Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3763100

RESEARCH-ARTICLE

Shaking Up Quantum Simulators with Fuzzing and Rigour

VASILEIOS KLIMIS, Queen Mary University of London, London, U.K.

AVNER BENSOUSSAN, King's College London, London, U.K.

ELENA CHACHKAROVA, King's College London, London, U.K.

KARINE EVEN-MENDOZA, King's College London, London, U.K.

SOPHIE FORTZ, King's College London, London, U.K.

CONNOR LENIHAN, King's College London, London, U.K.

Open Access Support provided by:

Queen Mary University of London

King's College London

PDF Download
3763100.pdf
18 January 2026
Total Citations: 0
Total Downloads: 172

Published: 09 October 2025
Accepted: 12 August 2025
Received: 22 March 2025

Citation in BibTeX format

Shaking Up Quantum Simulators with Fuzzing and Rigour

VASILEIOS KLIMIS, Queen Mary University of London, UK
AVNER BENSOUSSAN, King's College London, UK
ELENA CHACHKAROVA, King's College London, UK
KARINE EVEN-MENDOZA, King's College London, UK
SOPHIE FORTZ, King's College London, UK
CONNOR LENIHAN, King's College London, UK

Quantum computing platforms rely on simulators for modelling circuit behaviour prior to hardware execution, where inconsistencies can lead to costly errors. While existing formal validation methods typically target specific compiler components to manage state explosion, they often miss critical bugs. Meanwhile, conventional testing lacks systematic exploration of corner cases and realistic execution scenarios, resulting in both false positives and negatives.

We present FuzzQ, a novel framework that bridges this gap by combining formal methods with structured test generation and fuzzing for quantum simulators. Our approach employs differential benchmarking complemented by mutation testing and invariant checking. At its core, FuzzQ utilises our Alloy-based formal model of QASM 3.0, which encodes the semantics of quantum circuits to enable automated analysis and to generate structurally diverse, constraint-guided quantum circuits with guaranteed properties. We introduce several test oracles to assess both Alloy's modelling of QASM 3.0 and simulator correctness, including invariant-based checks, statistical distribution tests, and a novel cross-simulator unitary consistency check that verifies functional equivalence modulo global phase, revealing discrepancies that standard statevector comparisons fail to detect in cross-platform differential testing.

We evaluate FuzzQ on both Qiskit and Cirq, demonstrating its platform-agnostic effectiveness. By executing over 800,000 quantum circuits to completion, we assess throughput, code and circuit coverage, and simulator performance metrics, including sensitivity, correctness, and memory overhead. Our analysis revealed eight simulator bugs, six previously undocumented. We also outline a path for extending the framework to support mixed-state simulations under realistic noise models.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Formal software verification**; *Compilers*; *Source code generation.*

Additional Key Words and Phrases: Quantum Software Engineering, Formal Methods, Alloy, Fuzzing, Differential Testing, Model-Based Testing, Quantum Circuit Simulation, Alloy Analyzer.

ACM Reference Format:
Vasileios Klimis, Avner Bensoussan, Elena Chachkarova, Karine Even-Mendoza, Sophie Fortz, and Connor Lenihan. 2025. Shaking Up Quantum Simulators with Fuzzing and Rigour. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 322 (October 2025), 29 pages. https://doi.org/10.1145/3763100

Authors' Contact Information: Vasileios Klimis, v.klimis@qmul.ac.uk, Queen Mary University of London, London, UK; Avner Bensoussan, King's College London, London, UK, avner.bensoussan@kcl.ac.uk; Elena Chachkarova, King's College London, London, UK, elena.chachkarova@kcl.ac.uk; Karine Even-Mendoza, King's College London, London, UK, karine.even_mendoza@kcl.ac.uk; Sophie Fortz, King's College London, London, UK, sophie.fortz@kcl.ac.uk; Connor Lenihan, King's College London, London, UK, connor.1.lenihan@kcl.ac.uk.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/10-ART322
https://doi.org/10.1145/3763100

