P_1(\rho_{T_1})$ and $P_2(\rho_{T_2})$ in Figure 2; and (b) the relation between these states, represented as a objective function involving multiple states, e.g., $P_3(\rho_{T_1}, \rho_{T_2})$. The predicate is validated on the classical computers.

Step 2. program characterization. MorphQPV characterizes the natural relations between quantum states by running the program on the quantum hardware. The characterization begins with a one-shot input sampling to record the labeled states across different inputs. By exploiting isomorphism, it then builds approximation functions based on the sampling results, e.g., $\rho_{T_1} = f_1(\rho_{in})$, representing the relations between the input and the labeled states. These approximation functions can be efficiently computed to obtain tracepoint states on classical computers.

Step 3. assertion validation. MorphQPV validates the assertion by checking whether the relations in the program satisfy the expected constraints in the assertion. Instead of testing tremendous inputs to identify the error, we apply a global search that packs the predicates and the approximation functions into a constraint maximization problem. The assume-guarantee assertion is true only if the maximum objective is less than 0. When the program is incorrect, the maximum argument ρ_{in} is the counter-example resulting in the bug. When the program is correct, MorphQPV estimates the confidence based on the accuracy of the characterization.

4 Assertion Statement

We introduce assertions to specify the relations between states, aiming to provide a comprehensive description of program behavior across different inputs. We label the qubit states at different times of the program by injecting tracepoints. A tracepoint T_i is defined as:

$$T_i \equiv (\{Q_i\}, time_i),\ Q_i \in Q, \tag{5}$$

where Q is the qubit set of the quantum program. $time_i$ is the time to obtain the density matrix ρ_{T_i} of qubits Q_i. The tracepoint is declared as a pragma in the QASM language [11], defined as "T index q[qubits]". For example,

```
1 h q[1];
2 cx q[1], q[2];
3 T 1 q[1,2]; // add tracepoint T1 on qubits 1,2.
4 cx q[2], q[3];
```

where tracepoint T_1 is injected into the GHZ [15] circuit at $time_1 = 3$.

Assertions are written in classical predicates to express the ranges of states at tracepoints and the relations between them.

Definition 1 (Assume-guarantee assertion)**.** An assume-guarantee assertion is defined as:

$$\begin{aligned} assert(T_i, T_j) \equiv \text{assume: } & P_1(\rho_{T_i}),\ P_2(\rho_{T_j}), \\ \text{guarantee: } & P_3(\rho_{T_i}, \rho_{T_j}), \end{aligned} \tag{6}$$

where predicate P_k is an inequality or objective function that takes density matrices as inputs. The objective function satisfies $P_k \le 0$ when and only when the predicate is true.

The assertion is set to ensure that the program always meets the following condition: when states ρ_{T_i} and ρ_{T_j} satisfy the predicates P_1 and P_2, they should satisfy the predicate P_3. Put simply, the assertion fails if an input satisfies P_1 and P_2 but violates P_3. The rule of predicates P_1 and P_2 is determined by the input format of the specific optimizer in Section 6.1. For example, when using quadratic programming in verification, the predicate is described as the inequality within two degrees. The theoretical model of MorphQPV supports taking arbitrary mathematical formulations as predicates

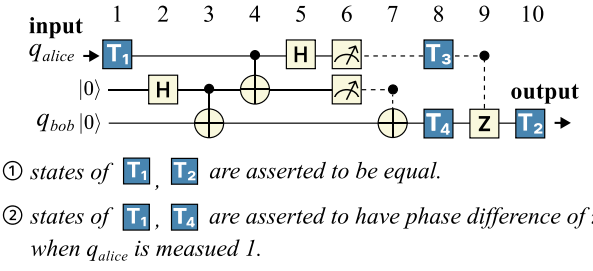


Figure 3. Example of assertions in quantum teleportation.

since the density matrix of the asserted state is obtained from the classical computer.

The positions of tracepoints are determined by the verification goals. There are two preferred strategies for placing the assertion. Firstly, we have some prior knowledge of the state at the position where we put the tracepoint. In this case, we usually place tracepoints at the start or end of sub-routines. Secondly, we place the statement in the area that can help understand the program functionality or areas suspected of having problems.

The assertion definition of MorphQPV gets inspiration from the classical assume-guarantee assertion, which is used to identify deadlock and concurrent exceptions in classical parallelized programs [44]. Considering that a quantum program is naturally parallelized, our assertion can detailly describe the amplitude transition of bases in the program. Besides, compared to the classical assertion [44], the assumption in MorphQPV is the condition of the guarantee, filtering the input space that required validation. This provides two advantages for debugging quantum programs. First, this enables the debugging of programs with mid-measurement by assuming that the asserted state equals the collapsed state after the measurement. Second, it helps prune the input space and minimize the overhead. We can use multiple assume-guarantee assertions to elaborate the functionality of the program.

Figure 3 presents an example of quantum teleportation [5] to illustrate our assertion statement. The teleportation is expected to transport the input state of q_{alice} to the output state of q_{bob}. We inject tracepoints T_1 and T_2 to label the input state and the output state. Using the notation in Equation 5, $\text{T}_1 = (\{q_{\text{alice}}\}, 1)$, $\text{T}_2 = (\{q_{\text{bob}}\}, 10)$. The expected program behavior is that when input and output are pure states, the input state equals the output state

$$\text{assume: } \underbrace{P_1(\rho_{\text{T}_1}) = \left\|\rho_{\text{T}_1}\rho_{\text{T}_1}^{\dagger} - \rho_{\text{T}_1}\right\|}_{\text{when } \rho_{\text{T}_1} \text{ is pure,}}, \underbrace{P_2(\rho_{\text{T}_2}) = \left\|\rho_{\text{T}_2}\rho_{\text{T}_2}^{\dagger} - \rho_{\text{T}_2}\right\|}_{\rho_{\text{T}_2} \text{ is pure,}},$$
$$\text{guarantee: } \underbrace{P_3(\rho_{\text{T}_1}, \rho_{\text{T}_2}) = \left\|\rho_{\text{T}_1} - \rho_{\text{T}_2}\right\|}_{\rho_{\text{T}_1} \text{ should equal } \rho_{\text{T}_2}}, \tag{7}$$

where $||\cdot||$ is the L2 norm. We can follow the same definition to check the relations of states involving feedback. For example, we validate whether, at timepoint 8, the state of q_{bob} has a phase difference of π with the input state when q_{alice} is measured 1. Clearly, we add tracepoint $\text{T}_3 = (\{q_{\text{alice}}\}, 8)$ and $\text{T}_4 = (\{q_{\text{bob}}\}, 8\}$ to the program, and define the assertion:

$$\text{assume: } \underbrace{P_1(\rho_{\text{T}_3}) = \left\|\rho_{\text{T}_3} - |1\rangle\langle 1|\right\|}_{\text{when } q_{\text{alice}} \text{ is measured 1,}},$$
$$\text{guarantee: } \underbrace{P_3(\rho_{\text{T}_3}, \rho_{\text{T}_4}) = \left\|tr(\rho_{\text{T}_4}^{\dagger}\rho_{\text{in}}).phase - \pi\right\|}_{\rho_{\text{T}_4} \text{ should have a phase difference of } \pi \text{ with the input state}}.$$

5 Isomorphism-based Characterization

The characterization for each tracepoint T_i aims to capture the natural relation between the program input ρ_{in} and the state ρ_{T_i} at this tracepoint during execution, which is formulated as an approximation function, i.e., $\rho_{\text{T}_i} = f(\rho_{\text{in}})$.

5.1 Input Sampling

The characterization starts with an input sampling to initialize the approximation function, where we run the program under various inputs and apply tomography to the tracepoint states. These inputs are carefully designed to ensure high characterization accuracy. Specifically, the input states should be orthogonal and cover more eigenstates to maximize their variety. Besides, the inputs are expected to be easily prepared. Based on this objective, we utilize the circuits from the orthogonal Clifford group [6] to prepare the inputs. Bravyi et al. [6] utilizes Hadamard-free circuits with up-bound depths linear to the number of qubits. Compared to using basis states, the Clifford group is more expressive in representing superposition and entanglement, making it closer to commonly used program inputs.

The density matrix of the tracepoint state under different sampled inputs is obtained by tomography. The number of sampled inputs N_{sample} is pre-determined based on the expected accuracy of approximation (discussed in Section 5.3). The i^{th} input and tracepoint states are recorded as $\sigma_{\text{in},i}$ and $\sigma_{T,i}$, respectively. These two states are collected as $\langle\sigma_{\text{in},i}, \sigma_{T,i}\rangle$ pairs of size N_{sample}.

5.2 Isomorphism-based Approximation

As discussed in Section 2.2, the quantum evolution in the quantum program is isomorphic, which, therefore, has additivity and homogeneity. We adopt this property to build the approximation functions.

Specifically, the relation between ρ_{in} and ρ_T in the program has the following property. For input represented as the linear combination of sampled inputs

$$\rho_{\text{in}} = \sum_i \alpha_i \sigma_{\text{in},i}\ . \tag{8}$$

where parameter α_i is a real value, tracepoint state ρ_{T} under this input is

$$\rho_{\text{T}} = \sum_i \alpha_i \sigma_{\text{T},i}\ . \tag{9}$$

This is because for the natural relation represented as function $\rho_T = F(\rho_{in})$, there is

$$\rho_T = F(\rho_{in}) = F(\sum_i \alpha_i \sigma_{in,i}) = \sum_i \alpha_i F(\sigma_{in,i}) = \sum_i \alpha_i \sigma_{T,i}\ .$$

which comes from additivity and homogeneity in Equation 4. $F(\sigma_{in,i}) = \sigma_{T,i}$ is correct as these two states are collected from real program executions.

