QuCheck: A Property-based Testing Framework for Quantum Programs in Qiskit

GABRIEL PONTOLILLO, Department of Informatics, King's College London, United Kingdom
MOHAMMAD REZA MOUSAVI, Department of Informatics, King's College London, United Kingdom
MAREK GRZESIUK, Department of Informatics, King's College London, United Kingdom

Property-based testing has been previously proposed for quantum programs in Q# with QSharpCheck; however, this implementation was limited in functionality, lacked extensibility, and was evaluated on a narrow range of programs using a single property. To address these limitations, we propose QuCheck, an improved property-based testing framework for Qiskit. By leveraging Qiskit and the broader Python ecosystem, QuCheck facilitates property construction, introduces flexible input generators and assertions, and supports expressive preconditions. We assessed its effectiveness through mutation analysis on five quantum programs (2-10 qubits), varying the number of properties, inputs, and measurement shots to assess their impact on fault detection and demonstrate the effectiveness of property-based testing across a range of conditions. Results show a strong positive correlation between the mutation score (a measure of fault detection) and number of properties evaluated, with a moderate negative correlation between the false positive rate and number of measurement shots. The most thorough test configurations achieved a mean mutation score of 0.90 averaged across all five algorithms, with a false positive rate between 0 and 0.06. QuCheck identified 47.8% more faults than QSharpCheck, with execution time reduced by 67.4%. These findings highlight the viability of property-based testing for verifying quantum systems.

CCS Concepts: • **Computer systems organization → Quantum computing**; • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Property-based testing, Quantum Software Testing, Quantum program verification, Quantum computing, Mutation analysis, Qiskit

ACM Reference Format:
Gabriel Pontolillo, Mohammad Reza Mousavi, and Marek Grzesiuk. 2018. QuCheck: A Property-based Testing Framework for Quantum Programs in Qiskit. *J. ACM* 37, 4, Article 111 (August 2018), 33 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

1.1 Background and Motivation

Testing and debugging are a laborious and costly [Hailpern and Santhanam 2002; Kafle 2014], yet indispensable part of software development [National Institute of Standards & Technology 2002]. Designing effective tests is already challenging for classical systems; the challenge intensifies with quantum systems, where capturing the underlying principles and mapping the intricate input-output relationships are particularly difficult.

Authors' addresses: Gabriel Pontolillo, Department of Informatics, King's College London, Strand, London, United Kingdom, WC2R 2LS, gabriel.pontolillo@kcl.ac.uk; Mohammad Reza Mousavi, Department of Informatics, King's College London, Strand, London, United Kingdom, WC2R 2LS, mohammad.mousavi@kcl.ac.uk; Marek Grzesiuk, Department of Informatics, King's College London, Strand, London, United Kingdom, WC2R 2LS, marekgrzesiuk99@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 ACM.
ACM 0004-5411/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

Property-based testing [Claessen and Hughes 2000] is an approach to mitigate this problem; it has been successfully applied to various classical systems, including telecommunication software, vehicular communication stacks, and databases [Hughes 2016]. In 2020, an initial prototype called QSharpCheck [Honarvar et al. 2020] was developed for the Microsoft Q# language [Microsoft 2020] to extend property-based testing to quantum systems. QSharpCheck has attracted attention and has been utilised in subsequent research [Castro 2021; Hu et al. 2024]. However, this prototype has limited functionality: assertions are restricted to single qubit comparisons in the computational basis, it lacks support for generating multi-qubit states or oracle circuits, and precondition specifications are tied to input generation.

1.2 Contributions

In this paper, we present QuCheck, a redesigned property-based testing framework for the popular Qiskit platform [Javadi-Abhari et al. 2024]. It addresses the shortcomings of QSharpCheck and makes it available to the larger community of researchers and practitioners using Qiskit. Moreover, we design and carry out experiments to evaluate and compare the efficiency and effectiveness of our newly developed framework with its past incarnation, QSharpCheck. To summarise, our contributions are listed below:

(1) QuCheck, an improved property-based testing framework for Qiskit, extending previous property-based testing approaches for quantum programs through the introduction of statistical corrections, optimisation via circuit deduplication and greedy measurement insertion, support for variable size circuits, diverse and customisable input generators with the ability to generate input oracles, and assertions that compare multiple qubits and bases;
(2) Evaluation of the effectiveness of property-based testing for quantum programs;
(3) Analysis on how the number of properties, inputs and measurement shots relates to the effectiveness of property-based testing in the quantum computing context; and
(4) A qualitative and quantitative comparison of QuCheck with QSharpCheck.

1.3 Research Questions and Results

1.3.1 Research Questions. We pose the following research questions and answer them through carefully designed experiments to evaluate the effectiveness of property-based testing for quantum computing in general, and of QuCheck in particular, by comparing it with QSharpCheck.

- **RQ1:** How does the *thoroughness* of a property-based testing suite affect the ability to identify faults in quantum programs?
  - **RQ1.1:** How does the *number of properties per program* affect the ability to identify faults in the quantum programs?
  - **RQ1.2:** How does the *number of inputs per property* affect the ability to identify faults in quantum programs?
  - **RQ1.3:** How does the *number of measurement shots* affect the ability to identify faults in quantum programs?
- **RQ2:** Is property-based testing *effective* for the identification of faults in quantum programs?
- **RQ3:** How does QuCheck compare to QSharpCheck?
  - **RQ3.1:** What are the qualitative differences between QuCheck and QSharpCheck?
  - **RQ3.2:** How does QuCheck compare to QSharpCheck in identification of faults in quantum programs?

To determine property-based testing's *effectiveness* at identifying quantum program faults, we employed mutation score and execution time as our primary metrics. Additionally, the *thoroughness* of the suite is determined by the number of properties, inputs per property, and number of measurement shots, which we investigate through three sub-questions, labeled RQ1.1 to RQ1.3.

1.3.2 Results. Our experiments revealed that increasing the thoroughness of the test suite increases the ability to identify faults while reducing the false positive rate. Increasing the number of properties evaluated showed a strong positive correlation with fault detection (r = 0.579). Increasing the number of inputs showed a positive correlation with fault detection (r = 0.190). While increasing the number of measurement shots only weakly improved fault detection (r = 0.112), it was effective in reducing false positive rates (r = -0.235).

Overall, property-based testing proved to be an effective method for identifying faults in quantum programs, as evidenced by high mutation scores (0.90) across two heuristics-guided measurement configurations, low false positive rates (0 to 0.06), and feasible execution times.

QuCheck provides a more expressive and extensible testing framework than QSharpCheck that is able to detect more mutants, with a reduced execution time. This is supported by its ability to enable more customisable input generation, precondition checking, expressive assertions, and the execution of complex operations, such as those found in metamorphic properties. As a result, QuCheck identified 47.8% more faults than QSharpCheck with a 67.4% reduction in average execution time.

The artefact for QuCheck can be accessed at [Gabriel Pontolillo et al. 2024]. QuCheck is also distributed as a PyPI package [Gabriel Pontolillo et al. 2026].

1.4 Paper Structure

The remainder of this paper is organised as follows. Section 2 reviews related work on property-based testing in both classical and quantum contexts, contrasting them to other testing methodologies proposed for quantum programs. In Section 3, we describe property-based testing in more detail, outlining the advantages of its application in the context of quantum computing. In Section 4, we describe the property-based testing framework that we have developed, detailing its major components. In Section 5, we outline the experimental setup for each research question, including the quantum programs used to evaluate our testing framework. We present and analyse the experimental results in Section 6. Section 7 describes the threats to the validity of our work. Finally, in Section 8 the paper concludes with a brief discussion of the results, the impact of the work, its limitations and future directions.

2 RELATED WORK

There has been considerable progress in the field of testing for quantum programs, with multiple approaches proposed. Nevertheless, the fundamental challenges posed by quantum mechanics remain to be addressed. In this section, we review related work in three areas: classical property-based testing, property-based testing for quantum programs, and general testing approaches and tools in the quantum domain.

2.1 Property-based testing for classical programs

Property-based testing was initially introduced in the Haskell programming language through a tool named QuickCheck [Claessen and Hughes 2000], enabling the automated testing of program characteristics and invariants via the generation and execution of random test cases. Over time, this methodology has been ported across multiple programming languages, including Scala [Nilsson 2014], Prolog [Amaral et al. 2014], and Erlang [Hughes 2007]. It has also been adopted commercially,

with applications in software testing across the telecommunications [Arts et al. 2006], e-commerce, and automotive industries [Hughes 2016].

Hypothesis [MacIver et al. 2019], a widely used Python property-based testing library, like the above frameworks, is primarily designed for deterministic or reproducible behaviour (i.e., the same input always produces the same test outcome). In contrast, quantum circuits produce probabilistic measurement outcomes. To verify correctness in such settings, recent work [L. Huang et al. 2024; Y. Huang and Martonosi 2019; Muqeet et al. 2024; Paltenghi and Pradel 2023; X. Wang, Arcaini, et al. 2022] has employed statistical methods, which in turn produce probabilistic results. QuCheck addresses this gap by natively supporting statistical assertions and a statistical correction method for the multiple test problem (Section 4.5). In addition, QuCheck provides quantum-specific input generators such as oracle circuits and quantum states.

2.2 Property-based testing for quantum programs

Honarvar, Mousavi, and Nagarajan [Honarvar et al. 2020] proposed QSharpCheck, a property-based testing framework for Q#, allowing the specification of properties in a plain text file, which are parsed and executed by their tool, yielding a counterexample when a property does not hold. QSharpCheck's approach to specifying property inputs and preconditions is limited to defining the range of θ and ϕ for each individual qubit, denoting the polar and azimuthal angles of the qubit state on the Bloch sphere. This approach does not allow for the generation of entangled states, or more general inputs that may be required, such as oracle circuits. Furthermore, the statistical analysis performed does not apply a correction (such as the Holm-Bonferroni correction) for the multiple test problem to control for the family-wise error rate when performing sequential statistical tests.

In our previous work [Pontolillo and Mousavi 2022], we curated a set of benchmark quantum programs and compared the performance and popularity of Cirq, Qiskit, and Q#:

- Popularity was investigated by analysing quantum computing repositories on GitHub, finding Qiskit to be more prevalent in the top hundred results across all the tested sorting options.
- We identified a disparity in performance when running the algorithms between the different platforms, finding Q# to be least performant, citing our experimental results for the execution of the quantum phase estimation algorithm.

In [Pontolillo and Mousavi 2024], we investigated the application of property-based testing and delta-debugging for quantum programs, where a set of changes between a passing and failing quantum circuit are recursively tested to isolate a minimal set of changes that cause a failure. We applied a property-based test oracle, comprised of a set of three properties of the circuit under test to determine whether a tested subset of changes passes or fails. The oracles' ability to differentiate between correct and faulty circuits was demonstrated through its successful integration with delta debugging. However, the tests were implemented on an ad-hoc basis, and would have benefited from a rigorous property-based testing framework.

The findings mentioned above, the gaps identified in prior work, and the accessibility of Python motivated our decision to develop a property-based testing framework for quantum circuits in Qiskit. To the best of our knowledge, no other published property-based testing approaches currently exist for Qiskit.

2.3 Other testing techniques for quantum programs

Researchers have proposed various approaches for testing quantum programs, including:

- Search-based testing [X. Wang, Arcaini, et al. 2022; X. Wang, T. Yu, et al. 2022], which uses genetic, or evolutionary algorithms to generate test cases. The goal of these approaches differs from that of QuCheck, where the focus *is on efficiency*. Using a program specification

that describes the input-output relation of the programs: MutTG maximises the number of mutants killed with a fixed number of test cases, and QuSBT attempts to generate a minimal test suite that can kill all mutants from a set. In contrast, QuCheck *verifies the correctness* of a program using a set of invariants and logical rules of a quantum program encoded into properties.
- Fuzz testing [J. Wang et al. 2018], which analyses the source code to identify branches associated with measurements in the circuit, then dynamically executes the circuit, receiving feedback to generate test inputs with higher branch coverage than random testing. In contrast, property-based testing explores a wide range of inputs to verify specified program properties, rather than maximising for branch coverage directly, though with sufficient inputs, it will eventually traverse all branches.
- Combinatorial testing [X. Wang, Arcaini, et al. 2023] generates computational basis test suites of a specified strength, or until a failure is found. While combinatorial testing is effective for evaluating discrete inputs; QuCAT, unlike QuCheck, lacks the ability to generate continuous input states or states in alternative bases.