1 Introduction

Noisy Intermediate-Scale Quantum (NISQ) devices define the current frontier of quantum computing, providing access to quantum systems with modest qubit counts and inherent noise characteristics [2, 6, 49]. While not yet fault-tolerant or sufficiently large to demonstrate definitive quantum advantage, these systems serve as crucial development platforms for quantum algorithms through frameworks including Cirq [17], Qiskit [26], Pennylane [72], Braket [3], Ocean [16], Azure Quantum [41], and Rigetti Forest [59]. As quantum platforms scale to support larger qubit counts and increased circuit complexity, ensuring their reliability and correctness has emerged as a critical challenge in quantum software engineering [30, 37, 46, 48, 58, 66, 73]. This necessitates rigorous testing of the simulators and compilers that constitute their foundational components. Analogous to classical computing infrastructure, comprehensive testing of these quantum platforms is essential for two key functions: the correct *translation* of quantum programs into executable circuits and the precise *simulation* of quantum circuit behaviour. Given the limited availability and substantial cost of accessing physical quantum hardware, these simulation tools play a pivotal role in quantum algorithm development, circuit optimisation, and supporting both research and educational initiatives in quantum computing.

Given the high cost of physical hardware access and the often proprietary nature of quantum toolchains[1], simulators have become the primary mechanism for developing and debugging quantum programs. The deep integration of simulators with compilers and development environments makes them effective tools for identifying bugs, from compilation crashes to subtle simulation inaccuracies, across the entire software stack. However, while reliable simulators are essential for quantum computing advancement [22], current validation techniques remain underdeveloped.

Existing automated testing approaches like metamorphic [30] and differential testing [7] often lack systematic exploration, while formal verification methods for quantum compilers typically target narrow components or transformations rather than holistic correctness [35, 36, 64], with evaluations often confined to small-scale circuits [24, 35]. Furthermore, a fundamental challenge is differentiating quantum-specific bugs from the large proportion of classical implementation errors that manifest in these complex platforms [46], complicating bug isolation and diagnosis.

Our Work. To overcome these limitations, our method: (i) formalises quantum circuit specifications using the Alloy specification language [27] and (ii) implements a structured fuzzing campaign for quantum simulators leveraging this formal model. Our approach uses the model as a guided test case generator, producing well-formed inputs for quantum simulators and validating outputs through differential reasoning. When simulators produce divergent outputs, three possible explanations emerge: 1) the Alloy model requires refinement, 2) an undocumented or underspecified behaviour exists in one of the quantum platforms, or 3) a logical bug is present in either the compiler or simulator. By integrating formal modelling with automated testing, we establish mechanised formal semantics for quantum simulator specifications. We present our experience designing and implementing a formal model that represents the standard quantum gate library using the Alloy modelling language [27]. Alloy's declarative approach to formally defining and analysing quantum circuits proves particularly effective for designing circuits that explore deep state spaces and edge cases. We iteratively refined the model in collaboration with quantum simulator experts, as illustrated in Figure 1, to enhance its expressiveness and accuracy. This refinement ensures the model faithfully represents standard quantum gates while supporting verification, validation and debugging tasks. Using this refined model, we generated simulable quantum test circuits and applied differential testing, complemented by additional verification techniques. This integration of formal methods and testing overcomes the limitations of each approach when used in isolation.

[1]Though portions of platforms & docs can be public e.g. https://github.com/quantumlib/Cirq, https://github.com/Qiskit/qiskit.

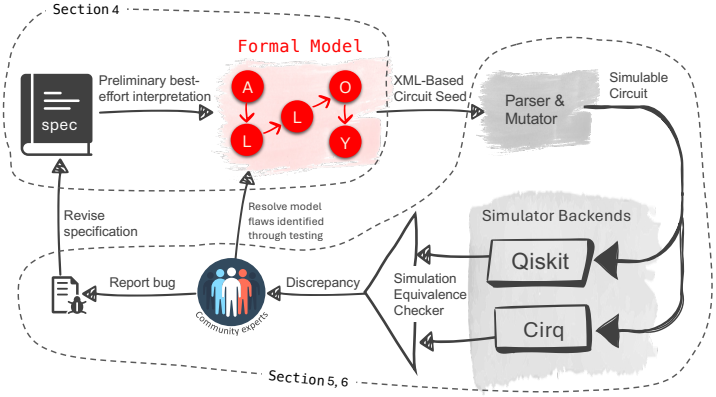


Fig. 1. Iterative validation of quantum simulators: Specifications are encoded in Alloy to generate XML-based circuit seeds. These are then parsed and mutated with random gate angles before execution on simulator backends. Discrepancies are reviewed by experts, leading to either formal model refinement or bug reporting.