Based on this relation, all possible inputs of a program can be categorized into two parts. The first part consists of the inputs that can be decomposed into Equation 8. The tracepoint states under these inputs are accurately computed by Equation 9. The second part consists of the inputs that cannot be decomposed. However, we can approximate them by Equation 8. The tracepoint state under these inputs can be approximated by Equation 9. Overall, we can build the following function to obtain the approximated tracepoint states for any input.

Theorem 1 (Approximation function)**.** Function $\rho_T = f(\rho_{in})$ is an under-approximation of the real relation F between input ρ_{in} and tracepoint state ρ_T. The function is computed in two steps:

1. For input ρ_{in}, it first approximates the ρ_{in} by Equation 8 to obtain parameters $\{\alpha_i\}$.
2. It then outputs tracepoint state ρ_T by Equation 9 with parameters α_i.

Note that parameters α_i mathematically equals the expectation of input ρ_{in} on sampled input $\sigma_{in,i}$.

The approximation can be applied to programs with non-overlapping qubits in the input and tracepoint. The approximation also holds for programs with mid-measurements and simple feedback, where the relations of qubit states are not isomorphic. We put the detailed proof in Appendix A.

Figure 4 shows a visual example of a single-qubit program. In the input sampling, we execute three orthogonal input states $\sigma_{in,1} = |+\rangle\langle+|$ (on x-axis of the Bloch sphere), $\sigma_{in,2} = |+i\rangle\langle+i|$ (on y-axis), and $\sigma_{in,3} = |1\rangle\langle 1|$ (on z-axis). The tracepoint states are recorded as $\sigma_{T,1}$, $\sigma_{T,2}$, and $\sigma_{T,3}$. Given input state ρ_{in}, the first step approximates it according to Equation 8:

$$\rho_{in} = \alpha_1 |+\rangle\langle+| + \alpha_2 |+i\rangle\langle+i| + \alpha_3 |1\rangle\langle 1|,$$
$$\alpha_1 = \mathbb{E}_{|+\rangle}[\rho_{in}],\ \alpha_2 = \mathbb{E}_{|+i\rangle}[\rho_{in}],\ \alpha_3 = \mathbb{E}_{|1\rangle}[\rho_{in}].$$

In the second step, the tracepoint state under this input is computed according to Equation 9:

$$\rho_T = \alpha_1 \sigma_{T,1} + \alpha_2 \sigma_{T,2} + \alpha_3 \sigma_{T,3}.$$

Note that in this example, the approximation of the input state and tracepoint state have 100% accuracy, as input can be elaborately represented by Equation 8.

Building the approximation function can generalize the information obtained from individual input into a broader input space. Compared to simulation on classical computer and quantum program execution, it significantly reduces the complexity of the following verification. Specifically, the computation of the approximation function has linear complexity to test an input in the verification, as it only involves simple summing. In contrast, the complexity of the simulation is exponential to the overall number of qubits of the program. Obtaining the runtime state by executing the quantum program also has exponential complexity, as it requires state tomography to overcome the limitation of quantum collapse.

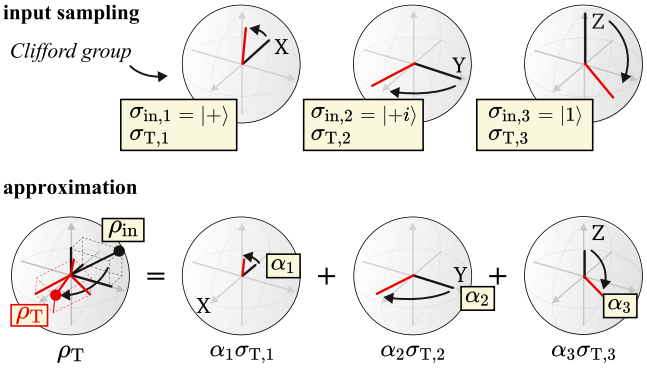


Figure 4. Visual example of the approximation in a single-qubit program.

5.3 Approximation Accuracy

The inaccuracy of the approximation occurs when the input cannot be decomposed into the linear combination of the sampled inputs. More samplings ensure less uncovered eigenstates in the sampled inputs and higher accuracy. We present a quantitative analysis of this trade-off in this section. Specifically, the accuracy of the approximation is determined by the number of qubits of the state (N_{in}) and the number of sampled inputs (N_{sample}) in the input sampling.

Theorem 2 (Approximation accuracy)**.** For different inputs, there are two cases:

1. For inputs that can be accurately represented by Equation 8, the accuracy is 100%.
2. For inputs with eigenstates that cannot be represented by Equation 8, the average accuracy is $N_{sample}/\ 2^{N_{in}+1} \times 100\%$.

When the number of samples N_{sample} equals $2^{N_{in}+1}$, the approximation accuracy is 100% for any input. In other words, the approximated tracepoint state is always the same as the real state obtained by state tomograph.

Here, we define accuracy as the Hilbert–Schmidt inner product between approximated ρ_T and real ρ_T obtained by executing the quantum program. The proof of the theorem is in the Appendix.

Theorem 2 suggests that the inaccuracy comes from the space that the sampled inputs cannot represent (Equation 8). By increasing the number of sampled inputs N_{sample}, we can

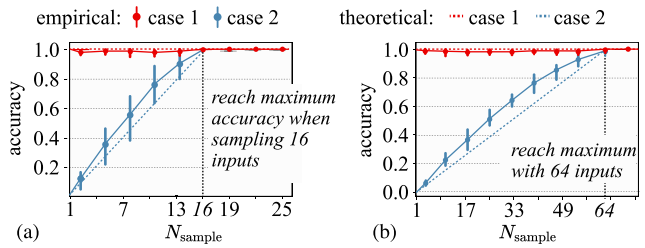


Figure 5. Experimental and theoretical accuracy of the approximation in the (a) 7-qubit and (b) 15-qubit quantum teleportation programs.

exponentially increase the space of case 1 and linearly increase the accuracy of case 2. Verification in case 1 is exactly accurate. For case 2, when bugs occur, the incorrect state usually shows a significant difference compared to expected state and can be identified with a certain accuracy threshold.

Figure 5 compares the experimental accuracy with theoretical value in 7-qubit and 15-qubit quantum teleportation programs, where $N_{\text{in}} = 3$ and 5, respectively. For case 1, the experimental accuracy is close to 1. The inaccuracy mainly results from the inaccuracy of the tomography. For case 2, the experimental accuracy grows linearly and reaches the maximum when the number of the samplings N_{sample} are 16 (2^{3+1}) and 64 (2^{5+1}), respectively. Besides, we observe that the experimental accuracy is usually larger than the theoretical value, as the Clifford group used in the input sampling is expressive when approximating the state evolution.

5.4 Pruning the Sample Space

The accuracy of the approximation linearly increases as the number of sampled inputs grows. We further propose three strategies to prune the sample space during the characterization process.

Strategy-adapt: adaptively determining the inputs for sampling. The input of a program is usually a sub-space of the whole quantum state, which can be represented by fewer orthogonal states. MorphQPV can apply eigendecomposition to the input space. The sampling only uses the eigenvectors with large eigenvalues as the program inputs, which reduces the number of sampled inputs. For example, when verifying a 5-qubit quantum neural network program that classifies handwritten numerals, MorphQPV can apply eigendecomposition to the training dataset and uses the top-9 eigenvectors with large eigenvalues as sampled inputs.

Strategy-const: setting a part of the input state as constant. The input of a program may consist of multiple parts of qubits that represent different data. We can make the state of some input qubits constant to reduce the size of input space. For example, the quantum adder program has two parts of inputs $|x\rangle$ and $|y\rangle$, represented as $f(|x\rangle, |y\rangle) = |x + y\rangle$ (e.g., $f(|01\rangle, |10\rangle) = |11\rangle$). By keeping $|x\rangle$ constant, we prune the sample space and focus on verifying the program under different states of $|y\rangle$.

Strategy-prop: checking a single property rather than the entire density matrix. Usually, we only check limited properties of the state (e.g., probability distribution, expectation, and purity) that also satisfy the additivity and homogeneity. MorphQPV can reduce the complexity of the tomography by only measuring the specific properties validated in the assertion.

6 Assertion Validation

6.1 Constraint Optimization

For assertion $assert(T_1, T_2)$, we adopt an optimization-based method to validate the assertion, which checks whether the characterized functions (e.g., $\rho_{T_1} = f_1(\rho_{\text{in}})$) satisfy the predicates of the assertion. If not, the validation outputs the counter-example that makes the assertion fail.