Static analysis techniques have been developed for quantum circuits [Paltenghi and Pradel 2024; P. Zhao et al. 2023]. These offer a resource-efficient way to detect faults before circuit execution, helping to address scalability challenges in quantum software development. However, certain faults will only manifest at run time, after circuit initialisation. In such cases, our approach, which executes the circuits, is well suited to detect these faults.

QuraTest [Ye et al. 2023] is an automated test case generation tool that leverages three test case generators, UCNOT, IQFT, and random, in order to generate inputs that are entangled, or have a phase. These inputs are evaluated based on input diversity, output coverage, and mutation score. On the other hand, QuCheck is a property-based testing framework for specifying and statistically verifying semantic properties, through the provision of various input generators, the ability to check preconditions on inputs, and statistical assertions. For the purposes of our experiments, we use a simulator and initialise qubits to state vectors provided by the random statevector generator. However, our framework also includes a replication of QuraTest's UCNOT input generator, that initialises a random state by applying U and CNOT gates, rather than by direct statevector initialisation.

Metamorphic testing [Abreu et al. 2022] has been proposed for oracle quantum programs, they define metamorphic relations using the properties of the program under test, demonstrate how to encode them into a quantum circuit. In contrast, while we implement some metamorphic properties for our case studies, they are not encoded into the quantum circuit. Instead, two executions of the circuit are performed with different inputs, and the outputs are compared.

Proq [Li et al. 2020] is a runtime assertion scheme that employs projection-based assertions to verify quantum states throughout program execution. Proq's assertions can verify quantum states more efficiently compared to statistical assertions, but they are less suitable for property-based testing, as each randomly generated input requires a different projection-based assertion (if it modifies the asserted subspace). However, this approach may be implemented for the verification of invariants, where the subspace is not modified by the input, thus allowing for a fixed projection-based assertion. In contrast, our approach incorporates statistical assertions in the spirit of Huang and Martonosi [Y. Huang and Martonosi 2019] and, for some cases, local-measurement assertions related to Yu [N. Yu 2020], which verify more efficiently but cannot distinguish all multi-qubit states.

Mutation testing tools such as Muskit [Mendiluze et al. 2021] and QMutPy [Fortunato et al. 2022] have been previously proposed and applied for the evaluation of test suites [Honarvar et al. 2020;

Long and J. Zhao 2024; Ye et al. 2023], they apply mutation operators that modify the underlying quantum circuits of a program, such as through the addition, removal or replacement of quantum gates at random locations in the circuit. In our experiments, we chose to use QMutPy due to its compatibility with the structure of our case study code. Specifically, QMutPy allows mutations to be applied to the code that generates quantum circuits, rather than directly to the circuits themselves.

To the best of our knowledge, none of the discussed methodologies consider oracles, sub-circuits, or unitary gates *as potential inputs to quantum programs.* QuCheck provides a significant level of customisability, allowing for various inputs such as the insertion of random sub-circuits as inputs into a larger quantum program.

3 PROPERTY-BASED TESTING FOR QUANTUM PROGRAMS

Property-based testing consists of the specification of properties of the program under test, comprised of inputs, preconditions, operations, and postconditions. The property-based testing methodology generates a diverse array of inputs that satisfy the preconditions. These inputs are then passed to the program (circuit), which is executed, and the postconditions are checked against the results. Importantly, this approach does not require the user to manually identify specific inputs that could cause failures. Instead, properties serve as abstract specifications, allowing for the automated exploration of a wide range of input scenarios.

The application of property-based testing to quantum programs allows for:

- **Handling diverse quantum inputs:** Quantum programs may receive oracle circuits or quantum states as input, comprised of multiple qubits in superposition or entanglement. property-based testing allows for the automatic exploration of these states and oracle circuits, without the necessity to manually specify them.
- **Catching edge cases:** Because of this input diversity, it may be difficult for a developer to manually identify edge cases that cause the program to fail. property-based testing mitigates this problem through the generation of inputs that may be missed by a developer when designing tests.
- **Flexible Testing for Parameterized Circuits:** Quantum circuits are often parameterized, where their size or structure depends on input parameters, such as an integer specifying the number of qubits or gates. Property-based testing enables testing such parameterized circuits by allowing properties to be evaluated against a range of circuits generated from different parameter values. This removes the need to create and maintain tests for each potential circuit size or structure (i.e. different numbers of Grover diffusion operators applied in a Grover's algorithm circuit).
- **Facilitation of Test-Driven Development:** In test driven development (TDD), developers first implement tests for their programs, then work on the implementation until they fail. However, it is challenging to construct test suites that catch all edge cases for quantum programs. Using an incomplete or incorrect suite of tests during TDD, leads to the development of an incomplete, or faulty program. Property-based testing avoids this scenario, as developers consider the broader semantics of the programs, and encode them into properties, rather than identify specific test cases.

4 QUCHECK

4.1 Architecture

The QuCheck architecture is based on four key components: property specification, test preparation, circuit preparation and execution, and test outcome evaluation, as shown in Figure 1.

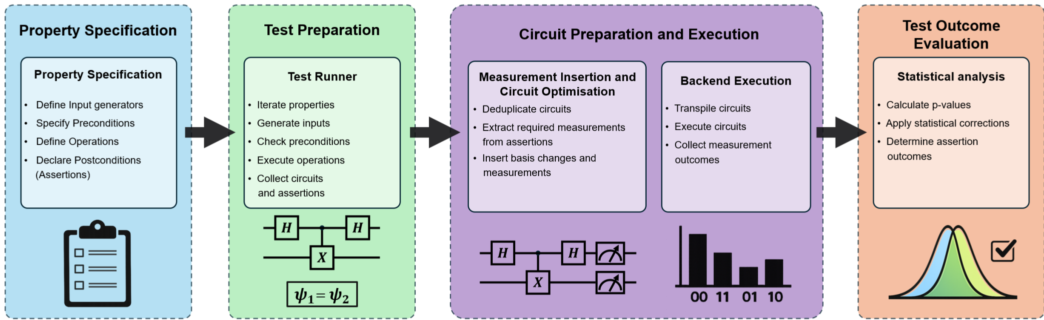

Fig. 1. Major components of the framework.

- **Property Specification:** Encompasses the tools provided by QuCheck that aid the specification of properties, such as input generation, precondition specification, and assertion specification.
- **Test Preparation:** Generates inputs using the provided input generators, evaluates them against preconditions, and executes the operations function defined in the property, recording assertions for later processing.
- **Circuit Preparation and Execution:** Handles circuit optimisation, inserts the required basis changes and measurements, transpiles and executes prepared circuits, and collects resulting measurement outcomes.
- **Test Outcome Evaluation:** Verifies both statistical and non-statistical assertions using the collected measurements.

4.2 Property Specification

Our framework streamlines the process of specifying and implementing property-based tests by providing tools and helper functions to aid in converting conceptual properties into executable code.

In QuCheck, properties are specified using four core elements, which inherit from a base property class provided by the framework. These elements include:

- **Input generation** through the selection of a set of deterministic input generators that are needed for the property.
- **Precondition verification** by checking the inputs that have been generated through the generators.
- **Operation execution** using the verified inputs for initialisation or inserting oracles.
- **Postcondition specification** by inserting assertions in the operation body.

4.2.1 Property Example. An example of a property that has been specified using QuCheck can be seen in Listing 1. This property tests quantum teleportation by verifying that the state of the qubit in the target register after teleportation matches a qubit initialised to the same state. The `get_input_generators` function receives a list of input generators. In this case, `RandomState` is a wrapper around Qiskit's `random_statevector` function, which samples from the uniform distribution induced by the Haar measure, meaning that every pure state is equally likely to be drawn. The parameter passed to the `RandomState` input generator indicates the size of the state to generate, in qubits.

No preconditions are required for this property, therefore, this preconditions function always returns True. An example usage of the precondition function is in the properties used to test Grover's algorithm, where the randomly generated Grover's oracle is checked to ensure that more (or less) than half of all possible states are marked.

In the operations function, two circuits are initialised with the randomly generated state on register zero: qc and qc2. The quantum teleportation circuit is appended into qc, while qc2 remains in its initialised state. The final step is to use assert_equal to compare the second register of qc (which underwent teleportation) with register zero of qc2 (the unaltered initialization).

```
class TeleportationOutputEqualToInput(Property):
    # specify the inputs that are to be generated
    def get_input_generators(self):
        state = RandomState(1)
        return [state]

    # specify the preconditions for the test
    def preconditions(self, q0):
        return True

    # specify the operations to be performed on the input
    def operations(self, q0):
        qc = QuantumCircuit(3)
        qc.initialize(q0, [0])
        qt = quantum_teleportation()

        # stitch qc and quantum_teleportation together
        qc = qc.compose(qt)

        # initialise qubit to compare to:
        qc2 = QuantumCircuit(1)
        qc2.initialize(q0, [0])
        self.statistical_analysis.assert_equal(qc, [2], qc2, [0])
```

Listing 1. Code example for a quantum teleportation property in QuCheck

4.2.2 Input Generators. Properties require the specification of a range of inputs, which must be passed to the precondition check and body of the operation. To streamline this, we provide a predefined set of configurable, deterministic input generators, relevant for the formulation of common types of inputs to properties. Additionally, an interface is provided to allow user-defined input generators, which can also be created using a composite of predefined generators.

These input generators must be deterministic to ensure consistent input generation and enable developers to reproduce failing test cases during debugging. In QuCheck, this is achieved by requiring the input generators to use a seed, and ensuring that the same input is produced for each identical seed.

4.2.3 Precondition. The generated inputs are then passed to the precondition check, which verifies whether the randomly generated inputs satisfy the preconditions. QuCheck will attempt to generate an input that satisfies the precondition a configurable number of times, after which the property will time out and fail. Where possible, input generators should be constrained to produce only valid inputs, as this avoids the overhead of repeatedly generating, checking, and discarding inputs that do not satisfy the preconditions.

4.2.4 Operation. The purpose of the operation function is to initialise the relevant circuits to evaluate, and specify the assertions that need to be verified.

First, the operations function within a property receives the verified inputs and uses them to set up the circuit or sub-circuit to be executed. For metamorphic properties, this function may construct two or more quantum circuits to compare their outcomes, avoiding the need for an oracle. The postconditions are verified by assertions defined within the operations function and are recorded for later processing.

4.3 Test Preparation

The test preparation component passes the verified inputs into the operation function of the property, which typically instantiates and initialises the circuit under test and specifies the assertions needed to verify the postcondition. After this process has completed for all properties, the assertions are collected for subsequent circuit execution and outcome evaluation.

4.4 Circuit Preparation and Execution

The circuit preparation and execution component processes the collected assertions, inserts the required basis changes and measurements, applies circuit optimisations, and prepares the executable circuits. These circuits are then transpiled, executed on the selected backend, and their measurement outcomes are collected for evaluation.

4.4.1 Measurement optimisation. Multiple properties are evaluated in a property-based test suite, and some may require the evaluation of identical quantum circuits. In such cases, duplicate measurements (i.e., same basis and qubit registers on the same circuit) need to be performed only once, we describe the process below:

Before performing any measurements, the framework collects all assertions by executing each properties' operation method, from which the quantum circuits, measurement registers, and basis to measure are extracted and stored. It then identifies and removes duplicate circuits and determines what measurements need to be performed. For each unique circuit, a copy is first made. The list of measurements (and basis changes) to attach is then iterated through, adding each one to the circuit if it does not conflict with an already inserted measurement. If a conflict occurs, the measurement is added to a new, or existing copy of the circuit where the conflict does not occur. Finally, a pass is done to verify that no duplicate circuits are present after the basis changes and measurements are inserted.

It should be noted that when the same set of input generators is passed to multiple properties, the test runner will attempt to reuse the same seeds, resulting in identical inputs across these properties. This may cause identical circuits to be executed (if preconditions fail, additional seeds are generated), which are then deduplicated and optimised as described above, allowing measurements to be reused for the verification of properties.