Our work introduces the following key contributions:

- The *unitary parity benchmark*: An equivalence oracle for cross-simulator validation based on our *inter-unitary property*, which verifies that unitary matrices from different simulators are equivalent modulo global phase.
- *Seed cultivation*: A technique transforming abstract model-derived test cases into executable circuits across quantum frameworks while preserving their logical consistency through the unitary parity benchmark.
- A comprehensive *case study* demonstrating how integrating testing strategies with formal methods creates a refinement cycle that iteratively improves the formal model and reveals cross-platform specification inconsistencies.
- FuzzQ: An *open-source framework* implementing our approach that identified 8 distinct bugs, 6 previously undiscovered, in quantum simulation platforms.

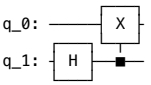
Paper Structure. §2 defines the research problem, followed by an overview of key quantum concepts in §3. §4 outlines the circuit model framework, while §5 discusses invariant-based testing. §6 explores fuzzing-based benchmarking, and §7 covers threats to validity of our model and approach. §8 presents our evaluation and analyses performance, simulator issues and limitations, §9 discusses related work. §11 summarises our findings. Data Availability Statement is in §12.

2 Testing of Quantum Platforms

This paper proposes a hybrid methodology that integrates formal verification techniques with systematic testing to create a comprehensive benchmarking framework for quantum simulators. While testing enhances scalability by accommodating complex systems under test (SUTs) that may comprise thousands of lines of code, formal methods provide precise control over test diversity and ensure adherence to physical constraints. This approach is particularly valuable for designing realisable quantum circuits with specific structural characteristics and operational semantics, addressing fundamental limitations of purely random testing approaches. By combining formal verification with systematic testing, we establish a rigorous framework that significantly improves the reliability and correctness guarantees of quantum simulators.

Why Testing Quantum Simulators is Intrinsically Tied to Testing Quantum Platforms?

```qiskit
from qiskit import QuantumCircuit
qc = QuantumCircuit(2) # Create a 2-qubit circuit
qc.h(1)     # Apply Hadamard gate to qubit 1 to create a superposition
qc.cx(1,0)  # Apply CNOT with qubit 1 as control and qubit 0 as target
```




```cirq
import cirq
q0, q1 = cirq.LineQubit.range(2) # Define qubits
circuit = cirq.Circuit( # Create a circuit with H-gate on q1 and CNOT
  cirq.H(q1),
  cirq.CNOT(q1,q0)    )
```

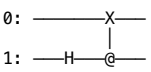


Fig. 2. Circuits compiled from Qiskit (top) and Cirq (bottom), both featuring a Hadamard gate and a CX gate. The circuit diagrams on the right are rendered in their respective native notations.

Quantum platforms constitute integrated ecosystems that combine hardware and software components for developing, simulating, and executing quantum algorithms. The hardware layer encompasses physical infrastructure – quantum processors and communication systems based on various technologies such as superconducting qubits, trapped ions, and photonic systems. Complementing this, the software layer comprises frameworks, libraries, and simulators that facilitate quantum algorithm design, compilation, optimisation, and validation. These platforms, typically implemented using high-level, general-purpose languages like C/C++, Java, and Python [50], enable researchers and developers to validate quantum programs before deployment on expensive and resource-intensive quantum hardware. This economic reality makes thorough and systematic simulator benchmarking essential for the quantum computing ecosystem.