According to the definition of the assume-guarantee assertion (cf. Definition 1), predicate P_k is defined as an inequality or objective function. The inequality can also be transferred into the objective function. The objective function satisfies $P_k \le 0$ if and only if the predicate is true. Therefore, the assertion validation is transformed into a constraint optimization problem:

$$\begin{aligned} &\underset{\rho_{T_1}, \rho_{T_2}}{\text{maximize}}\ P_3(\rho_{T_1}, \rho_{T_2}), \\ &\text{subject to } P_1(\rho_{T_1}) \le 0, \\ &\qquad\qquad P_2(\rho_{T_2}) \le 0, \end{aligned} \tag{10}$$

where P_3 is guarantee and P_1, P_2 are assumptions. We substitute the ρ_{T_1} and ρ_{T_2} with the characterized approximation functions $\rho_{T_1} = f_1(\rho_{\text{in}})$ and $\rho_{T_2} = f_2(\rho_{\text{in}})$ to check whether the runtime states satisfies the assertion:

$$\begin{aligned} &\underset{\rho_{\text{in}}}{\text{maximize}}\ P_3(f_1(\rho_{\text{in}}), f_2(\rho_{\text{in}})), \\ &\text{subject to } P_1(f_1(\rho_{\text{in}})) \le 0, \\ &\qquad\qquad P_2(f_2(\rho_{\text{in}})) \le 0. \end{aligned}$$

As ρ_{in} is represented by parameters $\{\alpha_i\}$ in the approximation function (Theorem 1). We directly optimize $\{\alpha_i\}$ to find the maximum. The correctness of the assertion equals:

$$\begin{aligned} assert(T_1, T_2) \equiv \text{if } \max[P_3(\{\alpha\})] &\le 0 \rightarrow \text{true} \\ &> 0 \rightarrow \text{false}. \end{aligned}$$

The optimization can be efficiently solved by current state-of-the-art optimizers, such as stochastic gradient descent [56], genetic algorithm [24] and quadratic programming [51]. For example, for the assertion of quantum teleportation in Equation 7, the verification aims to check that:

$$\begin{aligned} &\underset{\rho_{\text{in}}}{\text{maximize}} \left\| \rho_{\text{in}} - f_2(\rho_{\text{in}}) \right\|, \\ &\text{subject to } \left\| \rho_{\text{in}} \rho_{\text{in}}^{\dagger} - \rho_{\text{in}} \right\| \le 0, \\ &\qquad\qquad \left\| f_2(\rho_{\text{in}})\ f_2(\rho_{\text{in}})^{\dagger} - f_2(\rho_{\text{in}}) \right\| \le 0. \end{aligned}$$

Here, $\rho_{T_1} = \rho_{\text{in}}$, as tracepoint T_1 labels the input of the program.

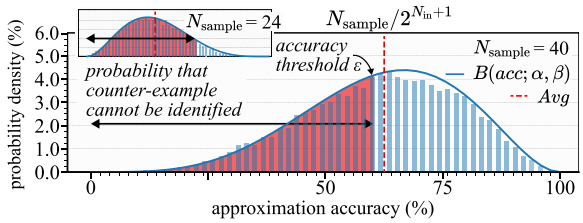


Figure 6. Distribution of approximation accuracies under various inputs.

6.2 Confidence Estimation

The approximation accuracy may not support discriminating errors in the corner cases, making the verification result invalid for all inputs of the program. Such inaccuracy can be alleviated by sampling more inputs. We propose a quantitative method to analyze the relation between the confidence and the number of samples, which helps programmers estimate the overhead to achieve the expected confidence.

The confidence is mainly determined by the approximation accuracy and an accuracy threshold ϵ defined to identify the counter-example. Confidence measures the probability that the verification result is valid for all inputs. Conversely, 1−confidence is the probability that at least one counter-example exists but is not identified, as their approximation accuracies are smaller than the accuracy threshold ϵ.

As shown in Figure 6, we observe that the approximation accuracies under various inputs follow the Beta distribution $B(\beta_1, \beta_2)$. Parameters β_1 and β_2 are determined by the number of samples N_{sample} and the characteristic of the quantum program. Besides,

$$\frac{\beta_1}{\beta_1 + \beta_2} = \frac{N_{\text{sample}}}{2^{N_{\text{in}}+1}}$$

is the average accuracy of case 2 in Theorem 2.

The values of β_1 and β_2 can be characterized by running a set of benchmarking inputs. In the Beta distribution, the probability that the approximation accuracy *acc* of a counter-example is smaller than the accuracy threshold ϵ, which can not be identified, is

$$P(acc < \epsilon) = \int_0^{\epsilon} B(x; \beta_1, \beta_2) dx,$$

which equals the integral of the Beta probability function $B(x; \beta_1, \beta_2)$ between range $[0, \epsilon]$.

Theorem 3 (Confidence)**.** The confidence that the program is correct when the validation does not find the counter-example is:

$$\text{confidence} = 1 - P(acc < \epsilon). \tag{11}$$

Overall, Here, we assume that the program only has one counter-example to simplify the estimation. An error program usually has more counter-examples. So the real confidence is higher, as it equals the probability that all these counter-examples have accuracy below the threshold, which is $1 - P(acc < \epsilon)^{N_{\text{c-e}}}$. $N_{\text{c-e}}$ is the number of counter-examples. Thus, we provide a lower-bound estimation of the confidence.

6.3 MorphQPV Complexity

Overall, the complexity of MorphQPV in verifying an assertion is determined by the following parameters: the number of qubits of inputs N_{in}, the number of qubits involved in two tracepoints N_{T_1}, N_{T_2}, and the number of inputs executed in input sampling N_{sample}.

- In the input sampling, state tomography at each tracepoint has $\mathcal{O}(e^{N_{T_1}})$ or $\mathcal{O}(e^{N_{T_2}})$ complexity [10]. N_{sample} input samples leads to complexity of $\mathcal{O}(N_{\text{sample}}\ (e^{N_{T_1}} + e^{N_{T_1}}))$. Input sampling is a one-shot process that does not need repeated running during verification.
- In the assertion validation, the optimization augments to validate each assertion consists of N_{sample} parameters $\{\alpha_i\}$. Assume that we adopt the quadratic programming solver to find the maximization. The optimization requires up to $\mathcal{O}(N_{\text{sample}}^3)$ number of iterations to find the global maximization [17].

Overall, to ensure 100% confidence, the characterization requires 100% accuracy with $2^{N_{\text{in}}+1}$ sampled inputs. The complexity of the chracterizaion is upper-bounded by $\mathcal{O}(2^{N_{\text{in}}+1}\ (e^{N_{T_1}} + e^{N_{T_2}}))$. The complexity of the validation is upper-bounded by $\mathcal{O}(2^{3N_{\text{in}}+3})$.

7 Case Study

We debug three algorithms as case studies, including quantum lock [30] (QL), quantum neural network [23] (QNN), and Quantum Random Access Memory (QRAM). These three quantum algorithms are not discussed by prior assertion works for two reasons: First, verification of these programs relies on the search for error inputs. However, the error inputs take a small proportion of the entire input space. For example, there is only one input leading to the error output in QL, while other outputs cannot help identify the bug. Second, they require complex comparisons to discriminate the error. For example, we compare the intermediate states of a QNN model with other QNN models, which requires obtaining the density matrix on the classical computer.

7.1 Case Study 1: Quantum Lock

The importance of QL has been introduced in Section 1. Moreover, the QL program is also known as the phase kickback module that is used to provide quantum speedup in many important quantum algorithms, including the Bernstein–Vazirani algorithm [4] and the quantum phase estimation algorithm [16]. A QL program is encoded with a key. If and only if the input state equals the key, the output is $|1\rangle$. If the input state does not equal the key, the output is $|0\rangle$.

$$\rho_{output} = \begin{cases} |1\rangle\langle 1|, & \text{if } \rho_{\text{in}} = |\text{key}\rangle\langle\text{key}| \\ |0\rangle\langle 0|, & \text{if } \rho_{\text{in}} \neq |\text{key}\rangle\langle\text{key}| \end{cases}.$$

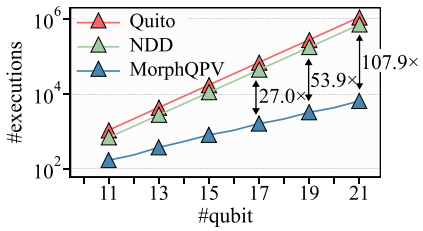


Figure 7. Numbers of sampled inputs to identify bugs in the quantum lock program.

As shown in Figure 1 (a), by inserting a controlled gate with an unexpected key, the program also outputs 1 with this unexpected key.

Verification of unexpected keys. The objective of Verification is to check whether the unexpected key exists. For a N-qubit QL program, the input space consists of 2^{N-1} bit strings. By exhaustively testing each bit string with the state assertion [20, 28, 29], the number of program executions follows the hypergeometric distribution, leading to close to $\mathcal{O}(2^{N-1}/2)$ complexity.

We first inject tracepoint into the QASM language of QL program [11]:

```
1 T 1 q[2,3,4]; // add tracepoint T1 on qubits 2,3,4.
2 h q[1];
3 x q[2,3,4];
4 mcz q[1,2,3],q[4];
5 x q[2,3,4];
6 h q[1];
7 T 2 q[1]; // add tracepoint T2 on qubits 1,
```

and specify the assertion:

$$\text{assume: } \underbrace{P_1(\rho_{\mathrm{T}_1}) = (\rho_{\mathrm{T}_1} \neq |\text{key}\rangle\langle\text{key}|)}_{\text{when input is not } |\text{key}\rangle},$$

$$\text{guarantee: } \underbrace{P_3(\rho_{\mathrm{T}_1}, \rho_{\mathrm{T}_2}) = \left\|\rho_{\mathrm{T}_2} - |0\rangle\langle 0|\right\|}_{\text{output } |0\rangle\langle 0|}.$$

We apply `Gurobi` [17], a quadratic programming solver, to find the maximum of the objective function in the assertion validation. Figure 7 presents the number of program executions required by Quito [47], NDD [29] and our method. MorphQPV achieves 107.9× speedup compared to the baselines (from 9.3×10^5 program executions to 8,974) for the 21-qubit QL algorithm. Note that the speedup exponentially grows as the number of qubits increases.