4.5 Test Outcome Evaluation

The test outcome evaluation component iterates through the collected assertions and evaluates them using the measurement outcomes obtained during circuit execution. For statistical assertions, a set of p-values is calculated for each generated input and property pair. A statistical correction is then applied to account for the multiple testing problem before determining the final assertion outcomes.

4.5.1 Assertions. Statistical assertions have been previously proposed [Y. Huang and Martonosi 2019] for the verification and validation of quantum states, these have been thus applied in [Honarvar et al. 2020]. QuCheck contains a suite of predefined assertions that can be applied within the operation function that must be specified by each property. The common feature shared between

all of the predefined assertions is the need to specify at least one quantum circuit, and a list of its qubit indexes that need to be measured.

Since performing full tomography on a set of qubits is computationally intensive, the predefined assertions within QuCheck perform measurements on the X, Y, Z basis for each qubit in the quantum state, comparing the marginal distribution of outcomes of one qubit at a time. This removes the assertions' ability to distinguish between different entangled states, yet drastically lowering the computational overhead of gathering measurement results for the assertions. As we will demonstrate in this paper, this is an effective method for detecting faults.

- **AssertEqual:** Receives two quantum circuits and two sets of qubit indices to compare with each other, as well as optionally the basis to compare, which defaults to X, Y, Z, but can be limited if necessary.
  Fisher's exact test is applied to the contingency tables constructed outcome distribution of each qubit and basis to determine the equality of the qubit's state. If any p-value returned from the test is below the adjusted significance level (from the statistical correction), the null hypothesis that the states are equal is rejected.
- **AssertDifferent:** The same as AssertEqual, except when verifying the p-values from the statistical tests, rather than rejecting the null hypothesis if any p-value is below the significance level, it is rejected if all p-values are above the significance level.
- **AssertEntangled:** This assertion checks for entanglement within a circuit in a given basis between a provided set of qubits, assuming that the subset is maximally entangled. This is implemented by verifying whether the outcomes of the measured qubits consist of two complementary outcomes, such as $|000\rangle$ and $|111\rangle$, or $|101\rangle$ and $|010\rangle$. This approach scales linearly with number of qubits, but is limited to detecting entanglement within a single basis. It assumes that the subset of qubits passed to the assertion is maximally entangled in that basis.
- **AssertSeparable:** Checks for entanglement between the specified set of qubits and basis, then negates the result.
- **AssertProbability:** For a given basis, asserts that the probability of measuring $|0\rangle$ for a set of qubits matches with a set of probabilities.
- **AssertMostFrequent:** No statistical test is applied for this assertion, we check that the most frequent outcome when measuring a provided quantum circuit is equal to the outcome provided.

Listing 1 provides an example of using AssertEqual to verify that the states of a given set of qubits are equal in two quantum circuits.

4.5.2 Statistical correction. Property-based testing generates multiple test cases, each of which calls a statistical assertion to verify its post-condition. A Type I error (false positive, reporting a property violation when the property actually holds) is more likely when multiple statistical tests are applied consecutively. To address this, the family-wise error rate is controlled through a multiple testing correction such as the Holm-Bonferroni correction [Holm 1979].

To apply this correction, the p-values from each statistical test need to be collected. Accordingly, a fixed number of inputs are generated, tested as a batch, and evaluated simultaneously. Unlike QSharpCheck, this approach does not test each input individually until a property fails (or a set number of inputs has been tested). This is discussed further in Section 4.6.

4.6 Limitations

One limitation of this approach is the inability adaptively to control the number of inputs based on the test results. A fixed number of inputs must first be specified and tested, as the assertion check

is only performed after all the test cases have been evaluated. Practically, this may lead to more circuits being executed and measured than necessary to verify a given property. This design choice was made to allow for the application of a statistical correction, which requires knowledge of all the statistical tests that are performed.

Several issues arise from property-based testing's input generation approach compared to fixed test case methods. For some programs, specific, carefully chosen inputs are necessary to validate program behaviour under known conditions. Furthermore, exploring the input space through random generation is more resource-intensive than using optimised, pre-planned input sets that maximise coverage and adequacy metrics with minimal inputs. Constructing input generators for complex inputs, such as the oracles or sub-circuits within quantum circuits, requires significant domain knowledge and effort.

As shown in our experiments, defining an adequate suite of properties for complex systems is essential for property-based testing to be effective. However, this is not a trivial task, as it requires thorough knowledge of the program under test to design multiple properties whose combined coverage adequately represents the program's semantics.

Verifying postconditions for quantum programs remains challenging due to the trade-off between accuracy and performance. Various approaches can be used to implement assertions for this purpose, ranging from full tomography, which is accurate but resource intensive for large quantum systems, to evaluating the individual qubits outcomes on a limited number of bases. Other techniques, such as classical shadows [H. Huang et al. 2020], could be used to improve the efficiency of some assertions (i.e., AssertEqual), although they may not be efficiently applicable to all assertion types and quantum states.

5 EXPERIMENT DESIGN

Three sets of experiments were performed to address our research questions. The first set evaluated the effect of property-based testing under varying configurations of *thoroughness*. The second set evaluated the *effectiveness* of QuCheck against a unit test baseline. The third set involved a quantitative comparison between QuCheck and QSharpCheck, where two case studies from the original experiment were repeated, with newly translated mutants. Below, we provide details on the mutation analysis used in our experiments, which was used to measure the effectiveness of the testing techniques. Furthermore, we describe the independent variables used to define the thoroughness of the test suite, and the quantum programs used as our subject systems.

5.1 Mutation Analysis

Mutation analysis is employed to assess the effectiveness of a testing suite. The general technique is to apply small changes to a correct implementation of a program to create a set of "mutants". The test suite to be evaluated is then applied to each mutant, and the test outcome is recorded. The effectiveness of the test suite is subsequently determined by calculating the mutation score, which is the percentage of mutations that failed the tests over the total number of mutations. However, achieving 100% mutation score is unlikely with a large set of mutants, as some mutant versions of the program may be semantically equivalent to the original program (known as equivalent mutants), making them impossible to be detected by a test suite.

QMutPy was used in the experiments to generate twenty mutants for each of the five quantum programs in our case studies. However, some mutants generated by the tool were syntactically incorrect. In such cases, new mutants were generated until no more syntactic errors occurred.

To measure the *false positive rate* of the property-based tests, a separate set of ten equivalent mutants was intentionally created. This was done by randomly selecting a set of gate identities,

inserting them into the case study implementation of the circuit. Once the mutants were generated, QuCheck was applied to them, the outcomes were recorded, and are listed in Section 6.

5.2 Subject Systems

For the evaluation of QuCheck, a total of thirty mutations of each of the five diverse quantum programs were tested, comprising the aforementioned twenty QMutPy-generated mutants and ten equivalent mutants. These subject systems were chosen for their variation in gate depth, circuit width, and usage of oracles. Specifically, Deutsch Jozsa and Grover's algorithms were included to evaluate how QuCheck handles programs that receive oracles as input. Quantum phase estimation was selected for its more complex inputs involving a unitary operator and a state vector. Additionally, Quantum Fourier Transform and teleportation were also included as simpler algorithms to test. Finally, each algorithm except quantum teleportation is configurable, allowing their width and depth to vary depending on their inputs. For example, Listing 2 generates a random oracle for Grover's algorithm with a width between 4 and 7 qubits, which is then inserted into a Grover circuit of the corresponding size. This enables us to evaluate how well property-based testing performs for algorithms that scale in size, rather than only fixed circuit instances of each program.

- **Quantum Fourier Transform (QFT):** The Quantum Fourier transform algorithm efficiently applies the Fourier transform to the amplitudes of any inputted state. The algorithm is applied within many other programs, such as Shor's algorithm and quantum phase estimation.
- **Quantum Phase Estimation (QPE):** QPE estimates the value of φ when given a unitary operator U, and an eigenvector of the unitary $|u\rangle$ that has an eigenvalue of $e^{2\pi i\varphi}$. A well known application is its use within Shor's algorithm to find the order r of $a^r \equiv 1 \bmod N$, where a is co-prime with N, which can be then be used to efficiently find the factors of a large number N.
- **Quantum Teleportation (QT):** Quantum teleportation transfers the quantum state of a single qubit $|\psi\rangle$ to another qubit. This process requires the transmission of one qubit from an entangled pair, as well as a means to transfer two bits of classical information, and has been experimentally realised with an average fidelity of 0.80 ± 0.01 over a distance of up to 1400 kilometers [Ren et al. 2017]. Quantum teleportation and its derivatives have applications in quantum networks [Hermans et al. 2022], allowing for the reliable transfer of quantum information over long distances.
- **Grover's algorithm (GR):** Grover's algorithm, often called the quantum search algorithm, can be applied to find the inputs, when given the solution to a function (find x where $f(x) = 1$). The algorithm repeatedly increases the amplitudes of the indexes (input values) that satisfy the given black box function by repeating the application of the Grover diffusion operator before performing measurements. This algorithm can provide a quadratic speedup to NP-complete problems [Fürer 2008; Nielsen and Chuang 2016; Raj and Shivakumar 2006].
- **Deutsch Jozsa's algorithm (DJ):** The Deutsch Jozsa algorithm is able to determine whether a black box oracle applies either a constant or balanced function to the lower register using only a single query.

5.3 Properties used in experiments

For each subject program we define a small set of properties covering different semantic aspects. Table 1 lists the property names and a one-line description; implementations of these properties using QuCheck can be found within the artefact.

5.3.1 Property Design Procedure. Identifying suitable properties is a recognised challenge in classical property-based testing, and this difficulty persists for quantum programs. To structure this

Table 1. Properties defined for each quantum program.

Program	Property	Semantic Aspect	Property Pattern
QFT	Linear Shift to Phase	A linear shift to the statevector causes a specific phase shift in the output after QFT	Output
	Phase Shift to Linear	A shift in phase to the statevector causes a linear shift in the output after inverse QFT	Output
	QFT–H Equivalence (Zero Input)	Starting from $\|0^n\rangle$, applying QFT then Hadamards behaves like no operation; appending the same unitary to both circuits yields identical outcomes.	Metamorphic
QPE	Eigenstate Preservation	If the system register starts in an eigenvector of U, QPE leaves that register unchanged	Invariant
	Exact Phase Recovery	When the eigenphase is exactly representable with the available estimation qubits, the estimation register yields the exact eigenphase	Output
	Eigen vs Non-Eigen Input	Replacing an eigenstate with a non-eigen state changes the QPE estimated phase output	Equivalence
QT	State Preservation	After teleportation, Bob's output qubit reproduces the input state $\|\psi\rangle$	Output
	Measurement Registers Reset	Measurement register in $\|++\rangle$ after teleportation	Invariant
	Metamorphic Consistency	For any unitary U, teleportation of $U\|\psi\rangle$ is equivalent to teleportation of $\|\psi\rangle$ followed by U on Bob's qubit	Metamorphic
GR	Workspace $\|-\rangle$ Preservation	The workspace qubit remains in $\|-\rangle$ after the algorithm.	Invariant
	Marked most frequent	With at most half of the data states marked, the most frequent outcome on the data register lies in the marked set.	Equivalence
	Complement set most frequent	If more than half of the data states are marked, the most frequent outcome on the data register lies outside the marked set.	Equivalence
DJ	Constant oracle to All-Zero Output	For a constant oracle, the data register is $\|0^{\otimes n}\rangle$.	Output
	Balanced oracle to Non-Zero Output	For a balanced oracle, the data register output is not $\|0^{\otimes n}\rangle$.	Output
	Workspace $\|-\rangle$ Preservation	The workspace qubit remains in $\|-\rangle$ after the algorithm.	Invariant

process, we organise properties into a set of property patterns. In property-based testing, properties are generally program-specific rather than universal, capturing the expected behaviour of the particular program under test. These properties may be derived from externally visible behaviour, but can also leverage implementation-specific knowledge when such knowledge is available. Across our case studies, properties fall into four classes: invariants, output-form properties, relational

or metamorphic properties, and equivalence-class or boundary-value properties. The taxonomy below serves as a guideline for deriving properties for new quantum programs and is the procedure we used to design the properties in Table 1. Accordingly, the same process can be used to extend QuCheck to additional quantum programs by identifying expected behaviours, selecting suitable property patterns, and expressing them as concrete properties through generators, preconditions, operations, and assertions.