Our benchmarking methodology focuses on the software layer, targeting two key issues: (1) *miscompilations* – errors in quantum gate sequence generation producing incorrect circuit implementations, and (2) *missimulations* – discrepancies in simulated behaviour across different platforms. Addressing both is essential for ensuring correct execution and enabling effective debugging for quantum program developers. We evaluate two predominant quantum frameworks: Qiskit [26] and Cirq [17]. These Python-based frameworks provide comprehensive toolchains for designing, simulating, and optimising quantum algorithms through circuit-based representations for eventual hardware execution. Despite architectural differences, these frameworks should produce functionally equivalent compiled circuits from identical logical inputs, yielding identical results as illustrated in Figure 2. Yet, subtle discrepancies may emerge due to variations in compilation strategies, optimisation techniques, simulation fidelity, and numerical precision handling. This observation motivates our differential testing methodology, which compares outputs from minimal quantum programs derived from identical logical circuit specifications. By identifying inconsistencies between implementations, we can pinpoint potential defects in either framework. We explore this approach in §6.

Challenges in Testing Quantum Simulators. Testing quantum simulators presents several inherent challenges that significantly complicate verification and validation efforts. First, *state space explosion* occurs as the state space grows exponentially with the number of qubits, making exhaustive verification impractical. *Non-deterministic outputs* pose another challenge. Due to the probabilistic nature of quantum measurements, multiple runs of identical circuits may yield different results, making it impossible to define a single correct output. This requires sophisticated statistical methods to compare output distributions against theoretical expectations, especially for large systems. The probabilistic nature also leads to a *lack of ground truth*. This ambiguity significantly complicates correctness verification, particularly when evaluating subtle implementation differences or novel algorithms. *Simulation fidelity* is another issue, as simulators often approximate quantum behaviour due to computational resource constraints. Ensuring accurate replication of quantum hardware behaviour, especially when modelling noise, decoherence, and error correction mechanisms, presents

substantial technical challenges. Finally, *divergent implementations* across simulators, due to differences in internal representations, approximations, and optimisations, can manifest as subtle inconsistencies in simulation results, complicating cross-platform verification and validation, and potentially masking real errors.

Formal methods offer a powerful approach for handling non-determinism and diverse implementations by generalising the verification process to all possible inputs (e.g. model checking and symbolic execution). Yet, their applicability is limited by state-space explosion, making them impractical for verifying complex quantum platforms. In contrast, testing scales more effectively to larger circuits but comes with its limitations. Some testing approaches lack in-depth code and bug analysis capabilities (e.g. failing to capture error models or producing false positives/negatives), while others provide detailed analysis but suffer from limited scalability (e.g. allowing a restricted number of mutations) [7, 47]. Both formal methods and testing face exponential computational costs when conducting precise static analysis, simulation, or comparisons with concrete quantum hardware execution.

Combining Formal Methods and Testing. We propose an approach that integrates formal methods (via Alloy) with systematic testing (via fuzzing) to address these challenges. Figure 1 illustrates `FuzzQ` 's methodology for validating quantum simulators. The process begins with encoding specifications in Alloy to generate *XML-based circuit seeds*. These templates are parsed and instantiated into *concrete simulatable circuits*, with random gate angles applied before execution on simulator backends, like Qiskit and Cirq. We perform differential testing on the compiled circuits, with discrepancies undergoing expert review, to drive model refinement, bug identification, or classification as "not interesting".

- Model Refinement: A mismatch may indicate that our formal model is too coarse. We refine the model iteratively to address specification gaps and improve the model and underlying specification.
- Bug Identification: Beyond specification gaps, discrepancies may stem from errors in quantum platforms or bugs in classical libraries (e.g. Python or Java). For example, during 16-qubit circuit simulations, we encountered a known issue in `numpy` [44, 45], previously reported as a reliability concern for quantum platforms [47]. Other bugs (reported in §8) include compilation and simulation failures, like crashes, hangs, miscompilations, and missimulations.

Divergent simulation results across platforms typically signal a mismatch. If the test circuit is valid, this suggests a missimulation or a miscompilation bug. In §3.1 and §5, we describe how we model quantum programming language specifications to generate quantum circuits of varying sizes and complexities. In §6, we explore how this modelling approach enables fuzzing by applying various test criteria to differential testing outputs (e.g. gate fidelity, unitary consistency) and validating results using statistical tools such as *Jensen-Shannon Divergence* (JSD) and the *Chi-Square Goodness-of-Fit Test* (χ^2). In §7, we outline strategies to address non-bug discrepancies like global phase differences.