7.2 Case Study 2: Quantum Neural Network

In this case study, we debug the QNN model that predicts the species of Iris flowers. The Iris dataset includes 100 flowers with 4 attributes (e.g., width of the petal) and 2 species (e.g., Setosa and Virginica). Figure 8 presents the program of the QNN model, which consists of an encoder and layers of parameterized gates. The attributes of the flower are input into the program via an encoder. The expectation on the

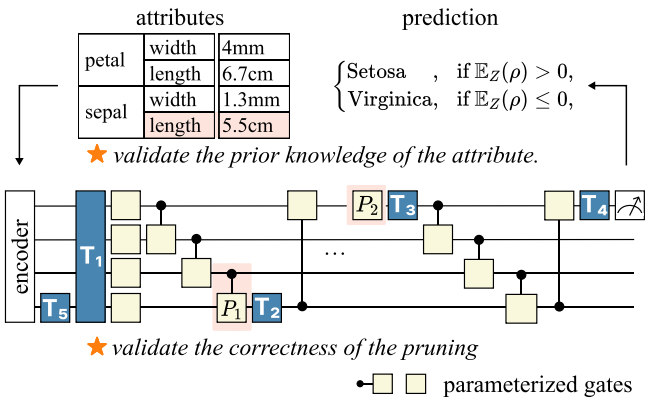


Figure 8. Quantum neural network to classify the species of Iris flowers.

Z-axis of the first qubit at the end of the program determines the prediction output. If $\mathbb{E}_Z(\rho) > 0$, the flower is Setosa. If $\mathbb{E}_Z(\rho) \leq 0$, it is Virginica. MorphQPV provides a new ability to verify the model after the gate pruning and validate the prior knowledge to improve the model interpretability.

Verification of gate pruning. Current quantum gates suffer from noise, motivating works [19, 45] to prune unimportant gates. For example, in the case study, the pruning removes two gates (P_1 and P_2 in Figure 8). We expect that the prediction does not change after the pruning. Besides, if the prediction changes, we should find the incorrectly pruned gates.

To identify the incorrectly-pruned gates, we inject tracepoints T_1 after the encoder, T_2 and T_3 after each pruned gate, and T_4 before the output. We apply an assume-guarantee assertion to check whether each tracepoint $\mathrm{T}_i \in \{\mathrm{T}_2, \mathrm{T}_3, \mathrm{T}_4\}$ remains in similar states before and after the pruning. We have QNN models before pruning QNN^* and after pruning QNN'. The assertion is

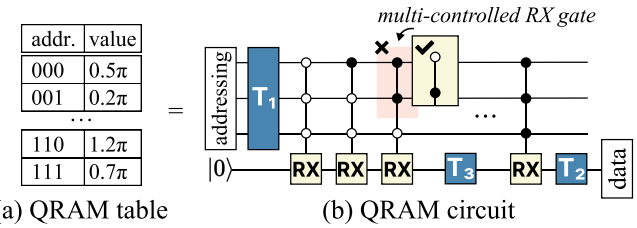
$$\text{assume: } \underbrace{P_1(\text{QNN}^*.\rho_{\mathrm{T}_1}) = \left\|\text{QNN}^*.\rho_{\mathrm{T}_1}\text{QNN}^*.\rho_{\mathrm{T}_1}^{\dagger} - \text{QNN}^*.\rho_{\mathrm{T}_1}\right\|}_{\text{QNN}^*.\rho_{\mathrm{T}_1} \text{ is pure}},$$

$$\underbrace{P_2(\text{QNN}'.\rho_{\mathrm{T}_1}) = \left\|\text{QNN}'.\rho_{\mathrm{T}_1}\text{QNN}'.\rho_{\mathrm{T}_1}^{\dagger} - \text{QNN}'.\rho_{\mathrm{T}_1}\right\|}_{\text{QNN}'.\rho_{\mathrm{T}_1} \text{ is pure}},$$

$$\text{guarantee: } P_3(\text{QNN}^*.\rho_{\mathrm{T}_i}, \text{QNN}'.\rho_{\mathrm{T}_i}) = (\left\|\text{QNN}^*.\rho_{\mathrm{T}_i} - \text{QNN}'.\rho_{\mathrm{T}_i}\right\| \leq \beta),$$

where β is a distance threshold. Tracepoints states $\text{QNN}^*.\rho_{T_i}$ and $\text{QNN}'.\ \rho_{T_i}$ are obtained by the approximation. Besides, we adopt the optimization strategy *Strategy-adapt* to reduce the overhead in the initializing approximation function.

If the assertion toward the output fails, we can apply a binary search to check each pruned gate. Verifying pruning is difficult using prior assertion techniques, as they have tested various inputs and run quantum state tomography at each tracepoint, requiring exponential complexity for each input.



(a) QRAM table (b) QRAM circuit

Figure 9. Example of the quantum random access memory.

Verification of prior knowledge. Studying the interpretability of the machine learning model is also an important aspect, as it provides optimization opportunities to improve the model accuracy [42] and assist the domain experts [7]. The Verification is performed with some prior knowledge. For example, biologists suggest that the flowers with sepal lengths in the range $[4, 6]$ cm belong to Setosa. Verifying the QNN model can give the Biologist an answer to this prior knowledge.

The Verification can be performed by prior works with a test dataset consisting of pre-collected inputs. However, works have pointed out that accuracy in the test dataset cannot guarantee high generality [36], as the test dataset only covers a small proportion of the input space. MorphQPV can declare an assertion to verify this prior knowledge, which checks that all model outputs satisfy the prior knowledge. We inject a tracepoint T_5 to the fourth qubit at the beginning of the program (see Figure 8):

$$\text{assume: } \underbrace{P_1(\rho_{T_5}) = (4 \le \rho_{T_5}[1][1] \le 6)}_{\text{when the sepal length is in [4, 6] cm}},$$

$$\text{guarantee: } \underbrace{P_2(\rho_{T_4}) = (\mathbb{E}_Z(\rho_{T_4}) > 0)}_{\text{the output should be Setosa}}.$$

Here, we assume that the sepal length is encoded into the amplitude of $|1\rangle$ of the fourth qubit, normalized to $[0, 2\pi]$ in the implementation. The Verification aims to find the flower that belongs to Virginica but in the range of $[4, 6]$ cm. If the assertion fails, the prior knowledge is wrong. If the assertion passes, the prior knowledge is correct.

7.3 Case Study 3: Quantum Random Access Memory

In this section, we debug QRAM, which is an important quantum program that allows data to be accessed based on addresses. QRAM is a fundamental component of many quantum applications [3, 43], which is widely studied [14, 50], as the information stored in it can be repeatedly reused without being destroyed by decoherence errors and non-duplicability of the quantum state.

As shown in Figure 9 (a), similarly to classical RAM, the information stored in QRAM is represented as a table, where each address i identifies a value $\theta_i \in [0, 2\pi]$. The size of the table is 2^N for N addressing qubits. QRAM allows inputting a superposition state to the addressing qubits to query information. The result is output as the state of a data qubit. Mathematically, for input state $\sum_{i=0}^{2^N} \lambda_i |i\rangle$. The output of the data qubit is

$$\sum_{i=0}^{2^N} \lambda_i |\theta_i\rangle,$$

where we define $|\theta_i\rangle = cos\theta_i |0\rangle + sin\theta_i |1\rangle$. For example, when the input state is $\frac{\sqrt{2}}{2}|00\rangle + \frac{\sqrt{2}}{2}|11\rangle$, the output state is $\frac{\sqrt{2}}{2}(cos\theta_{00} + cos\theta_{11})|0\rangle + \frac{\sqrt{2}}{2}(sin\theta_{00} + sin\theta_{11})|1\rangle$ As shown in Figure 9 (b), in the circuit implementation of QRAM, the values are read into the data qubit by sequentially applied multi-controlled RX gates. The controlled state corresponds to the address i, while the rotation of the gate corresponds to value θ_i. For example, address 101 and the value $\pi/3$ are implemented by the 5^{th} multi-controlled RX gate with rotation $\pi/3$, controlled by the state $|101\rangle$ of the addressing qubit.

Verification of error address. It is important to check the data in QRAM and locate the error address when the data is wrong. Since the QRAM input space includes all superposition states, it is impossible to traverse the whole space. MorphQPV provides an efficient debugging method to verify QRAM and identify the error address. As shown in Figure 9 (b), we first define tracepoint T_1 at the start of the addressing qubit and tracepoint T_2 at the end of the data qubit to check the overall functionality of QRAM:

$$\text{assume: } \underbrace{P_1(\rho_{T_1}) = \left\|\rho_{T_1} - \sum_{i,j=0}^{2^N} \lambda_i\lambda_j^* |i\rangle\langle j|\right\|}_{\text{when input state is } \sum_{i=0}^{2^N}\lambda_i|i\rangle},$$

$$\text{guarantee: } \underbrace{P_2(\rho_{T_2}) = \left\|\rho_{T_2} - \sum_{i,j=0}^{2^N} \lambda_i\lambda_j^* |\theta_i\rangle\langle\theta_j|\right\|}_{\text{the output state is } \sum_{i=0}^{2^N}\lambda_i|\theta_i\rangle}.$$

If the program is incorrect, we apply a binary search to identify the error address. We inject tracepoint T_3 into the middle of the program and check the addresses before T_3 by assertion:

$$\text{assume: } P_1(\rho_{T_1}) = \left\|\rho_{T_1} - \sum_{i,j=0}^{2^N/2} \lambda_i\lambda_j^* |i\rangle\langle j|\right\|,$$

$$\text{guarantee: } P_2(\rho_{T_3}) = \left\|\rho_{T_3} - \sum_{i,j=0}^{2^N/2} \lambda_i\lambda_j^* |\theta_i\rangle\langle\theta_j|\right\|,$$

which means that the values can be correctly read for the first 1/2 addresses. If the error exists in the first 1/2 addresses, we inject a tracepoint to validate the first 1/4 addresses. Otherwise, we will validate the second 1/2 addresses. The search is repeated until the error address is identified.

Figure 10 presents the numbers of sampled inputs to identify the error in QRAM by Quito [47], NDD [29] and MorphQPV . MorphQPV shows a 31,563.2× reduction of sampling inputs compared to Quito. This reduction is even more significant than debugging the QL program since the input space of QRAM is a larger superposition state with more optimization opportunities, while the inputs of the QL program only consist of classical states.