Program Analysis: Identifying properties requires a sufficient understanding of correct behaviour, so that it can then be formally specified. To that end, we ask questions such as:

- What is the expected state at key points in the circuit?
- What relationships hold between inputs and outputs?
- How are multiple inputs/executions related, and how should that affect the outputs?
- Does behaviour differ across input ranges/classes? Where are the boundaries?
- What behaviours never change (e.g., ancillas/workspace remain unmodified)?

The aim is to maintain a level of abstraction and generality; specific inputs are of less interest than general trends across input ranges. However, a limited simulation may be conducted with concrete inputs to sanity test the list of observations made.

Observation to Specification: Once observations about a program's behaviour are made, they must be turned into concrete, checkable specifications. In the following, we identify four patterns of properties as guidelines when converting observations into full property specifications for each program.

- **Invariants.** Behaviours that remain unchanged across correct executions (e.g., workspace or ancilla preservation).
- **Output form.** The expected form or structure of (part of) the output under correct execution for given inputs (e.g., prescribed outcome states).
- **Relational or metamorphic effects.** Relationships between multiple executions or inputs and how they affect the output (e.g., equivalences between applying a transformation before or after teleportation).
- **Equivalence classes and boundary values.** Effects of different input classes on behaviour (e.g., eigenstate vs. non-eigenstate, constant vs. balanced oracle) or effects of inputs at boundaries between classes.

Each observation is then mapped to a property pattern, and formalised in the format of input, precondition, operation, postcondition introduced in Section 4.2. In this step, it can be helpful to begin by identifying the postcondition to be asserted, determine what must be measured, and work backwards.

Properties may also be derived from previously established specifications or proofs. These can then be used to empirically verify specific program instances, helping to identify implementation errors or integration issues (e.g., transpilation). Section 5.6.4 details how specific properties were selected for experimentation.

5.4 Procedure

For the rest of this section, we define the experimental procedure and rationale to answer each research question:

- **RQ1:** How does the *thoroughness* of a property-based testing suite affect the ability to identify faults in the quantum programs?
- **RQ2:** Is property-based testing *effective* for the identification of faults in quantum programs?
- **RQ3:** How does QuCheck compare to QSharpCheck?

5.5 Experimental setup for RQ1

To answer **RQ1**, we define the metrics used to evaluate the effectiveness of the property-based testing suite in detecting faults, as well as methods to adjust its thoroughness for comparison.

- **Effectiveness:** The mutation score was the primary metric used to evaluate effectiveness, although it is assessed differently depending the set of mutants being tested. Two sets of mutants were used in the evaluation. The first consisted of intentionally equivalent mutants, created by inserting circuit identities at random locations within the quantum circuits. The second was generated with QMutPy and manually inspected to remove obviously equivalent mutants. For the former, the mutation score indicates the rate of false positives, which need to be minimised; conversely, for the semantically different set of mutants, the mutation score indicates the ability to detect potential faults. The execution time of the property-based tests for each configuration was also employed to evaluate the cost-benefit trade-off in increasing the test suite thoroughness.
- **Thoroughness:** The thoroughness of the test suite was varied by adjusting three variables: the number of properties considered in the property-based tests, the number of inputs generated for each property, and the number of measurement shots per basis. All combinations of these independent variables (Table. 2) were used to evaluate the technique. We randomly sampled the properties for each run when evaluating fewer than 3 properties (e.g., 1 or 2 out of 3). This minimises the influence of any single property, ensuring that the results reflect the general benefit of using more properties rather than the specific strength of individual ones.

Number of Properties	1	2	3						
Number of Inputs	1	2	4	8	16	32	64		
Number of Shots	12	25	50	100	200	400	800	1600	3200

Table 2. Independent variables considered.

QuCheck was configured to execute each of these configurations against every mutant of the program being tested. A mutant was deemed "killed" if any of the property-based tests failed during execution. Figure 2 illustrates this experimental process, demonstrating the flow from configuration generation through mutant testing to result analysis.

For the evaluation of the relationship between the thoroughness and effectiveness of the test suite (**RQ1.1** to **RQ1.3**), we systematically tested all possible combinations of these variables from Table 2, henceforth referred to as configurations. Unless stated otherwise (i.e., Figure 5), data from all test configurations described above are used in our analysis, grouped by the independent variable in the x-axis.

5.6 Experimental setup for RQ2

To determine the overall effectiveness of property-based testing for quantum programs, we establish principled values for the three main experimental parameters: number of measurement shots, number of input circuits, and number of properties. The same experimental setup from RQ1 is used, but here we additionally compare the results of property-based testing against a conventional unit testing suite as a baseline to contextualise its relative effectiveness. The following subsections describe the proposed baseline and the heuristics used to select independent variables.

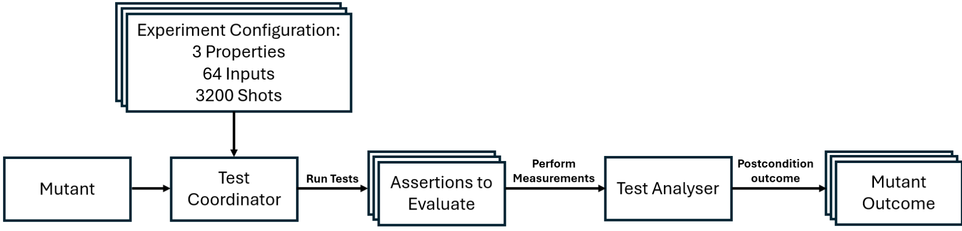

Fig. 2. Experimental setup.

5.6.1 Comparison Baseline. Given the range of proposed approaches for testing quantum programs, we established the following criteria to guide the selection of a fair and practical baseline: the test code needed to be publicly available, compatible with our programs and mutants, and the oracles could not depend on direct statevector information to enable a fair comparison. These criteria led us to use the MITRE QSFE repository [jclapis 2019], which provides unit tests compatible with four subject systems and required only minor modifications to integrate with our code and mutants.

5.6.2 Measurement Shot Heuristic. Half of the assertions (AssertEqual, AssertDifferent, and AssertProbability) observe the marginal distributions of qubits, and are also the set of assertions that were most frequently used in our property-based test suite.

The outcome of repeated projective measurements on a two-level quantum system can be modelled as a binomial distribution between 0 and 1 [Itano et al. 1993; Song and Cai 2025].

$$P(k \mid n, p) = \binom{n}{k} p^k (1-p)^{n-k},$$

Where n is the number of measurement shots, k is the number of times the outcome '1' is observed, and p is the probability of obtaining outcome '1'.

Wald's method provides an effective means to estimate the confidence interval of an observed binomial distribution [Agresti and Coull 1998].

$$\hat{p} \ \pm \ z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}},$$

Where $\hat{p} = \frac{k}{n}$ is the observed proportion of '1' outcomes, n is the number of shots, and z is the $(1 - \frac{\alpha}{2})$ quantile of the standard normal distribution.

To plan the number of measurement shots for our experiments, we will substitute the worst-case variance $p_0 = \frac{1}{2}$, and rearrange for n:
The margin of error is:

$$\varepsilon \ = \ z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}.$$

Assuming the worst-case variance, substitute $p_0 = \frac{1}{2}$:

$$\varepsilon \ = \ z\sqrt{\frac{1}{4n}} \ = \ \frac{z}{2\sqrt{n}}.$$

Solving for n gives the shot count:

$$n \ = \ \frac{z^2}{4\,\varepsilon^2}.$$

For our experiments, we target a margin of error of 2% ($\varepsilon = 0.02$), and 95% and 99%, confidence intervals ($z = 1.960, 2.576$).

$$n = \frac{1.960^2}{4(0.02)^2} \approx 2401 \quad (95\%\ \text{CI}), \qquad n = \frac{2.576^2}{4(0.02)^2} \approx 4147 \quad (99\%\ \text{CI}).$$

For simplicity, we round up to the next hundred, yielding 2500 and 4200 shots. Although not all assertions directly reduce to marginal distributions (e.g., AssertMostFrequent), we standardise on these shot counts for all assertions. This provides a conservative and uniform budget for the test suite.

5.6.3 Input Generation Budget. Although property-based tests can run indefinitely, projects typically cap the number of inputs per property, based on program runtime, available resources, and the program's criticality. Our previous work [Pontolillo and Mousavi 2024], which applied property-based test oracles within a delta debugging context, did not see a statistically significant impact. Establishing a fair baseline is therefore challenging; we decided to follow the original QuickCheck study [Claessen and Hughes 2000], which had a default cap of 100 inputs.

5.6.4 Property Selection. For each quantum algorithm, we identified multiple candidate properties and selected a subset that captures distinct aspects of the program's behaviour. The selection was guided by the property groups defined in Section 5.3.1, with the aim of representing at least two groups per algorithm. This approach sought to maximise behavioural coverage through the defined properties, and properties were further chosen to generate a broad range of structured inputs (states, unitaries, oracles) so that, collectively, they exercise all qubits and registers.

For instance, in the teleportation algorithm, the three chosen properties collectively cover *three property groups* and *all output qubits*: the direct output expected to contain the state $|\psi\rangle$ (output), the measurement register qubits (invariant), and the application of a unitary before or after teleportation (metamorphic). To ensure comparability across our programs and to constrain the experimental budget, we fixed the number of properties per algorithm to three. This cap is a pragmatic choice rather than an optimal one. As with other testing approaches, determining when a quantum program is sufficiently tested remains an open and context-dependent question.

5.7 Experimental setup for RQ3

To the best of our knowledge, the only other published property-based test framework for quantum programs is QSharpCheck [Honarvar et al. 2020]. The process for determining the qualitative differences (**RQ3.1**) between QSharpCheck and QuCheck is described in section 6.3.1, the following defines the experimental setup to evaluate the differences in effectiveness and execution time between the frameworks (**RQ3.2**). To perform this evaluation, we utilise the largest two configurations of independent variables as in **RQ2**, except testing only one property. Since QSharpCheck does not have the ability to apply multi-basis assertions, we control for the **total number of measurement shots** performed in each assertion by testing QSharpCheck with three times the number of shots per basis (i.e., 12600 instead of 4200).

5.7.1 Conversion of mutants: To ensure the fairness of the comparison, the same mutants that were generated for the evaluation of QuCheck (**RQ1-2**) were also used to compare QSharpCheck. Since QSharpCheck's framework's programming language is different, we needed to translate the Qiskit mutants into Q#. This was achieved by first converting each Qiskit mutant into QASM 2.0 directly through Qiskit's API, then using Quantastica's QConvert to translate from QASM 2.0 into Q# 0.10. This automatic translation often produced syntactically incorrect code, which was manually refactored for its execution in QSharpCheck.

5.7.2 Conversion of properties and assertions: Unfortunately, due to the lack of features, it was not possible to convert the majority of the QuCheck properties into QSharpCheck properties. For example, in QuCheck, AssertEqual is available for multiple qubits, while QSharpCheck limits AssertEqual to single qubits. We also encountered failures when attempting to use a portion of the provided assertions, preventing us from replicating many properties. We expect changes to the Q# compiler may have contributed to these.

5.7.3 Conversion of configurations: QSharpCheck's statistical analysis configuration requires three variables: inputs, experiments, and shots. The total measurements per input is equal to the number of experiments multiplied by shots. For example, to perform a total number of 12600 measurements, we set experiments to 90 and shots to 140.

5.8 System configuration

The experiments were performed in Qiskit 1.0.1 using a Windows 11 machine (R7 5700X, RTX 3060ti, 32GB RAM).

6 RESULTS

6.1 RQ1: Impact of test suite thoroughness on effectiveness

Figure 3 shows six heatmaps that illustrate the effects of increasing property count, input size, and number of measurement shots on the average mutation scores across all algorithms. The top row corresponds to QMutPy-generated mutants, while the bottom row represents equivalent mutants. The columns, from left to right, depict an increasing number of properties, from one to three.