3 Background

3.1 Formal Methods and Testing

Formal methods provide a rigorous approach for specifying, designing, and verifying systems, ensuring correctness and reliability. They enable formal specification of properties and systematic exploration of behaviours, typically using a specification language. In this work, we use Alloy, a declarative lightweight specification language [27].

Alloy – Constraint-based modelling languages like Alloy [27] offer a powerful approach to system specification, transforming abstract system descriptions into rigorous, analysable models. Central to Alloy's power is its Analyser – an automated reasoning engine that translates relational specifications into Boolean satisfiability (SAT) problems. The Analyser acts as a sophisticated mechanical theorem prover, automatically exploring model properties by systematically generating and testing instances within user-defined scopes. By leveraging modern SAT solving techniques, it can exhaustively search through potential system configurations, uncovering subtle structural properties, invariants, and potential design vulnerabilities.

In our quantum circuit modelling (§4), Alloy's type hierarchy, comprising of **`abstract sig`**natures and concrete **`sig`**natures[2], allows for nuanced representation of quantum circuit architectures. Abstract signatures capture generalised structural patterns, while concrete signatures instantiate specific computational elements.

Testing – Traditional testing methods validate system behaviour by comparing actual outcomes with expected ones, or by ensuring compliance with a predefined specification. In this work, we integrate several complementary techniques into a unified framework. **Model-based testing** [32] generates test cases from formal specifications to ensure adherence to the intended design. Complementing this, **property-based testing** shifts the focus from fixed test cases to high-level properties that the system should exhibit; in quantum program testing, a common property is a unitary preservation. **Metamorphic testing** checks relationships between inputs and outputs that should remain consistent under transformations, such as applying and then reversing a unitary operation to recover the original state. **Fuzz testing** involves introducing randomised or edge-case inputs to the system to stress-test its robustness. This technique is widely used in classical computing to identify vulnerabilities, and it is equally valuable in the quantum computing domain. Fuzz testing helps uncover corner cases that might expose weaknesses in the system's resilience or error-correction mechanisms. Fuzz testing can be extended with error injection, where controlled faults or noise assess the system's ability to handle errors and maintain stability. Finally, **differential testing** compares outputs from multiple implementations under identical inputs, helping to tackle the oracle problem [4], where it is difficult or impossible to determine the expected output a priori.

While these approaches often overlap, strategically combining them enables a balance between theoretical precision and practical reliability, bridging the gap between formal specifications and real-world implementation.

3.2 Quantum Computing

To maintain conciseness, this section highlights the specific set of gates used in this work as well as the GHZ state. We start with a brief introduction to quantum states and circuits, the fundamental building blocks of quantum computing. In the supplementary materials, we provide further details on quantum states, gates, circuits, unitarity, Hermitian properties, and measurements. Readers interested in a more thorough exploration can refer to [43].

Quantum State. A *quantum bit* (or *qubit*) is the fundamental unit of quantum information. Unlike classical bits, which are either '0' or '1', qubits can exist in a *superposition* of both states. The two fundamental states of a qubit, denoted as $|0\rangle$ and $|1\rangle$, form the *computational basis*. A linear combination, or *superposition* of the basis states, is a state:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{1}$$

where α and β are complex numbers called *amplitude coefficients*. The coefficients encode both the likelihood of observing each state and the *phase* of the quantum system [43]. A system of n qubits

[2]The keywords shown in blue represent Alloy's syntax for declaring these types.

can be described as: $|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle$, where each ψ_i ($1 \leq i \leq n$) represents a single qubit state as given in Equation 1, and $\otimes$ denotes the tensor product. The system's state is no longer a simple tensor product of individual qubits if its qubits exhibit *entanglement*, where two or more qubits are correlated in such a way that the state of one qubit cannot be described independently of the others. In this case, the system is described by a more complex state, reflecting the non-classical correlations between the qubits. The Greenberger–Horne–Zeilinger (GHZ) state, e.g., for 3 qubits takes the form $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$, is a key multi-qubit entangled state [8, 43].