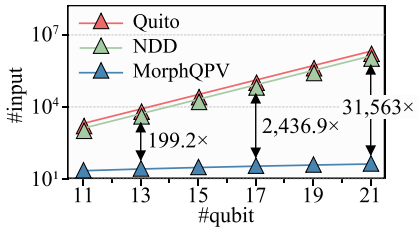


Figure 10. Numbers of sampled inputs to identify bugs in the quantum random access memory program.

Table 2. Comparsion of expressiveness of MorphQPV to four assertion techniques. "Full", "Part", or "No" means the technique has full, part, or no ability.

	Stat [20]	Proj [27]	NDD [28]	SR [13]	MorphQPV
Verified object	Probability distribution	Mixed state	Mixed state	Mixed state	Mixed state & Evolution
Comparison	Part	Equal & In	Equal & In	Equal & In	Full
Interpretability	Part	No	No	No	Full
Debug circuit with feedback	No	No	No	Full	Full

8 Comparison With Prior Works

We compare MorphQPV with four recent works to implement quantum program assertion and recent work to determine test inputs in the expressiveness, confidence, and overhead. The quantum program assertions are based on statistical analysis (Stat) [20], projection-based measurement (Proj) [27], non-destructive discrimination (NDD) [28], and symbolic reasoning (SR) [13]. Quito determines test inputs based on the grid search [47].

8.1 Expressiveness Analysis

In terms of the verified object, MorphQPV directly verifies the evolution of the program, which is independent of specific inputs, while prior works [13, 20, 27, 29] only verify single states under specific inputs. Stat [20] check the amplitudes that ignore the phase. When checking the qubit state, the predicates of MorphQPV, Proj [27], NDD [29], SR [13] allows verifying the mixed state.

In terms of the comparison type, Proj [27], NDD [29] verify assertion by injecting circuit blocks to the program. They can only check if the runtime state is in a specified state set and cannot support greater than or less than comparison. Moreover, they cannot compare the states at different times of the program. By the isomorphism-based characterization, MorphQPV obtains the density matrix of the states on the classical computer, supporting complex comparison types and comparisons of states at different times.

In terms of interpretability, Stat [20] outputs the probability distribution of the error state when the assertion fails. Proj [27], NDD [29] output no information. MorphQPV provides the counter-example and the density matrix of program intermediate states. Moreover, MorphQPV estimates the confidence when the program is verified to be correct.

In terms of the ability to debug nondeterministic programs with simple feedback, Stat [20], Proj [27], and NDD [20] have to redefine the predicates for different mid-measurement results. SR [13] can verify nondeterministic programs. As suggested in the characterization, the approximation of MorphQPV is applicable for mid-measurement and feedback. The assume-guarantee assertion can also verify the program execution under different conditions by asserting the state after the mid-measurement (cf. the example of quantum teleportation in Section 4).

Table 3. Benchmarking programs used in the evaluation.

Abrv.	Program	Abrv.	Program
QNN	Quantum neural network [23]	QL	Quantum lock [40]
QEC	Quantum error correction [37]	Shor	Shor's algorithm [9]
XEB	Cross entropy benchmarking [2]		

8.2 Success Rate Analysis

We compared the success rate of MorphQPV with NDD [27] and Quito [47] in Table 4 with five benchmarking programs listed in Table 3. We adopted mutation testing [21] to evaluate the confidence in verification. We generated 100 test cases for each program, all with bugs. These bugs are implemented by randomly injecting phase gates into the programs. To make a fair comparison, each verification method tests five inputs when verifying each program. The success rate is the probability that the verification result from the method is valid for all inputs.

Overall, MorphQPV achieves the maximum 100% success rate to identify bugs in all five benchmarks. The success rate of Quito [47] decreases exponentially as the number of qubits grows, as the size of the input space increases. Quito shows a less than 50% success rate when debugging the QL and XEB programs with more than three qubits because most errors are in the phase of the tracepoint states, while Quito only validates the probability distribution of these states.

NDD [28] shows a high success rate in QEC, Shor, and XEB, as it can identify phase differences so that a few inputs can help find the errors. However, NDD shows a confidence of 0% in the 9-qubit QL program, as the program has only one counter-example. Besides, NDD cannot debug QNN, as debugging QNN needs to compare the value of expectations, while NDD only supports equal or inclusive comparison.

8.3 Overhead Analysis

Table 4 also compares the overhead of the programs. The overhead is defined as the number of quantum operations introduced to validate the assertion. For each program, we

Table 4. Comparison of the success rate and the overhead in the verification.

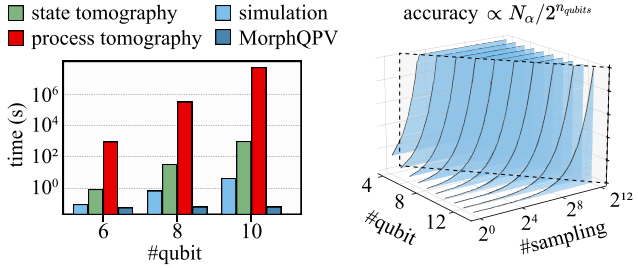
Bench-mark		Success rate (%)			Overhead ($\times 10^3$ operations)		
		NDD [28]	Quito [47]	Morph-QPV	NDD [28]	Quito [47]	Morph-QPV
QL	3q	38	36	100	10.0	5.0	5.0
	5q	12	11	100	10.0	5.0	5.0
	7q	3	2	100	10.0	5.0	5.0
	9q	0	0	100	10.0	5.0	5.0
QNN	3q	/	100	100	/	5.0	5.0
	5q	/	100	100	/	5.0	5.0
	7q	/	67	100	/	5.0	5.0
	9q	/	50	100	/	5.0	5.0
QEC	3q	100	0	100	1.9×10^3	5.0	101.5
	5q	100	0	100	4.3×10^4	5.0	175.0
	7q	100	0	100	7.5×10^7	5.0	326.5
	9q	100	0	100	2.4×10^{10}	5.0	520.5
Shor	3q	100	0	100	2.0×10^3	5.0	101.0
	5q	100	0	100	4.3×10^4	5.0	177.0
	7q	100	0	100	9.0×10^7	5.0	306.5
	9q	100	0	100	2.8×10^{10}	5.0	488.0
XEB	3q	100	100	100	2.0×10^3	5.0	103.0
	5q	100	50	100	4.4×10^4	5.0	181.5
	7q	100	44	100	8.5×10^7	5.0	323.5
	9q	100	37	100	2.6×10^{10}	5.0	505.5

executed 10^3 shots to obtain the measured probability distribution. For example, when NDD [29] debugs the QL program, it inserts a NOT gate following measurement to the output qubit, resulting in overall 10^4 $((1+1)\times10^3\times5)$ operations for testing 5 inputs.

Compared to NDD [29], which validates the correctness of the mixed states, MorphQPV achieves a high overhead optimization (e.g., from 2.8×10^{10} to 488.0 for the 9-qubit Shor algorithm). This originates from the fact that MorphQPV applies an approximation function to obtain the density mat rix instead of executing the program. NDD [29] requires synthesizing unitary gates to basis gates for the sub-space projection. The resulting number of gates exponentially increases as the number of qubits grows in the common case. For example, to verify a 9-qubit mixed state, these methods may require up to 2.1×10^4 gates and 144.4 hours with the state-of-the-art synthesis method [53]. Qutio [47] only checks the probability distribution, which has the minimum overhead (5.0×10^3) but the lowest confidence in the verification. For the QL and QNN programs, MorphQPV also achieves the minimum overhead by leveraging the optimization strategies to prune the characterization space in Section 5.4.

9 Detailed Evaluation

This section evaluates the detailed techniques proposed in this work. MorphQPV was implemented with `Python` (3.9.13) and the `NumPy` package (1.23.1). The objective optimization involved in assertion validation was performed by the `Groubi` (11.0.0) solver. The `Pennylane` (0.33.1) package was used to simulate the program execution. The number of shots of each program execution was set to 1000. All data were tested



(a) Comparison of the computation time to obtain the intermediate program states.

(b) Accuracy of approximation under different numbers of qubits and sampled inputs.

Figure 11. Evaluation of Theorem 1 and Theorem 2.

on a Linux platform with two 64-core AMD EPYC 9554 CPUs and 1.6TB memory. The duration of the program on the real-world quantum hardware was estimated based on the IBMQ quantum cloud platform, which has 60*ns* single-qubit gate time, 340*ns* two-qubit gate time, and 732*ns* readout time. We evaluated MorphQPV on five quantum algorithms listed in Table 3. The number of input qubits and output qubits of these algorithms were set to their overall number of qubits for fair comparison.

9.1 Evaluation of Theorems

Evaluation of Theorem 1. The approximation is an efficient method to obtain the program intermediate states when comparing its computation time to obtain tracepoint states under each input with the quantum simulation on `Qiskit` (0.45.0), quantum state tomography [10] and the quantum process tomography [41] in Figure 11 (a). MorphQPV achieves 74.3×, $1.2\times10^4\times$, and $7.3\times10^6\times$ reduction compared to the simulation, state tomography, and process tomography in 10-qubit programs, respectively. The quantum process tomography requires 11.4 days to obtain the density matrix of a 10-qubit program state. Compared to it, MorphQPV takes less than 0.5 seconds. This is attributed to the fact that MorphQPV only involves simple summation of density matrices, while simulation requires numerous matrix-matrix multiplications. State tomography and process tomography have to measure the state on all basis.