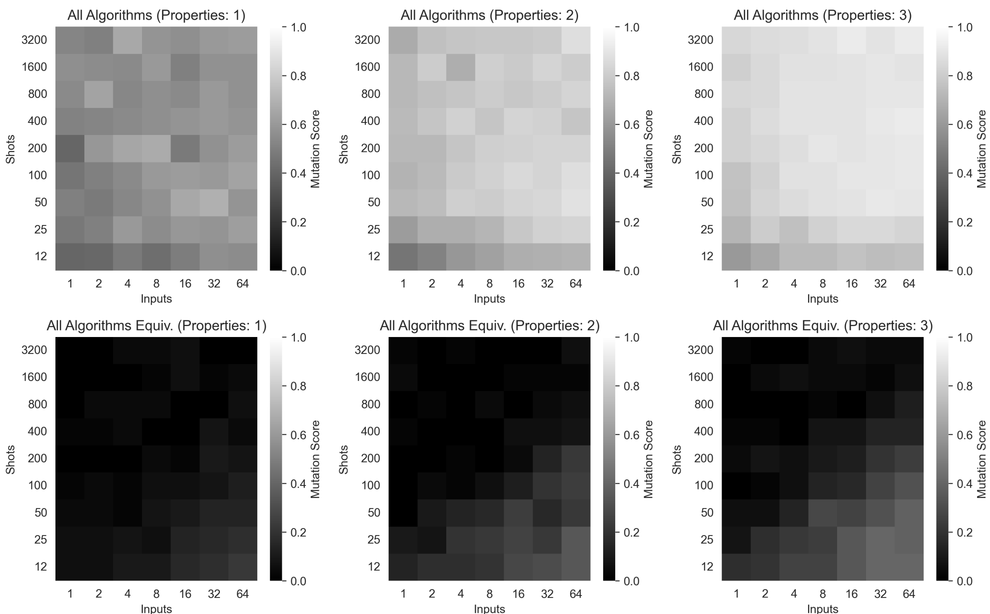

Fig. 3. Heatmaps illustrating the impact of property count, number of inputs, and measurement shots on mutation scores. The top row represents the average mutation scores for QMutPy-generated mutants across all algorithms, and the bottom row, the average mutation scores for equivalent mutants.

Two key patterns emerge from these heatmaps. First, for **QMutPy mutants** (top row), there is a clear trend of higher mutation scores in the top right region of each heatmap, indicating that a larger number of inputs and property counts lead to an increase in the mutation score. This trend is more pronounced when three properties are evaluated. This increase in mutation score for the QMutPy mutants is supported by a statistically significant positive correlation observed in the individual analysis of each independent variable, as investigated in RQ1.1, RQ1.2 and RQ1.3.

In contrast, for **equivalent mutants** (bottom row), a different overall pattern emerged. Lower mutation scores are concentrated at higher number of measurement shots, while still increasing with the number of inputs and properties evaluated. Although a positive correlation with the number of inputs and properties remains, *a stronger negative correlation* with the number of measurement shots was observed. Furthermore, when restricting the number of properties in Figure 5 to utilise the highest three configurations of measurements, *this positive correlation disappears*.

Answer to RQ1: How does the *thoroughness* of a property-based testing suite affect the ability to identify faults in the quantum programs?

Increasing the thoroughness of the property-based testing suite **increases the mutation score for the QMutPy mutants**, enhancing the ability to identify faults in quantum programs. At the same time, increasing the number of measurements performed per input **decreases the mutation score for the set of equivalent mutants**, thus reducing the likelihood of false positive results.

In the subsections below, we answer the sub-questions (RQ1.1-RQ1.3), illustrating the impact of each independent variable on the mutation score of each algorithm. In Figures 4-7, the dotted lines represent the mean values for each individual algorithm, stratified by the independent variable plotted on the x-axis. The solid red line represents the mean value over all algorithms and measurement configurations, with the vertical bars representing the 95% confidence interval of the true population mean. The Spearman's rank correlation coefficient was used to assess the relationship between the variables, which is suited to the monotonic (and sometimes non-linear) trend observed.

6.1.1 RQ1.1: Effect of Property Count per Program. The mutation score of mutants generated through **QMutPy** showed a strong positive correlation ($r = 0.579$, $p = 1.62\text{e-}85$), indicating a statistically significant relationship between the number of properties evaluated in the test suite and the mutation score. This trend can be seen in Figure 4. The median mutation scores for tests evaluating 1, 2 and 3 properties were 0.55, 0.80, and 0.85, respectively, with corresponding standard deviations of 0.195, 0.165, and 0.144. The reduction in standard deviation highlights the increased consistency of the mutation score as the number of properties evaluated in the test suite (thus program semantics covered by tests) also increases.

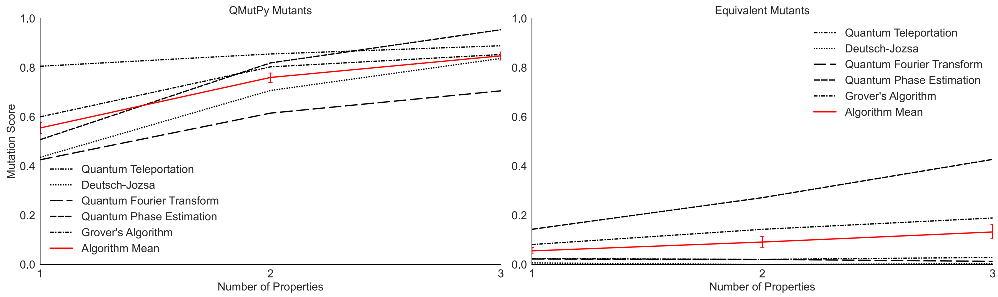

Fig. 4. Effect of *number of properties* on mutation score.

A weak positive correlation ($r = 0.078$, $p = 0.017$) was observed between the number of properties and the mutation score for **equivalent mutants**, indicating a slight increase in false positive results when more properties were evaluated. Among the five algorithms studied, *quantum phase estimation* showed the most pronounced increase in false positives as properties increased. The overall trend of rising false positives is primarily caused by the insufficient measurement shots in test configurations. This was verified by averaging only the highest-shot test configurations, namely those with 1600 and 3200 measurement shots (Figure 5), which are shown in Section 6.1.3 to be less affected by false positives. With this setup, no statistically significant correlation was found with the number of properties ($r = 0.117$, $p = 0.090$). This suggests a dependency between algorithmic constructs, the specific properties being tested, and the number of measurement shots required to minimise false positive results.

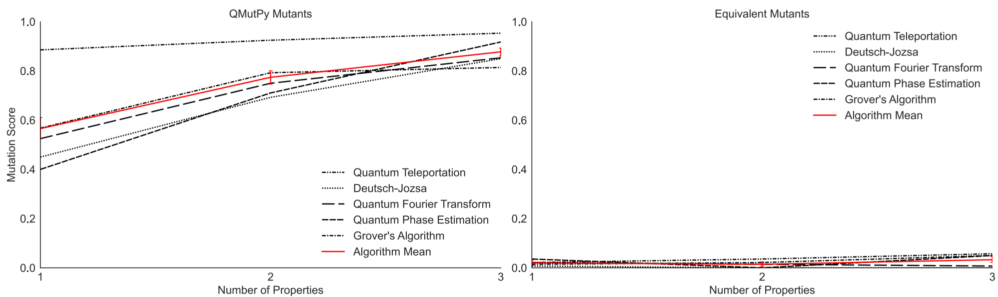

Fig. 5. Effect of *number of properties* on mutation score with 1600, and 3200 measurement shots.

Answer to RQ1.1: How does the *number of properties per program* affect the ability to identify faults in the quantum programs?

Increasing the number of properties in quantum property-based test suites significantly improves fault detection in quantum programs. While this initially raises the false positive rate, using an adequate number of measurement shots (which is algorithm-property dependent) mitigates this effect.

6.1.2 RQ1.2: Effect of Number of Inputs per Property. Analysis of Figure 6 revealed a weak positive correlation between the number of inputs generated within a property-based test and the mutation

score (r = 0.190, p = 4.19e-9). The median mutation score observed was 0.7 when generating 1 input, 0.75 for 2 and 4 inputs, 0.8 for 8 and 16 inputs, and 0.85 for 32 and 64 inputs. The standard deviation ranged between 0.188 to 0.221 across the number of inputs, without a clear decreasing pattern as seen when varying the number of properties.

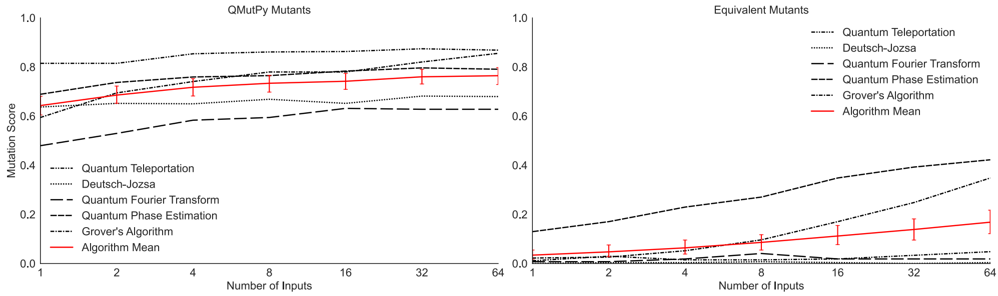

Fig. 6. Effect of *number of inputs* per property on mutation score.

The **equivalent mutant** set showed a similar trend to the QMutPy set; a weak positive correlation (r = 0.217, p = 1.56e-11) was observed. The median mutation score remained at zero for all configurations of number of inputs. However, the standard deviation of the mutation score increased by approximately 0.03 when doubling the number of inputs, ranging from 0.119 at one input, to 0.284 at 64 inputs, indicating a decrease in consistency of the mutation scores. As in RQ1.1, the *quantum phase estimation algorithm* (Figure 6) had the highest false positive rate across the range of inputs.

When considering only the subset of results containing *three properties and two largest numbers of measurement shots*, the positive correlation between the number of inputs and the mutation score for the **equivalent mutant set** lost its statistical significance (r = 0.209, p = 0.083). Conversely, the correlation for **QMutPy mutants** increased to r = 0.337, p = 0.004.

Answer to RQ1.2: How does the *number of inputs per property* affect the ability to identify faults in the quantum programs?

Increasing the number of inputs generated per property has a weak impact on fault identification for QMutPy mutants, with a weak positive correlation observed for the equivalent mutants. However, with the most thorough configurations, the positive correlation with the mutation score for equivalent mutants loses statistical significance, while the QMutPy mutant's positive correlation coefficient increases.

6.1.3 RQ1.3: Effect of Measurement Shots. A weak statistical correlation was observed between the number of measurements performed (Figure 7) for each input in the property-based test (r = 0.112, p = 5.94e-4). The observed increase in mutation score occurred between 12 and 200 measurements, where the mean increased from 0.598 to 0.743, after which only marginal gains were observed. The median mutation score increased from 0.6 at 12 measurements to 0.75 at 25, and 0.8 for the remaining measurement values. The standard deviation of the mutation score decreased from 0.263 to 0.184 across the range of inputs.

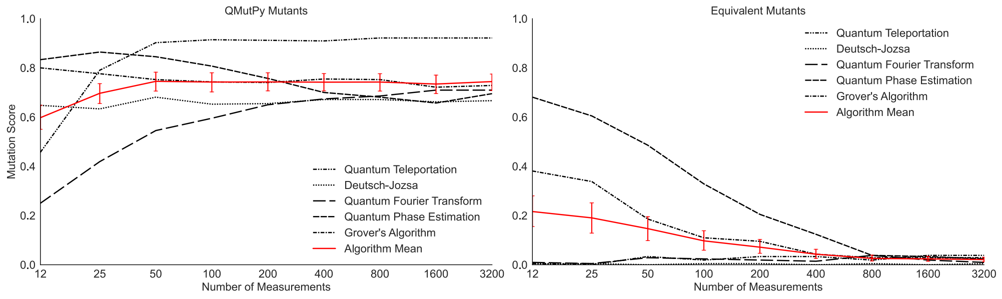

Fig. 7. Effect of *measurement shots* on mutation score.

The trend for **equivalent mutants** differed from the patterns observed with other variables. A statistically significant negative correlation (r = -0.235, p = 2.33e-13) was found between the number of measurements and mutation score. The median mutation score remained constant at zero across all measurements, while the consistency of the output improved, as shown by a reduction in the standard deviation of the mutation score from 0.339 to 0.047.