The state space of a system of n qubits grows exponentially, with the state space represented by a 2^n-dimensional vector. While quantum computers can theoretically explore these vast state spaces in parallel, simulation becomes increasingly challenging as n grows due to the exponential scaling of the statevector. Storing and manipulating these large vectors imposes significant computational resource limitations, discussed further in §8.

Quantum Circuits are composed of quantum *gates*, each represented by a unitary matrix that operates on the statevector of one or more qubits. When applied, a gate performs a linear transformation on the statevector, altering the probabilities and phases of the quantum state. These gates are inherently reversible, meaning every operation can be undone by applying its inverse. Quantum gates enable the creation of superpositions, entanglements, and rotations of qubit states, forming the building blocks of quantum computation. Figure 2 illustrates quantum circuits built via Qiskit (top-right) and Cirq (bottom-right). The *depth* of a quantum circuit is the minimum number of sequential layers of gates needed to implement the circuit, where each layer can operate on a set of qubits in parallel.

A unitary gate satisfies the condition $UU^\dagger = U^\dagger U = I$, where $U^\dagger$ is the Hermitian conjugate (or adjoint) of U and I is the identity matrix. A Hermitian matrix equals its conjugate transpose, i.e. $A = A^\dagger$, meaning the matrix is symmetric in the complex sense. For a matrix $A = \{a_{ij}\}$, the Hermitian conjugate $A^\dagger$ is obtained by transposing A (swapping rows with columns) and taking the complex conjugate of each element: $A^\dagger = \overline{\{a_{ji}\}}$. We use unitaries for cross-simulator consistency testing in §6.

Quantum measurements, generally associated with Hermitian operators, are the means by which results are obtained in quantum computing. This process is inherently probabilistic: it collapses the quantum state into a single basis state, making the system's evolution irreversible. For example, measuring the state $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ (Equation 1) in the computational basis yields the outcome $|0\rangle$ with probability $|\alpha|^2$ and $|1\rangle$ with probability $|\beta|^2$. Because a single execution provides only one of these possible outcomes, the full probability distribution must be estimated statistically by executing the circuit multiple times, a process known as applying *shots*. The precision of any estimated observable improves proportionally to the inverse square root of the number of shots. For rigorous testing, an entire experiment of many shots may itself be repeated multiple times (*repeats*) to further aggregate results and reduce statistical error.

In our formal model (§4), we capture a comprehensive set of common quantum gates. This includes single-qubit operations such as the Pauli gates (X, Y, Z)[3], the Hadamard (H) and $\sqrt{X}$ gates, and general rotation gates (R_x, R_y, R_z, and the parametrised $U(\theta, \phi, \gamma)$ gate). We also model the phase shift gate (P) and its specific instances: $S, S^\dagger, T$, and $T^\dagger$.

For multi-qubit interactions, our model includes controlled gates, such as the controlled-NOT (CX) gate and its generalisation, the CU gate, where U can be any of the Pauli gates. Larger multi-qubit

[3]The X, Y, Z gates are also known as the bit-flip, phase-flip, and combined bit/phase-flip gates, respectively, with standard notations $\sigma_x, \sigma_y, \sigma_z$.

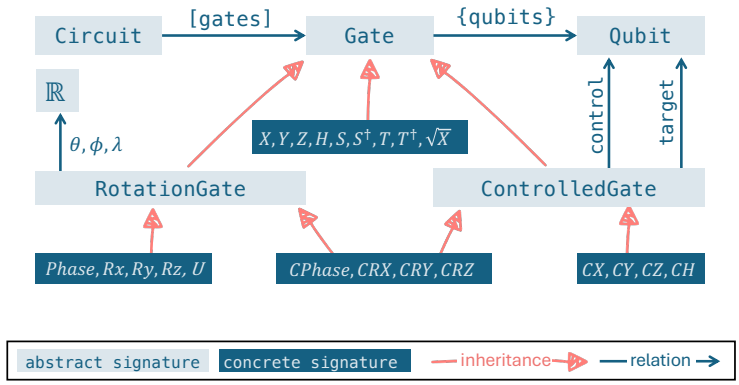


Fig. 3. Hierarchical Structure of Type/Subtype Signatures in Quantum Circuit Modelling with Alloy.

operations can be composed via the tensor product, e.g., $H \otimes H \otimes H$ to apply a Hadamard to three qubits simultaneously. The matrix definitions for these key gates are provided below.