Evaluation of Theorem 2. Figure 11 presents the average approximation accuracy of 5 algorithms under different numbers of qubits and sampled inputs. The accuracy curve is close to case 2 of Theorem 2, as the randomly generated inputs used in the experiments are more likely to stay in case 2. The maximum number of sampled inputs in the experiment to ensure the accuracy 100% is consistent with the theorem, which increases 2× as the number of qubits increases 1.

Evaluation of Theorem 3. Figure 12 compares the estimated confidence with the real success rate of generating the correct verification result in 7-qubit programs. Bugs are implemented by the mutation testing introduced in Section 8.2. Since Theorem 3 presents a lower bound for confidence, the real success rates of programs are above the theoretical value.

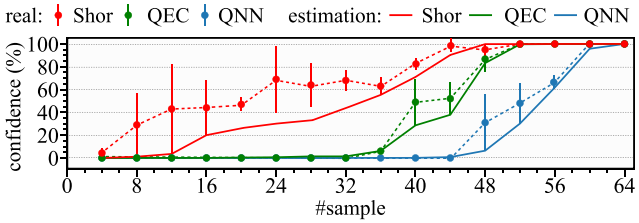
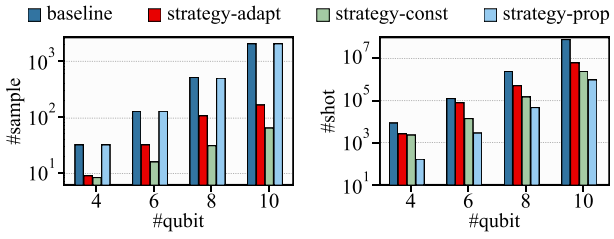


Figure 12. Evaluation of confidence estimation (Theorem 3).



(a) Number of sampled inputs with 3 optimization strategies.

(b) Number of shots with 3 optimization strategies.

Figure 13. Evaluation of three pruning strategies of the program characterization introduced in Section 5.4.

The QEC program has fewer counter-examples, and its curve is close to the estimated confidence, while the Shor program has more counter-examples and a higher success rate.

9.2 Evaluation of Techniques

Evaluation of space pruning strategies. Figure 13 (a) presents the ablation study of the space pruning strategies in Section 5.4 in the QNN and Shor algorithm. Strategy-adapt and Strategy-const decrease the number of sampled inputs by 12.4× and 32.0× compared to the characterization without optimization, respectively. The reduction of strategy-adapt comes from pruning unimportant eigenstates. For example, for the 10-qubit QNN program, the 2048 samples are pruned to 90 (22.8× reduction) with 95% accuracy. Strategy-const optimizes the overhead by pruning the input space. For example, for the 10-qubit Shor program, we set the state of half of the input qubit constant, leading to 32.0× reductions. In Figure 13 (b), strategy-prop achieves 82.1× reduction of shots in 10-qubit programs, which results from eliminating the state tomography in the characterization. For example, to validate a 6-qubit state in the Shor program, if only amplitudes are involved in the assertion, the characterization takes 63.0× fewer shots compared to the baseline with no optimization.

Evaluation of approximation accuracy on the noisy quantum simulator. Figure 14 presents the approximation accuracy of 5-qubit and 15-qubit Shor and QNN algorithms on the noisy quantum simulator. We employed the `Qiskit` simulator with the noise model of the IBM Cairo quantum device that has 99.45% fidelity for single-qubit gates and 98.4% fidelity for two-qubit gates. When tracepoints are injected into the start and end of the program, the approximation

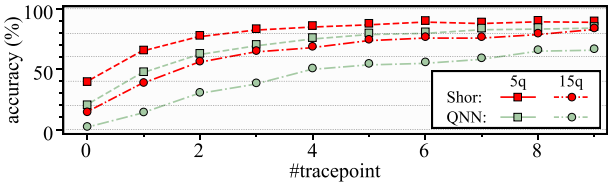
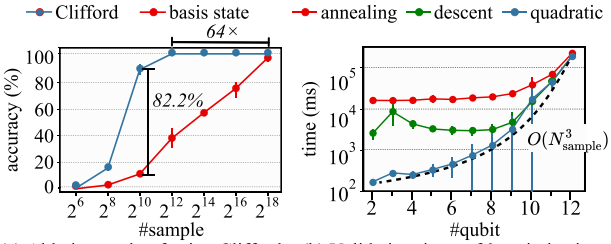


Figure 14. Approximation accuracy on the noisy quantum simulator. The accuracy is optimized by injecting different numbers of intermediate tracepoints.



(a) Ablation study of using Clifford group in 9-qubit programs.

(b) Validation times of 3 optimization solvers.

Figure 15. Evaluation of using Clifford group and optimization times in the validation.

accuracy is 13.7% and 1.6% for 15-qubit Shor and QNN algorithms, respectively. The inaccuracy comes from the fact that tracepoints are far from each other, leading to large decoherence errors. We can inject intermediate tracepoints to minimize this noise. Specifically, for tracepoints T_1 and T_2, we inject an intermediate tracepoint T_3 between them. We can characterize relations $\rho_{T_3} = f_1(\rho_{T_1})$ and $\rho_{T_2} = f_2(\rho_{T_3})$ to obtain the relation between ρ_{T_1} and ρ_{T_2}, which equals $\rho_{T_2} = f_2(f_1(\rho_{T_1}))$. By injecting four intermediate tracepoints, the optimization improves the approximation accuracy from 1.6% to 13.6% for the 15-qubit QNN algorithm. The accuracy is further improved to 65.0% after injecting nine tracepoints.

Ablation study of adopting Clifford group. Figure 15 (a) shows the ablation study of using the Clifford group in the characterization of 9-qubit benchmarking algorithms. Overall, the Clifford group reduces the number of sampled inputs to achieve 100% accuracy by 64.0× compared to using basis states as sampled inputs, as the states prepared by the Clifford group show entanglement and superposition, which are more representative compared to the basis states. When the number of sample inputs is 2^{10}, the Clifford group also achieves 82.2% accuracy improvement (from 10.9% to 93.1%).

Evaluation of optimization times based on different solvers in validation. In Figure 15 (b), we evaluated the optimization time of the resulting constraint objective function on three optimization solvers, including stochastic gradient descent [56], genetic algorithm [24] and quadratic programming [51] solvers. The quadratic programming solver shows the minimum validation time for programs with less than 12 qubits, which requires less than 12 minutes to find the global optimal. Note that the local optimal is usually enough as it can also determine the correctness, which further reduces

the optimization time. We observed that the curve of the validation cost polynomially grows as the number of sampled inputs increases.

10 Related Work

Quantum program can be verified by deductive verification [8, 34, 52, 54, 55, 57] and runtime assertion [13, 20, 27–29]. The deductive verification is mainly performed by extending the classical deductive verification frameworks, such as Hoare logic [52, 57] and semantic model [34]. However, this reasoning cannot be performed on quantum hardware, leading to high computational overhead, and requires human experts that limit the degree of automation. An orthogonal line is the assertion. The assertion validation can be performed by projection [27], swap-test [29], or focus on specific properties of the program, such as the amplitudes [20] and purity [55] of runtime states. Verification can also be performed by checking the equivalence of programs [33, 48, 49], while it requires the correct program for comparison. Gleipnir [39] enables a rigorous and efficient input-aware error analysis of the quantum programs. It approximates the program by tensor networks. Compared to it, the approximation of MorphQPV eliminates the simulation for each input. OSCAR [18] debugs the optimization of the variational quantum algorithm by constructing the loss function landscape in the parameter space. MorphQPV provides the ability to construct the landscape in the input space. The optimization-based validation gains inspiration from Ren et al. [35] that employs adversarial learning to search for counter-examples in quantum neural networks. MorphQPV generalizes the search to a variety of quantum algorithms and provides a theoretical estimation of the verification confidence.

11 Conclusion

We propose MorphQPV to boost the success rate in verifying the quantum program. We design an assume-guarantee assertion to specify the expected relations between states in the program. We then exploit the isomorphism property of the program to characterize ground-truth state relation as classical approximation functions. We check whether ground-truth relations satisfy the assertion by combining them into a constraint optimization problem. MorphQPV can output the counter-example and estimate the confidence of the verification. MorphQPV achieves an up to 107.9× reduction of program executions and an improvement of the 3.3×-9.9× probability of success when debugging five algorithms.

Appendix

A Proof of Theorems

Proof of Theorem 1. Assuming that the evolution of the gates between the input and the tracepoint is defined as unitary U, in the input sampling (Section 5.1), $\sigma_{\mathrm{T},i} = U\sigma_{\mathrm{in},i}U^\dagger$. We can obtain Equation 9:

$$\begin{aligned}\rho_{\mathrm{T}} &= U\rho_{\mathrm{in}}U^\dagger \quad \text{// Equation 8}\\ &= \sum\nolimits_i \alpha_i (U\sigma_{\mathrm{in},i}U^\dagger) \quad \text{// move } U \text{ and } U^\dagger\\ &= \sum\nolimits_i \alpha_i \sigma_{\mathrm{T},i} \quad \text{// } \sigma_{\mathrm{T},i} = U\sigma_{\mathrm{in},i}U^\dagger\end{aligned} \tag{12}$$

When there is a mid-measurement between the input and the tracepoint, the evolution of the program before and after the program is defined as U_1 and U_2. The measurement operator is defined as O, and the approximation holds as

$$\rho_{\mathrm{T}} = U_2 \frac{O(U_1\rho_{\mathrm{in}}U_1^\dagger)O^\dagger}{\mathbb{E}_O[U_1\rho_{\mathrm{in}}U_1^\dagger]}U_2^\dagger$$

As operators U_1, U_2, and O are all linear, we can prove the approximation of these operators holds following the same step of Equation 12.