Answer to RQ1.3: How does the *number of measurement shots* affect the ability to identify faults in the quantum programs?

Increasing the number of measurement shots for QMutPy mutants improved fault detection, with only marginal gains observed beyond 200 measurements.
For equivalent mutants, the mutation score decreases as the number of measurements increases, which can be leveraged to reduce false positives in property-based tests. Beyond 800 measurements, additional measurements yielded only a negligible reduction in false positives.

The specific values reported here should be interpreted in the context of the subject programs, circuit sizes, properties, and assertions used in this study, and may differ for other settings.

6.2 RQ2: property-based testing effectiveness

Configuration			Algorithm Average Time Taken (s)				
Properties	Inputs	Measurements	QT	DJ	QFT	QPE	Grover
3	100	4200	19.09	11.50	121.31	413.81	123.45
3	100	2500	15.17	9.87	107.06	298.34	121.94

Table 3. Execution time comparison of different configurations.

Figure 8 presents two bar plots that illustrate the relative mutation scores for the two selected configurations against a unit testing baseline. The upper bar plot represents the mutation scores for **QMutPy-generated mutants**, while the lower one displays the scores for **equivalent mutants**. Following the trend observed in RQ1.3, the 2500 shot per basis configuration performed identically in fault identification to the 4200 shot configuration for non-equivalent mutants, but observed an increase to false positives. For QMutPy mutants, the mean mutation score across all algorithms was 0.90 for both the 2500 and 4200 shot experiments, improving over the 0.74 mean mutation score

observed for the unit test baseline. In contrast, the mean mutation scores for equivalent mutants decreased when increasing measurement shots from 0.06 at 2500 shots to 0 at 4200 shots, with the unit test baseline having a mean of 0.03.

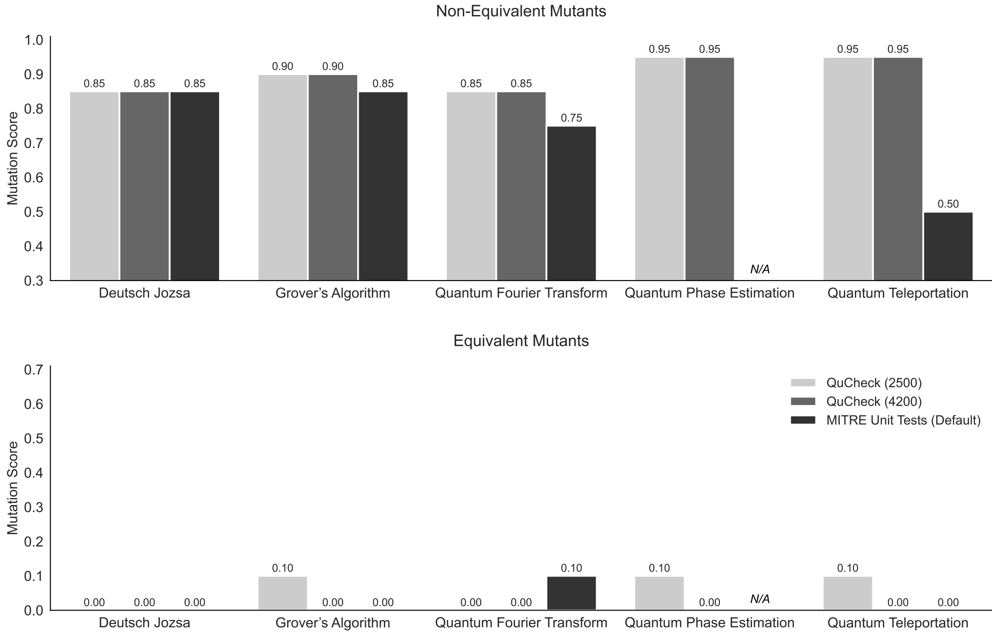

Fig. 8. Effectiveness of property-based testing on the 2500 and 4200 measurement shot configurations compared to the unit testing baseline.

The improved fault-detection capability came at the expense of execution time, though it remained feasible across all case studies. For the configuration with 4200 measurements, the execution times varied from 11.50 seconds for the Deutsch-Jozsa algorithm to 413.81 seconds for Grover's algorithm. This is in contrast to the average time taken for the weakest configuration, which ranged from 9.87 seconds to 298.34 seconds.

RQ2: Is property-based testing *effective* for the identification of faults in quantum programs?

The high mutation scores observed across the tested configurations demonstrate a significant ability to detect faults in quantum programs, showing a 22% increase over the unit test baseline. Furthermore, the false positive rate was lower with the QuCheck 4200 measurement shot configuration than with the unit testing baseline. This was achieved while maintaining feasible execution times across the properties and configurations tested.

6.3 RQ3: Comparison of QuCheck and QSharpCheck

In Section 6.3.1, we identify the qualitative differences between the two frameworks, finding QuCheck offers greater flexibility and extensibility in defining properties compared to QSharpCheck. Key differences include the programming language used, the format for property specification, the input generators and assertions, the generality of precondition checks, and statistical analysis.

Aspect	QuCheck	QSharpCheck
Programming Language	Python (Qiskit)	Q#
Property Specification	In a Python class	In a text file
Input Generation	Customisable Deterministic Various types (i.e., Oracle circuits, Quantum states, Integers) Multiple qubit states supported	Single qubit states can be chained together Basic types (i.e. Boolean)
Preconditions	Any boolean function using generated inputs	Limited to θ and ϕ of individual qubits
Assertions	AssertEqual AssertProbability AssertEntangled AssertDifferent AssertSeparable AssertMostFrequent (Multiple basis checks)	AssertEqual AssertProbability AssertEntangled AssertTransformed AssertEqualClassicalBits AssertTeleported (Computational basis only)
Statistical Analysis	Statistical correction applied	No statistical correction

Table 4. Comparison of QuCheck and QSharpCheck testing frameworks.

In Section 6.3.2, we analyse the quantitative performance of the two frameworks, demonstrating QuCheck's improved fault-detection capabilities. QuCheck consistently identified more mutants than QSharpCheck in both the quantum teleportation and superdense coding case studies, with a lower execution time on average.

> Answer to **RQ3:** How does QuCheck compare to QSharpCheck?
>
> QuCheck offers a more expressive and extensible testing framework than QSharpCheck, through improvements to property specification, statistical assertions, and a statistical correction for multiple test executions. Quantitatively, QuCheck identified significantly more mutants for quantum teleportation and superdense coding, with a lower average execution time, though it produced one false positive result.

6.3.1 RQ3.1: Qualitative differences. There are a number of qualitative differences that were considered for the evaluation of both frameworks, a summary of the differences can be seen in Table 4

- **Programming Language:** The choice of Python and Qiskit for QuCheck was informed by our previous research [Pontolillo and Mousavi 2022], which highlighted Qiskit's prevalence in quantum programming repositories. QuCheck builds on the features provided by QSharpCheck, emphasising extensibility to facilitate the creation of custom input generators and assertions tailored to the properties being tested. Additionally, Python's robust package ecosystem further supports this extensibility, enabling seamless integration of external libraries for specialised testing scenarios.

- **Property Specification:** Properties in QSharpCheck are defined in a separate text file, whereas in QuCheck, property specification is done within a Python class. This approach enables a higher degree of flexibility when implementing properties: custom input generators can be easily defined and inserted, specialised functions can be used to verify preconditions, and the operation function allows for the definition of more advanced tasks beyond just initialising the circuit.
- **Input Generation:** QSharpCheck's input generation is limited to chaining together single qubit quantum states. QuCheck expands on this by providing a set of predefined deterministic input generators, and an interface to develop new generators. This enables the generation of a wide variety of input types (i.e. oracle circuits), which facilitate the conversion of conceptual properties into property-based tests that can be executed. This capability was pivotal in our case study on Grover's algorithm, where random Grover's oracles that mark a customisable range of states were included as test inputs, an excerpt can be seen in Listing 2.
- **Preconditions** In QSharpCheck, preconditions can be used to ensure the randomly generated quantum states fall within a specified range for θ and ϕ. Preconditions in QuCheck are a boolean function that receives the randomly generated inputs. If valid inputs cannot be generated within a set number of attempts, QuCheck times out and returns a failure for the property.
- **Assertions:** QuCheck offers unique assertions such as AssertMostFrequent, AssertDifferent and AssertSeparable. While it does not include AssertTeleported, or AssertEqualClassicalBits, similar results can be achieved through the application of AssertEqual, as exemplified in Listing 1. Additionally, QuCheck's assertions consider different measurement bases; for AssertEqual, the option to check the X, Y, and Z bases is available, whereas QSharpCheck only measures the computational (Z) basis. Similar to input generation, QuCheck provides an interface for creating custom assertions, which can exploit the extensive Python ecosystem.
- **Statistical analysis:** QSharpCheck performs a fixed number of experiments with a specified number of shots for each test case, and applies a statistical test using the mean of the outcomes. QuCheck conducts one experiment containing all measurement shots and applies the Holm-Bonferroni correction across all statistical tests.

Impact of capabilities: *examples and consequences.* In the following, we illustrate how each capability affects what can be specified or detected.

Specification in Python. Expressing the entire property within Python enables the use of all features and abstraction mechanisms of the language (functions, modules, parameterisation). In contrast, a plaintext property file is bounded by the grammar and semantics of its parser; while it may be possible to express behaviour not directly supported by the parser, doing so requires workarounds (e.g., pushing input-generation logic into the operation body), which increases the effort required to implement the properties, or reduces the ability to re-use code.

Generic inputs and arbitrary preconditions. Some properties require algorithm-specific inputs beyond what a fixed generator library can supply. QuCheck allows a user to provide custom input generators when defining properties. This capability facilitates the implementation of the "Marked Most Frequent" property from Table 1, which tests Grover's algorithm using randomly generated marking circuits as input. Listing 2 shows the input generation and precondition verification steps of the property. `RandomGroversOracleMarkedStatesPairGenerator(4, 7)` generates a random marking circuit between 4 and 7 qubits and selects the number of marked states at random. The property requires an oracle that marks at least one but fewer than half of the total number of states. This constraint is enforced using a precondition, which is a simple Boolean function, saving the need to create new input generator. One may avoid using a precondition function by creating a

more sophisticated input generator, although this can be more challenging than just inserting a precondition.

```
class GroversAlgorithmMostFrequentMarked(Property):
    def get_input_generators(self):
        return [RandomGroversOracleMarkedStatesPairGenerator(4, 7)]

    def preconditions(self, oracle_pair):
        oracle, marked_states = oracle_pair
        if len(marked_states) == 0 or len(marked_states) > 2**(oracle.num_qubits
            - 1) - 1:
            return False
        return True

    def operations(...)
```

Listing 2. Grover's algorithm property excerpt that requires a specific range of states to be marked

Test Case Reproducibility. QuCheck's test input generation is deterministic, a seed can be passed to the framework, which can be used to generate unique sets of inputs to evaluate. If a failure is observed, its inputs can be reproduced by passing in the same seed, allowing a developer to confirm whether a future fix removes the failure.