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad S = P\left(\frac{\pi}{2}\right) \quad S^\dagger = P\left(-\frac{\pi}{2}\right) \quad CU = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{11} & u_{12} \\ 0 & 0 & u_{21} & u_{22} \end{bmatrix} \quad R_x(\theta) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}$$

$$Y \equiv \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad T = P\left(\frac{\pi}{4}\right) \quad T^\dagger = P\left(-\frac{\pi}{4}\right) \quad CPhase = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \quad \sqrt{X} = \begin{bmatrix} \frac{1}{2}(1+i) & \frac{1}{2}(1-i) \\ \frac{1}{2}(1-i) & \frac{1}{2}(1+i) \end{bmatrix}$$

$$Z \equiv \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad H \equiv \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad P(\phi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix} \quad U(\theta, \phi, \gamma) = R_z(\phi) \cdot R_y(\theta) \cdot R_z(\gamma)$$

4 Quantum Circuit Formal Model

We developed a formal model of quantum gates in Alloy [27] to systematically generate diverse quantum circuits. As a declarative modeling language, Alloy is well-suited for exploring and generating randomised instances of complex systems that adhere to specified constraints, as depicted in our modeling hierarchy (Figure 3). This approach allows for a comprehensive exploration of the quantum operations space while guaranteeing the structural validity of the generated circuits. The overall high-level process is illustrated in Figure 1.

We derive a structured hierarchy of Alloy signatures from the QASM 3.0 [15] specification for modelling quantum circuits. Figure 3 depicts the structure we used in Alloy to model the hierarchy of quantum circuit elements, including different gate types, qubits, and the circuit itself. The model defines a set of signatures representing quantum components, establishing relationships between them. For example, we represent a quantum gate abstractly through the `Gate` signature, which can operate on one or more qubits, as indicated by the `qubits` relationship. This signature serves as a parent for various quantum gate types, including common gates like `X`, `H`, and more specialised gates such as rotation and controlled gates. Additionally, the abstract `Circuit` signature aggregates a sequence of gates, defining the overall architecture of the quantum circuit.

Example 1 (Example of Alloy Circuit Modelling). Figure 4 presents a toy Alloy model for a quantum circuit with `H` and `CX` gates. In the model, signatures (**`sig`**) define types like `Qubit`, abstract `Gates`, and a `Circuit`. Concrete gates (e.g. `HGate`, `CXGate`) extend the abstract ones. Constraints (**`facts`**) enforce structural rules like qubit connectivity and gate arity. The **`run`** command instructs the Alloy Analyzer to find an instance satisfying these rules, which for this example corresponds to the circuit diagram shown in Figure 2.

```
sig Qubit {}
abstract sig Gate { qubits: set Qubit }
one abstract sig Circuit { gates: seq Gate }
sig HGate extends Gate {}
abstract sig ControlledGate extends Gate {
    control, target: one Qubit,
} {
    qubits = control + target
    control != target
}
sig CXGate extends ControlledGate { }
fact CircuitConstraints {
    all g: Gate | g in Circuit.gates.elems  => some g.qubits  // Gates act on qubits
    all g: (HGate) | #g.qubits = 1 // Single-qubit gates connect to one qubit
    all q: Qubit | some g: Gate | q in g.qubits // All qubits are used by some gate
    Circuit.gates.elems = Gate // Every gate is fully integrated into the circuit
}
run {} for exactly 2 Qubit, 2 Gate
```

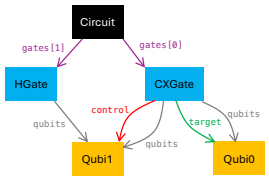
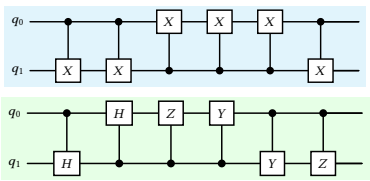