When there is feedback in the program, i.e., the classical controlled unitary gate, the quantum state is operated by unitary U_i with probability p_i, which is the probability that basis i is measured. The evolution is formulated as

$$\rho_{\mathrm{T}} = \sum_i p_i U_i \rho_{\mathrm{in}} U_i^\dagger$$

The relationship between states ρ_{T} and ρ_{in} in the equation is also linear, so the approximation holds in the program with feedback.

Proof of Theorem 2. The accuracy of the approximation is measured via the Hilbert–Schmidt inner product [22]. Therefore,

$$acc = tr(\sqrt{\rho_{\mathrm{approx}}\ \rho_{\mathrm{truth}}})^2,$$

where ρ_{approx} and ρ_{truth} are approximation state and ground-truth state, respectively. Since Hilbert–Schmidt's inner product is preserved by unitary evolution, the error of output equals the error of input state (Equation 8). For case 1, the approximation error of input is 0. For case 2, the approximation of input follows the same workflow as the quantum state tomography [31], which has been proven to be $N_{\mathrm{sample}} = 2^{N_{\mathrm{in}}+1}/(1-\epsilon)$. We can transform this equation to the equation in case 2.

B Comparison to Deductive Verification Methods

Table 5 and Table 6 compare the expressiveness, the success rate, and the overhead of MorphQPV with recent deductive verification methods. The experiment setup is the same as in Section 8.2.

In terms of expressiveness, the baselines all have limited verified objects and simple comparison types. None of them can verify the three cases in Section 7. NKA [34] and QHL [57] provide a part of interpretability by understanding the mathematical formulation. However, they cannot output the counter-example when the program is incorrect.

Table 5. Comparison of the expressiveness of MorphQPV to three deductive verification methods.

	KNA [34]	Twist [55]	QHL [57]	MorphQPV
Verified object	Expectation	Purity	Expectation	Mixed state & Evolution
Comparision	Equal or greater	Equal	Equal or greater	Full
Interpretability	Part	No	Part	Full

Table 6. Comparison of the success rate and the overhead of MorphQPV to two deductive verification methods.

	Benchmark	Success rate (%)			Overhead (seconds)		
		Twist [55]	Automa [8]	MorphQPV	Twist [55]	Automa [8]	MorphQPV
QEC	5q	98	100	100	0.3	0.3	0.1
	10q	98	100	100	4.5	1.2	0.1
	15q	99	100	100	156.5	3.1	0.4
	20q	100	100	100	5.9×10^3	4.8	8.0
Shor	5q	100	100	100	1.1	0.7	0.1
	10q	100	100	100	23.2	6.9	0.1
	15q	100	100	100	1.2×10^3	22.2	1.2
	20q	100	100	100	6.1×10^4	65.5	86.6
QNN	5q	/	/	100	/	/	0.1
	10q	/	/	100	/	/	0.1
	15q	/	/	100	/	/	1.9
	20q	/	/	100	/	/	82.9
XEB	5q	/	100	100	/	0.7	0.1
	10q	/	100	100	/	6.5	0.1
	15q	/	100	100	/	20.1	6.7
	20q	/	100	100	/	48.6	314.7

Twist [55] only outputs the purity that cannot help identify the source of error in most cases.

In terms of success rate, MorphQPV and two deductive methods all have a high success rate in identifying the bugs. However, both Twist [55] and Automa [8] cannot debug the QNN program. Twist also cannot verify the XEB program. This is because Twist validates the purity of the state, while the bugs in the QNN and XEB programs do not change the purity. Automa cannot verify the QNN program as it is based on the deduction, which cannot obtain the specific expectation of the program state.

In terms of overhead, Twist [55] suffers from high computational cost, which requires 6.1×10^4 s to verify a 20-qubit QEC program, as it relies on classical simulation to validate the purity. Automa [8] shows a small overhead, as it adopts a tree automata to speed up the runtime analysis. However, its complexity still exponentially increases as the number of qubits grows. Compared to them, the complexity of the MorphQPV is not determined by the overall number of qubits of the program but by the input qubits. The time-consuming part in MorphQPV is mainly the input sampling and the sampling time can be further reduced by parallel techniques.

C Artifact Appendix

C.1 Abstract

In this letter, we provide detailed information that will facilitate the artifact evaluation process. The artifact checklist section presents brief information about this artifact and outlines the basic requirements to reproduce the experiment results. Then, we describe the directory tree of our codebase and go into more detail about the requirements. Finally, in the experiment workflow section, we explain step by step how to reproduce the experiments.

C.2 Artifact check-list (meta-information)

Program: MorphQPV is implemented in Python.
Dataset: We use 5 algorithms to evaluate MorphQPV. A quantum neural network is one of the algorithms that uses the MNIST dataset.
Environment: Linux ubuntu 5.4.0 or MacOS 13.2.1.
Hardware: x86-64 CPU, 32GB Memory, 512GB of storage space.
Execution: An Internet connection is required to download the MNIST dataset and access the quantum device in IBMQ.
Metrics: Accuracy, the confidence of program verification, and execution time.
Results: The runtime logs are printed to the console, and the results of experiments are saved to CSV files or Graphs in `examples/`.
Experiments: Python scripts are provided in `examples/` to reproduce the results of the experiments.
Disk space required: 256GB.
Time needed to prepare workflow: Less than one hour.
Time needed to complete experiments: Less than one hour for each Python script, except *examples/fig7- quantum-lock_verify.py*. The overhead comparison experiments take much longer because they need to test over many benchmarks.
Code licenses: GNU GPLv3.
Archived: 10.5281/zenodo.10775069

C.3 Description

MorphQPV is built on Python scripts, which can be executed on Linux or MacOS systems. Below, we introduce the important files and directories in the artifact.
data/. This sub-directory includes code for benchmarking quantum algorithms (in data/Qbenchmark/) and the MNIST dataset downloaded from scripts.
examples/. This sub-directory includes the Python scripts to reproduce the experiments in the paper.
morphQPV/. This sub-directory stores the core code of MorphQPV, which implements the assume-guarantee pragma (rf. Definition 1 in the paper.), the characterization method (Section 5), and the optimization-based method to verify the program (Section 6).

Table 7. Relationship between the scripts and the experiments in the paper.

Content	Experiment	Script (in examples/)	Expected result	Notes
Overhead analysis	Numbers of samples to identify bugs in quantum lock	fig7-quantumlock_verify.py	Figure 7	Less than ten minutes
	Comparison of the verification success rate and overhead	table4-compare.py	Table 4	More than one hour
Evaluation of Theorems	Theorem 1: Approximation function	fig11a-theorem1.py	Figure 11 (a)	Less than ten minutes
	Theorem 2: Approximation accuracy	fig11b-theorem2.py	Figure 11 (b)	A few minutes
	Theorem 3: Evaluation of confidence estimation	fig12-confidence.py	Figure 12	A few minutes
Optimization comparison and ablation study	Evaluation of different optimization techniques	fig13-opt_strategy.py	Figure 13	Requires Internet connection, a few minutes
	Ablation study of using Clifford gates and basis gates	fig15a-ablation_study.py	Figure 15 (a)	More than one hour
	Runtime comparison of different optimization solvers	fig15b-solvers_compare.py	Figure 15 (b)	More than one hour

main_exp.py. This Python script gives an example using morphQPV to define and verify the assume-guarantee assertion.

morphQPV/assume_guarantee/. This sub-directory contains the source code to define the assume-guarantee pragma, the sampling of input-output pairs, and the solver to find the satisfied input states.

- **grammer.py** Implementation to the assume-guarantee pragma. The main class in this script is MorphQC, which defines the quantum gate, the tracepoint, and the assertion.
- **predicate.py** Template of predicates (cf. Definition 1 in the paper.) in MorphQPV, including the `IsPure`, `Equal`, `NotEqual`, `GreaterThan`, `LessThan`.
- **sample.py** Implementation to the Input Sampling in Section 5.1.
- **inference.py** Implementation to the Isomorphism-based Approximation in Section 5.2.
- **solver/.** Implementation to convert the assertion into a constrained optimization problem.
- **optimizer/.** Implementations to solve constrained optimization problems, including gradient descent, annealing, and quadratic programming.

morphQPV/baselines/. Each file named `[baseline].py` in this directory represents a baseline. For example, `ndd.py` corresponds to NDD and `stat.py` corresponds to Stat.

MorphQPV/execute_engine/. This directory contains the methods for executing the sampling circuits and getting the density matrix by simulation or quantum state tomography.

C.3.1 How to access.

DOI: 10.5281/zenodo.10775069
GitHub: https://github.com/JanusQ/MorphQPV/

C.3.2 Hardware dependencies.

The evaluation in the paper is performed on a server with AMD EPYC 9554 64-core Processor, 1511GB memory, and 32TB storage space. Running the code requires an x86-64 machine with at least 32GB memory and 512GB storage space.

C.3.3 Software dependencies.

The code has been tested in Python 3.9. We list all required Python packages in `requirements.txt` of the artifact.

C.4 Installation

1. Download the source code from the GitHub repository (https://github.com/JanusQ/MorphQPV/).
2. Create a virtual environment with a Python version at 3.9 by Anaconda.

```
conda create -n morphenv python=3.9
conda activate morphenv
```

3. Install required Python packages.

```
pip install -r requirements.txt
```

Please refer to the `README.md` in the artifact for a more detailed description.

C.5 Using MorphQPV

Users can 1) describe a quantum program, 2) state the assertion, and 3) verify the program by MorphQPV. We provide an example code in the `README.md` of the source code. There are hyper-parameters in each stage of the verification. we introduce them in `doc/morphconfig.md`.