Multi-basis, multi-qubit assertions. Some properties require measurements across multiple bases and over multiple qubits. For example, the "Measurement Registers Reset" property (Table 1) checks that the two non-teleported registers are in an expected state after teleportation. Any state that collapses to 0 and 1 with equal probability (e.g., $|-\rangle$) cannot be distinguished from $|+\rangle$ when measuring only in the Z basis. The ability to perform such measurements on multiple qubits enables us to verify the $|++\rangle$ state using one assertion, which can be seen on line 20, Listing 3. Furthermore, multiple qubit assertions save the need to call multiple single qubit assertions, and offer the ability to assert entangled states (Currently some multi-qubit assertions check individual qubit distributions separately; the framework, however, supports assertions that do not only consider marginal distributions, such as `AssertEntangled`).

```
class NotTeleportedPlus(Property):
    def get_input_generators(self):
        state = RandomState(1)
        return [state]

    def preconditions(self, q0):
        return True

    def operations(self, q0):
        qc = QuantumCircuit(3, 3)
        qc.initialize(q0, [0])
        qt = quantum_teleportation()
        qc = qc.compose(qt)

        # initialise another circuit to |++> state for comparison
        qc2 = QuantumCircuit(2, 2)
        qc2.h(0)
        qc2.h(1)

        self.statistical_analysis.assert_equal(self, [0, 1], qc, [0, 1], qc2)
```

Listing 3. Teleportation property that checks two pairs of qubits are in equal states

Metamorphic two-circuit comparisons. When no oracle (expected output or target state) is available for a given program, correctness can be specified via *relations* between two executions of the same program. QuCheck supports this natively: a property can construct multiple circuits with related inputs and directly compare their outcomes. QSharpCheck lacks native support, but can implement such properties with extra work. To compare two executions, users must embed both runs inside a single wrapper operation so that assertions can access both outcomes; this makes even simple metamorphic relations more verbose and error-prone to implement. An example of a metamorphic property implemented in QuCheck can be seen in Listing 4, it asserts that a pair of teleportation executions should be equal regardless of whether a unitary is applied before or after teleportation.

```
class UnitaryBeforeAndAfterTeleport(Property):
    def get_input_generators(self):
        state = RandomState(1)
        unitary = RandomUnitary(1, 1)
        return [state, unitary]

    def preconditions(self, q0, unitary):
        return True

    def operations(self, q0, unitary):
        # apply unitary on first qubit then teleport
        qc = QuantumCircuit(3, 3)
        qc.initialize(q0, [0])
        qc.append(unitary, [0])
        qt = quantum_teleportation()
        qc = qc.compose(qt)

        # apply teleport then apply unitary on third qubit
        qc2 = QuantumCircuit(3, 3)
        qc2.initialize(q0, [0])
        qt2 = quantum_teleportation()
        qc2 = qc2.compose(qt2)
        qc2.append(unitary, [2])

        self.statistical_analysis.assert_equal(self, [0, 1, 2], qc, [0, 1, 2],
            qc2)
```

Listing 4. Teleportation property that checks two pairs of qubits are in equal states

Answer to **RQ3.1:** What are the qualitative differences between QuCheck and QSharpCheck?

QuCheck offers greater expressivity and flexibility than QSharpCheck by defining properties directly in Python files, bypassing QSharpCheck's plain-text parsing limitations. This enables more customisable input generation, precondition checks, and complex operations (e.g., metamorphic properties) before assertions. Designed for extensibility, QuCheck simplifies the addition of new input generators and custom assertions.
QuCheck's predefined assertions are more distinct and versatile, supporting multiple qubits and measurement bases. It expands input generation beyond QSharpCheck's single-qubit states to include entangled multi-qubit states and specialised inputs like Grover's oracles. Additionally, QuCheck can also execute multiple properties in a single experiment with statistical corrections for multiple testing.

6.3.2 RQ3.2: Quantitative differences. To quantitatively compare the differences of QuCheck and QSharpCheck, we conducted the two case studies that were used for the original QSharpCheck paper [Honarvar et al. 2020]; Superdense Coding, and Quantum Teleportation.

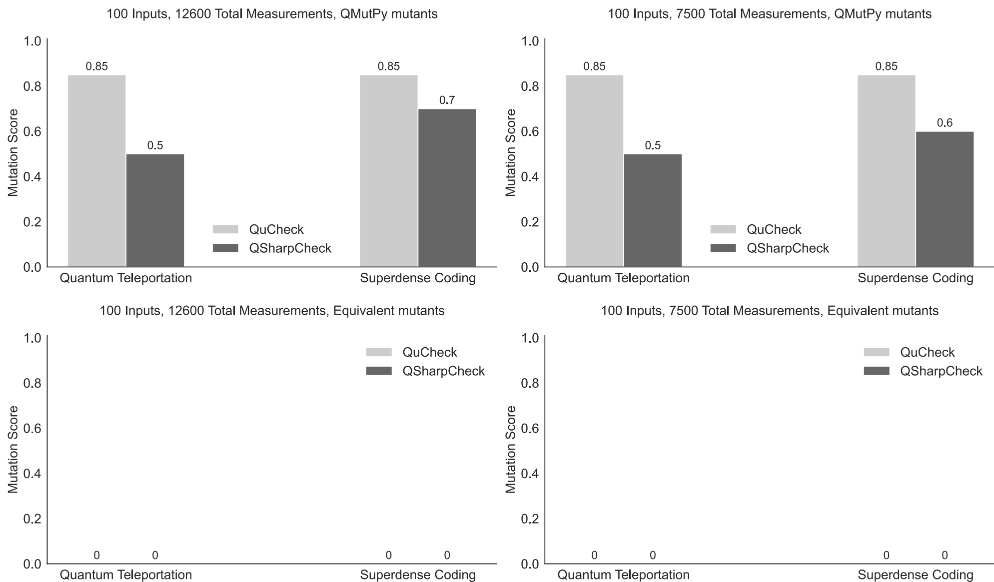

Fig. 9. Mutation score comparison between QuCheck and QSharpCheck for Quantum Teleportation, and Superdense Coding.

The property used for quantum teleportation was the same in both frameworks, with the difference being that our approach verified the output state across three bases. For superdense coding, our approach differed by verifying that the output qubits were in the $|0\rangle$ and $|1\rangle$ states, rather than checking their binary output. This approach enables the detection of phase-related mutations by verifying across three bases.

The impact of these differences can be seen across all configurations, with the greatest effect on quantum teleportation, where QuCheck always successfully detected at least three additional

mutants out of the *twenty* tested from the QMutPy set (Figure 9). Overall, when averaging across both algorithms and measurement configurations, QuCheck achieved an average mutation score 0.85 for the QMutPy mutants, compared to 0.58 with QSharpCheck.

Figure 10 presents the average execution time for each configuration. For *QMutPy mutants*, the median execution times for 12600 and 7500 total measurements in QuCheck were 5.28 and 4.91 seconds, compared to 0.744 and 0.953 seconds in QSharpCheck. The standard deviations for these measurements were 1.88 and 1.30 seconds for QuCheck, and 25.48 and 15.64 seconds for QSharpCheck. The low medians but high standard deviations in QSharpCheck are due to execution halting upon the detection of a failure, unlike QuCheck (Sec. 4.6). Despite this, QSharpCheck's execution time was, on average, 67.4% longer than QuCheck's.

For *Equivalent mutants*, no failures were detected in QSharpCheck, so this trend did not occur. Median execution times for 12600 and 7500 measurements in QuCheck were 4.01 and 3.26 seconds, while QSharpCheck recorded 55.84 and 32.48 seconds. The standard deviations were 1.26 and 0.72 seconds for QuCheck, and 10.10 and 7.04 seconds for QSharpCheck. For this set of mutants, QuCheck demonstrates an even greater performance advantage, requiring an average of 91.3% less time.

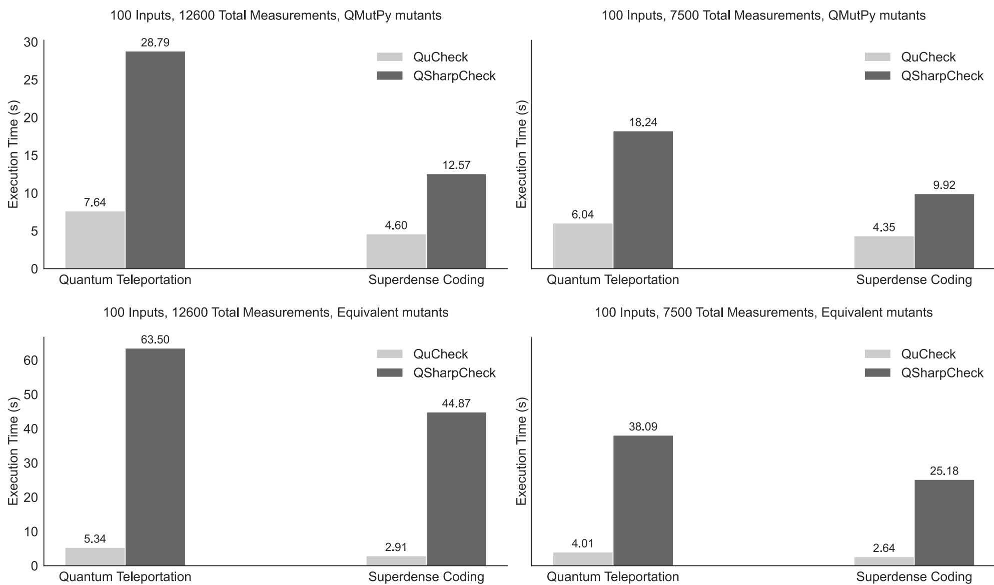

Fig. 10. Average execution time (s) comparison between QuCheck and QSharpCheck for Quantum Teleportation, and Superdense Coding.

Answer to **RQ3.2:** How does QuCheck compare to QSharpCheck in identification of faults in quantum programs?

QuCheck demonstrated a greater ability to identify faults in QMutPy mutants across the four most thorough test configurations, likely due to its measurement of multiple bases in assertions. It identified 47.8% more faults with a 67.4% decrease in average execution time compared to QSharpCheck when controlling for the total number of measurement shots. No false positives occurred in either framework in the equivalent mutant set, although QuCheck achieved a 91.3% decrease in execution time.

7 THREATS TO VALIDITY

- **Mutant generation and translation:** The quantitative analysis between QuCheck and QSharpCheck required the translation of our mutants into Q#. The translation was partly automated through the use of the Qiskit API and QConvert, this process returned some programs which were not syntactically correct, requiring manual translation that is prone to human error. Furthermore, the synthetically generated mutants generated using QMutPy may not accurately represent real faults that would be introduced to these programs by developers. Additionally, the pool of subject systems used for our experiments is of modest size, and it is not clear if the results can be generalised to other quantum programs.
- **Program selection and size:** The set of five quantum programs used as case studies is modest and may not be representative of the wider set of quantum programs. Moreover, our experiments were limited to relatively small circuits, since larger circuits become increasingly computationally intensive to analyse under classical simulation. This constrained the width and depth of the programs we could feasibly include, and therefore limits the extent to which the observed trends can be generalised to larger or more complex circuits. However, the properties considered in this work are based on input-output relations that are not inherently tied to the size of the circuit and are likely to remain relevant at larger scales. In this case, the main challenge is the computational cost of checking such properties rather than the applicability of the properties themselves.

8 CONCLUSION

In this paper, we introduced QuCheck, a property-based testing framework for Qiskit quantum programs. We demonstrated its efficiency and ability to detect faults at varying levels of thoroughness of the test suite. Furthermore, we compared QuCheck to QSharpCheck, highlighting how it addresses QSharpCheck's limitations and offers an accessible property-based testing solution for the Qiskit community.

Our experiments revealed a positive correlation between the number of measurements, inputs, and properties, and the mutation score, with the number of properties having the most significant impact. The latter finding highlights the importance of covering program semantics through the composition of multiple properties. Conversely, for the equivalent mutant set, we observed a negative correlation between the number of measurements and the mutation score, indicating a reduction in false positives as the statistical power of the tests increased. We also demonstrated the effectiveness and efficiency of the two most thorough test configurations along with their associated cost in execution time, confirming the feasibility of applying property-based testing to quantum programs. Finally, our results showed that QuCheck improved fault detection while reducing execution time compared to QSharpCheck.

Future work will focus on evaluating QuCheck's performance using real faults instead of synthetic mutants. Additional directions include applying property-based testing with a noisy simulator or real quantum hardware and exploring techniques such as classical shadows [H. Huang et al. 2020] to implement assertions more efficiently.

ACKNOWLEDGMENTS

The authors have been partially supported by the EPSRC project on Verified Simulation for Large Quantum Systems (VSL-Q), grant reference EP/Y005244/1 and the EPSRC project on Robust and Reliable Quantum Computing (RoaRQ), Investigation 009, grant reference EP/W032635/1. Also, King's Quantum grants provided by King's College London are gratefully acknowledged. The authors thank George Booth, Veronica Gaspes, anonymous referees, and VSL-Q project members for their constructive feedback.

REFERENCES

Rui Abreu, João Paulo Fernandes, Luis Llana, and Guilherme Tavares. 2022. "Metamorphic testing of oracle quantum programs." In: *3rd IEEE/ACM International Workshop on Quantum Software Engineering, Q-SE@ICSE 2022, Pittsburgh, PA, USA, May 18, 2022.* IEEE, 16–23. DOI: 10.1145/3528230.3529189.