C.6 Reproducing Experimental Results

Scripts. The `examples/` directory included the Python scripts to reproduce the experimental results in the paper. Table 7 lists the relationship between the scripts and the experiments. For example, run the following command at the root directory of the artifact,

```
python examples/table4-compare.py
```

Father the program is finished, we can get the results of Table 4 in the paper. A detailed description is presented in the `doc/evaluation.md`.

Results. The experimental results will be generated as graphs or CSV files in the `examples/` directory after running the scripts.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (No. 2023YFF0905200). This work was also funded Zhejiang Pioneer (Jianbing) Project (No. 2023C01036) and the National Natural Science Foundation of China under Grant (No. 61825205).

References

[1] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. Assessing the effectiveness of input and output coverage criteria for testing quantum programs. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 13–23. IEEE, 2021.

[2] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, Willliam Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P Harrigan, Michael J Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S Humble, Sergei V Isakov, Evan Jeffrey, Zhan Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandra, Jarrod R McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthhew Neeley, Charles Neil, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C Platt, Chris Quintana, Eleanor G Rieffel, Pedram Roushan, Nicholas C Rubin, Daniel Sank, Kevin J Satzinger, Vadim Smelyanskiy, Kevin J Sung, Matthew D Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[3] DV Babukhin. Harrow-hassidim-lloyd algorithm without ancilla post-selection. *Physical Review A*, 107(4):042408, 2023.

[4] Charles H Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM journal on Computing*, 26(5):1510–1523, 1997.

[5] Dik Bouwmeester, Jian-Wei Pan, Klaus Mattle, Manfred Eibl, Harald Weinfurter, and Anton Zeilinger. Experimental quantum teleportation. *Nature*, 390(6660):575–579, 1997.

[6] Sergey Bravyi and Dmitri Maslov. Hadamard-free circuits expose the structure of the clifford group. *IEEE Transactions on Information Theory*, 67(7):4546–4563, 2021.

[7] Richard J Chen, Judy J Wang, Drew FK Williamson, Tiffany Y Chen, Jana Lipkova, Ming Y Lu, Sharifa Sahai, and Faisal Mahmood. Algorithmic fairness in artificial intelligence for medicine and healthcare. *Nature biomedical engineering*, 7(6):719–742, 2023.

[8] Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. An automata-based framework for verification and bug hunting in quantum circuits. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1218–1243, 2023.

[9] D. Coppersmith. An approximate fourier transform useful in quantum factoring, 2002.

[10] Jordan Cotler and Frank Wilczek. Quantum overlapping tomographya. *Physical review letters*, 124(10):100401, 2020.

[11] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[12] Poulami Das, Christopher A Pattison, Srilatha Manne, Douglas M Carmean, Krysta M Svore, Moinuddin Qureshi, and Nicolas Delfosse. Afs: Accurate, fast, and scalable error-decoding for fault-tolerant quantum computers. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 259–273. IEEE, 2022.

[13] Yuan Feng and Yingte Xu. Verification of nondeterministic quantum programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 789–805, 2023.

[14] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Physical review letters*, 100(16):160501, 2008.

[15] DM Greenberger, MA Horne, and A Zeilinger. Going beyond bell's theorem, in "bell's theorem, quantum theory, and conceptions of the universe," m. kafakos, editor, vol. 37 of. *Fundamental Theories of Physics*, 1989.

[16] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[17] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[18] Tianyi Hao, Kun Liu, and Swamit Tannu. Enabling high performance debugging for variational quantum algorithms using compressed sensing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[19] Zhirui Hu, Peiyan Dong, Zhepeng Wang, Youzuo Lin, Yanzhi Wang, and Weiwen Jiang. Quantum neural network compression. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–9, 2022.

[20] Yipeng Huang and Margaret Martonosi. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 541–553, 2019.

[21] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[22] Richard Jozsa. Fidelity for mixed quantum states. *Journal of modern optics*, 41(12):2315–2323, 1994.

[23] Subhash C. Kak. Quantum neural computing. In Peter W. Hawkes, editor, *Advances in Imaging and Electron Physics*, volume 94. Elsevier, 1995.

[24] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia tools and applications*, 80:8091–8126, 2021.

[25] Irving Langmuir. Isomorphism, isosterism and covalence. *Journal of the American Chemical Society*, 41(10):1543–1559, 1919.

[26] Gushu Li, Yufei Ding, and Yuan Xie. Towards efficient superconducting quantum processor architecture design. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1031–1045, 2020.

[27] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. Projection-based runtime assertions for testing and debugging quantum programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[28] Ji Liu, Gregory T Byrd, and Huiyang Zhou. Quantum circuits for dynamic runtime assertions in quantum computation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1017–1030, 2020.

[29] Ji Liu and Huiyang Zhou. Systematic approaches for precise and approximate quantum state runtime assertion. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 179–193. IEEE, 2021.

[30] Cosmo Lupo and Seth Lloyd. Quantum-locked key distribution at nearly the classical capacity rate. *Physical review letters*, 113(16):160502, 2014.

[31] Ryan O'Donnell and John Wright. Efficient quantum tomography. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 899–912, 2016.

[32] Tirthak Patel, Abhay Potharaju, Baolin Li, Rohan Basu Roy, and Devesh Tiwari. Experimental evaluation of nisq quantum computers: Error measurement, characterization, and implications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15. IEEE, 2020.

[33] Tom Peham, Lukas Burgholzer, and Robert Wille. Equivalence checking paradigms in quantum circuit design: A case study. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 517–522, 2022.

[34] Yuxiang Peng, Mingsheng Ying, and Xiaodi Wu. Algebraic reasoning of quantum programs via non-idempotent kleene algebra. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 657–670, 2022.

[35] Wenhui Ren, Weikang Li, Shibo Xu, Ke Wang, Wenjie Jiang, Feitong Jin, Xuhao Zhu, Jiachen Chen, Zixuan Song, Pengfei Zhang, et al. Experimental quantum adversarial learning with programmable superconducting qubits. *Nature Computational Science*, 2(11):711–717, 2022.

[36] Rebecca Roelofs, Vaishaal Shankar, Benjamin Recht, Sara Fridovich-Keil, Moritz Hardt, John Miller, and Ludwig Schmidt. A meta-analysis of overfitting in machine learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[37] Joschka Roffe. Quantum error correction: an introductory guide. *Contemporary Physics*, 60(3):226–245, jul 2019.

[38] Stack Exchange Inc. Stack Overflow - Where Developers Learn, Share, & Build Careers, 2023.

[39] Runzhou Tao, Yunong Shi, Jianan Yao, John Hui, Frederic T Chong, and Ronghui Gu. Gleipnir: toward practical error analysis for quantum programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 48–64, 2021.

[40] Rasit Onur Topaloglu. Quantum logic locking for security. *J*, 6(3):411–420, 2023.

[41] Giacomo Torlai, Christopher J Wood, Atithi Acharya, Giuseppe Carleo, Juan Carrasquilla, and Leandro Aolita. Quantum process tomography with unsupervised learning and tensor networks. *Nature Communications*, 14(1):2858, 2023.

[42] Laura Von Rueden, Sebastian Mayer, Katharina Beckh, Bogdan Georgiev, Sven Giesselbach, Raoul Heese, Birgit Kirsch, Julius Pfrommer, Annika Pick, Rajkumar Ramamurthy, et al. Informed machine learning–a taxonomy and survey of integrating prior knowledge into learning systems. *IEEE Transactions on Knowledge and Data Engineering*, 35(1):614–633, 2021.

[43] Chunhao Wang and Leonard Wossnig. A quantum algorithm for simulating non-sparse hamiltonians. *arXiv preprint arXiv:1803.08273*, 2018.

[44] Dong Wang and Jeremy Levitt. Automatic assume guarantee analysis for assertion-based formal verification. In *2005 Asia and South Pacific Design Automation Conference*, page 561–566, New York, NY, USA, 2005. Association for Computing Machinery.

[45] Hanrui Wang, Zirui Li, Jiaqi Gu, Yongshan Ding, David Z Pan, and Song Han. Qoc: quantum on-chip training with parameter shift and gradient pruning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 655–660, 2022.

[46] Jiyuan Wang, Fucheng Ma, and Yu Jiang. Poster: Fuzz testing of quantum program. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 466–469. IEEE, 2021.

[47] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. Quito: a coverage-guided test generator for quantum programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1237–1241. IEEE, 2021.

[48] Chun-Yu Wei, Yuan-Hung Tsai, Chiao-Shan Jhang, and Jie-Hong R Jiang. Accurate bdd-based unitary operator manipulation for scalable and robust quantum circuit verification. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, pages 523–528, 2022.

[49] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. Synthesizing quantum-circuit optimizers. *Proceedings of the ACM on Programming Languages (PLDI)*, 7:835–859, 2023.

[50] Shifan Xu, Connor T Hann, Ben Foxman, Steven M Girvin, and Yongshan Ding. Systems architecture for quantum random access memory. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 526–538, 2023.

[51] Yinyu Ye. On the complexity of approximating a kkt point of quadratic programming. *Mathematical programming*, 80(2):195–211, 1998.

[52] Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):1–49, 2012.

[53] Ed Younis, Koushik Sen, Katherine Yelick, and Costin Iancu. Qfast: Conflating search and numerical optimization for scalable quantum circuit synthesis. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 232–243. IEEE, 2021.

[54] Nengkun Yu and Jens Palsberg. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 542–558, 2021.

[55] Charles Yuan, Christopher McNally, and Michael Carbin. Twist: Sound reasoning for purity and entanglement in quantum programs. *Proceedings of the ACM on Programming Languages (POPL)*, 6:1–32, 2022.

[56] Zijun Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–2. Ieee, 2018.

[57] Li Zhou, Nengkun Yu, and Mingsheng Ying. An applied quantum hoare logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1149–1162, 2019.