Alan Agresti and Brent A. Coull. May 1998. "Approximate Is Better than "Exact" for Interval Estimation of Binomial Proportions." *The American Statistician*, 52, 2, (May 1998), 119. DOI: 10.2307/2685469.

Cláudio Amaral, Mário Florido, and Vítor Santos Costa. 2014. "PrologCheck – Property-Based Testing in Prolog." In: *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings* (Lecture Notes in Computer Science). Ed. by Michael Codish and Eijiro Sumii. Vol. 8475. Springer, 1–17. DOI: 10.1007/978-3-319-07151-0_1.

Thomas Arts, John Hughes, Joakim Johansson, and Ulf T. Wiger. 2006. "Testing telecoms software with quviq QuickCheck." In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006.* Ed. by Marc Feeley and Philip W. Trinder. ACM, Portland, Oregon, USA, 2–10. ISBN: 1595934901. DOI: 10.1145/1159789.1159792.

Tiago Araújo Castro. 2021. "Arbitraries with Confidence: Externalizing PBT inner-workings for better insights." Master's thesis. Faculty of Engineering, University of Porto.

Koen Claessen and John Hughes. 2000. "QuickCheck: a lightweight tool for random testing of Haskell programs." In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.* Ed. by Martin Odersky and Philip Wadler. ACM, 268–279. DOI: 10.1145/351240.351266.

Daniel Fortunato, José Campos, and Rui Abreu. 2022. "QMutPy: a mutation testing tool for Quantum algorithms and applications in Qiskit." In: *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022* (ISSTA 2022). Ed. by Sukyoung Ryu and Yannis Smaragdakis. ACM, Virtual, South Korea, 797–800. ISBN: 9781450393799. DOI: 10.1145/3533767.3543296.

Martin Fürer. 2008. "Solving NP-Complete Problems with Quantum Search." In: *LATIN 2008: Theoretical Informatics, 8th Latin American Symposium, Búzios, Brazil, April 7-11, 2008, Proceedings* (Lecture Notes in Computer Science). Ed. by Eduardo Sany Laber, Claudson F. Bornstein, Loana Tito Nogueira, and Luérbio Faria. Vol. 4957. Springer, 784–792. DOI: 10.1007/978-3-540-78773-0_67.

Gabriel Pontolillo, Mohammad Reza Mousavi, and Marek Grzesiuk. Mar. 2026. *qucheck*. https://pypi.org/project/qucheck/. Python Package Index (PyPI). (Mar. 2026).

Gabriel Pontolillo, Mohammad Reza Mousavi, and Marek Grzesiuk. Nov. 2024. *QuCheck: A Property-based Testing Framework for Quantum Programs in Qiskit.* (Nov. 2024). DOI: 10.6084/m9.figshare.27919539.

Brent Hailpern and Padmanabhan Santhanam. 2002. "Software debugging, testing, and verification." *IBM Syst. J.*, 41, 1, 4–12. DOI: 10.1147/SJ.411.0004.

S. L. N. Hermans, M. Pompili, H. K. C. Beukers, S. Baier, J. Borregaard, and R. Hanson. May 2022. "Qubit teleportation between non-neighbouring nodes in a quantum network." *Nature*, 605, 7911, (May 2022), 663–668. DOI: 10.1038/s41586-022-04697-y.

Sture Holm. 1979. "A Simple Sequentially Rejective Multiple Test Procedure." *Scandinavian Journal of Statistics*, 6, 2, 65–70. http://www.jstor.org/stable/4615733.

Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. 2020. "Property-based Testing of Quantum Programs in Q#." In: *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020.* ACM, 430–435. DOI: 10.1145/3387940.3391459.

Tianmin Hu, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Huanting Wang, Meng Li, and Zheng Wang. 2024. "UPBEAT: Test Input Checks of Q# Quantum Libraries." In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software*

Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024. Ed. by Maria Christakis and Michael Pradel. ACM, Vienna, Austria, 186–198. DOI: 10.1145/3650212.3652120.

Hsin-Yuan Huang, Richard Kueng, and John Preskill. June 2020. "Predicting many properties of a quantum system from very few measurements." *Nature Physics*, 16, 10, (June 2020), 1050–1057. DOI: 10.1038/s41567-020-0932-7.

Linzhi Huang, Hanyu Pei, Yuechen Li, Beibei Yin, and Kai-Yuan Cai. 2024. "A Strategy of Dynamic Random Testing with Hybrid Distance Metrics for Quantum Programs." In: *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*, 1–12. DOI: 10.1109/QRS62785.2024.00011.

Yipeng Huang and Margaret Martonosi. 2019. "Statistical assertions for validating patterns and finding bugs in quantum programs." In: *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman. ACM, 541–553. DOI: 10.1145/3307650.3322213.

John Hughes. 2016. "Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane." In: *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (Lecture Notes in Computer Science). Ed. by Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella. Vol. 9600. Springer, 169–186. DOI: 10.1007/978-3-319-30936-1_9.

John Hughes. 2007. "QuickCheck Testing for Fun and Profit." In: *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007* (Lecture Notes in Computer Science). Ed. by Michael Hanus. Vol. 4354. Springer, 1–32. DOI: 10.1007/978-3-540-69611-7_1.

W.M. Itano, J.C. Bergquist, J.J. Bollinger, J.M. Gilligan, D.J. Heinzen, F.L. Moore, M.G. Raizen, and D.J. Wineland. Jan. 1993. "Quantum measurements of trapped ions." *Vistas in Astronomy*, 37, (Jan. 1993), 169–183. DOI: 10.1016/0083-6656(93)90029-j.

Ali Javadi-Abhari et al.. 2024. *Quantum computing with Qiskit*. (2024). arXiv: 2405.08810 [quant-ph]. DOI: 10.48550/arXiv.2405.08810.

jclapis. 2019. *QSFE: Quantum Software Framework Evaluation*. https://github.com/jclapis/qsfe. Accessed: 2025-10-19. (2019).

Lava Prasad Kafle. 2014. "An Empirical Study on Software Test Effort Estimation." *International Journal of Soft Computing and Artificial Intelligence*, 2, 2.

Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. "Projection-based runtime assertions for testing and debugging Quantum programs." *Proc. ACM Program. Lang.*, 4, OOPSLA, 150:1–150:29. DOI: 10.1145/3428218.

Peixun Long and Jianjun Zhao. 2024. "Testing Multi-Subroutine Quantum Programs: From Unit Testing to Integration Testing." *ACM Trans. Softw. Eng. Methodol.*, 33, 6, 147. DOI: 10.1145/3656339.

David MacIver, Zac Hatfield-Dodds, and Many Contributors. Nov. 2019. "Hypothesis: A new approach to property-based testing." *Journal of Open Source Software*, 4, 43, (Nov. 2019), 1891. DOI: 10.21105/joss.01891.

Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. 2021. "Muskit: A Mutation Analysis Tool for Quantum Software Testing." In: *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1266–1270. DOI: 10.1109/ASE51524.2021.9678563.

Microsoft. 2020. *Q# Language Specification*. https://github.com/microsoft/qsharp-language/tree/main/Specifications/Language#q-language.

Asmar Muqeet, Tao Yue, Shaukat Ali, and Paolo Arcaini. 2024. "Mitigating Noise in Quantum Software Testing Using Machine Learning." *IEEE Transactions on Software Engineering*, 50, 11, 2947–2961. DOI: 10.1109/TSE.2024.3462974.

National Institute of Standards & Technology. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Tech. rep. NIST Planning Report 02-03.

Michael A. Nielsen and Isaac L. Chuang. 2016. *Quantum Computation and Quantum Information (10th Anniversary edition)*. Cambridge University Press. ISBN: 978-1-10-700217-3.

Rickard Nilsson. Aug. 2014. *Scala check: The definitive guide: Property-based testing on the java platform*. Artima, Sunnyvale, CA, (Aug. 2014).

Matteo Paltenghi and Michael Pradel. 2024. "Analyzing Quantum Programs with LintQ: A Static Analysis Framework for Qiskit." *Proc. ACM Softw. Eng.*, 1, FSE, 2144–2166. DOI: 10.1145/3660802.

Matteo Paltenghi and Michael Pradel. 2023. "MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform." In: *Proceedings of the 45th International Conference on Software Engineering* (ICSE '23). IEEE Press, Melbourne, Victoria, Australia, 2413–2424. ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00202.

Gabriel Pontolillo and Mohammad Reza Mousavi. 2022. "A Multi-Lingual Benchmark for Property-Based Testing of Quantum Programs." In: *3rd IEEE/ACM International Workshop on Quantum Software Engineering, Q-SE@ICSE 2022, Pittsburgh, PA, USA, May 18, 2022*. IEEE, 1–7. DOI: 10.1145/3528230.3528395.

Gabriel Pontolillo and Mohammad Reza Mousavi. 2024. "Delta Debugging for Property-Based Regression Testing of Quantum Programs." In: *Proceedings of the 5th ACM/IEEE International Workshop on Quantum Software Engineering, Q-SE 2024, Lisbon, Portugal, 16 April 2024* (Q-SE 2024). ACM, Lisbon, Portugal, 1–8. DOI: 10.1145/3643667.3648219.

C. Vidya Raj and M. S. Shivakumar. 2006. "Applying Quantum Algorithm to Speed Up the Solution of Hamiltonian Cycle Problems." In: *Intelligent Information Processing III, IFIP TC12 International Conference on Intelligent Information Processing*

(IIP 2006), September 20-23, Adelaide, Australia (IFIP). Ed. by Zhongzhi Shi, Katsunori Shimohara, and David Dagan Feng. Vol. 228. Springer, 53–61. DOI: 10.1007/978-0-387-44641-7_6.

Ji-Gang Ren et al.. Sept. 2017. “Ground-to-satellite quantum teleportation.” en. *Nature*, 549, 7670, (Sept. 2017), 70–73.

Cong-Gang Song and Qing-yu Cai. July 2025. “Excessive precision compromises accuracy even with unlimited resources due to the trade-off in quantum metrology.” *npj Quantum Information*, 11, 1, (July 2025). DOI: 10.1038/s41534-025-01071-4.

Jiyuan Wang, Ming Gao, Yu Jiang, Jian-Guang Lou, Yue Gao, Dongmei Zhang, and Jiaguang Sun. 2018. “QuanFuzz: Fuzz Testing of Quantum Program.” *CoRR*, abs/1810.10310. http://arxiv.org/abs/1810.10310 arXiv: 1810.10310.

Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2023. “QuCAT: A Combinatorial Testing Tool for Quantum Software.” In: *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2066–2069. DOI: 10.1109/ASE56229.2023.00062.

Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. “QuSBT: search-based testing of quantum programs.” In: *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, New York, NY, USA, 173–177. DOI: 10.1145/3510454.3516839.

Xinyi Wang, Tongxuan Yu, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. “Mutation-based test generation for quantum programs with multi-objective search.” In: *GECCO ’22: Genetic and Evolutionary Computation Conference, Boston, Massachusetts, USA, July 9 - 13, 2022*. ACM, 1345–1353. DOI: 10.1145/3512290.3528869.

Jiaming Ye, Shangzhou Xia, Fuyuan Zhang, Paolo Arcaini, Lei Ma, Jianjun Zhao, and Fuyuki Ishikawa. 2023. “QuraTest: Integrating Quantum Specific Features in Quantum Program Testing.” In: *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1149–1161. DOI: 10.1109/ASE56229.2023.00196.

Nengkun Yu. 2020. *Quantum Closeness Testing: A Streaming Algorithm and Applications*. (2020). https://arxiv.org/abs/1904.03218 arXiv: 1904.03218 [quant-ph].

Pengzhan Zhao, Xiongfei Wu, Zhuo Li, and Jianjun Zhao. 2023. “QChecker: Detecting Bugs in Quantum Programs via Static Analysis.” In: *4th IEEE/ACM International Workshop on Quantum Software Engineering, Q-SE@ICSE 2023, Melbourne, Australia, May 17, 2023*. IEEE, 50–57. DOI: 10.1109/Q-SE59154.2023.00014.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009