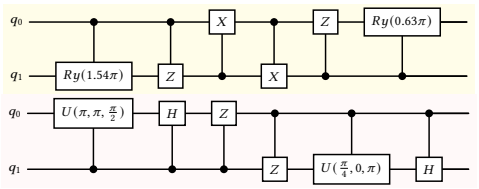
Fig. 4. A toy quantum circuit modelled in Alloy with one possible instance visualised in *Alloy Analyser*.

A key feature of Alloy is its ability to be strategically guided by constraints to generate specific configuration patterns. For example, to enforce symmetric interactions between qubits, we can add a constraint requiring that every controlled gate is matched by a corresponding gate in the reverse direction (e.g., from control to target, and target to control). This ensures the generation of circuits with exact, bidirectional gate pairs. The following Alloy code implements this symmetry constraint:

```
run { // For every controlled gate, a corresponding gate exists in reverse order
      all g: ControlledGate | some gRev: ControlledGate |
          gRev.control = g.target and gRev.target = g.control
    }
for exactly 4 Qubit, exactly 6 Gate, 6 seq, 4 int
```

Example 2 (Symmetric Quantum Circuits Generated by Alloy)**.** Below are several quantum circuits generated by FuzzQ's Alloy model that satisfy the symmetry constraint discussed previously. In these examples, each controlled gate (e.g. CX, CY, CZ, and controlled rotations/unitaries) is mirrored by a corresponding reverse gate, creating a balanced and symmetric circuit topology.





5 Invariant-Based Testing

As a preliminary step before full differential testing, we employ invariant-based testing. This serves the dual purpose of validating our Alloy model's interpretation of quantum specifications and acting as an early bug-detection mechanism. The approach verifies that specific physical or logical properties (e.g., symmetries, self-inverse operations) hold across computations. To this end, we use FuzzQ to generate circuit seeds that are structurally guaranteed to exhibit a given invariant, such as the symmetric circuits in Example 2. A subsequent violation of the invariant during simulation thus signals either a need for model refinement or a potential bug in the simulator itself. Below, we outline three such invariants explored in our work.

Involutory Circuits. Certain quantum gates, like X, H, and CX, exhibit self-inverse behaviour, meaning applying the gate twice restores the qubit to its original state ($U^2 = I$). Using Alloy, we

can systematically construct circuits that demonstrate this involutory property, whether through a single self-inverse gate or a sequence of gates that collectively exhibit this property.

Symmetry Preservation. In quantum mechanics, symmetries correspond to conserved physical properties. Our invariant tests verify that simulators correctly preserve these symmetries during transformations. For example, a quantum state that is symmetric under rotation around the z-axis, such as $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, must remain so after valid operations. Mathematically, an operation U preserves this symmetry if it commutes with the rotation operator $R_z(\theta)$, satisfying $UR_z(\theta) = R_z(\theta)U$. In a test case, we can verify that applying an H gate followed by a Z gate to a basis state correctly prepares and maintains such a z-axis rotational symmetry, ensuring the simulator adheres to this physical principle.

Circuit-Specific Invariants. Certain recognisable invariants are expected to hold for specific quantum circuits, regardless of the simulator used, such as in the Greenberger-Horne-Zeilinger (GHZ) state. A GHZ state is a maximally entangled quantum state. This means that measuring one qubit instantaneously determines the state of all others, regardless of their spatial separation, exemplifying quantum nonlocality. For an n-qubit GHZ state, $|GHZ\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle^{\otimes n} + |1\rangle^{\otimes n}\right)$, measurement outcomes will always yield either all zeros or all ones, demonstrating perfect correlation. For instance, a 3-qubit GHZ state, $|GHZ\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$, results in either $|000\rangle$ or $|111\rangle$ upon measurement. This perfect correlation reflects the symmetry and predictability inherent in entangled quantum states, highlighting the unique nature of quantum measurement outcomes.

6 FuzzQ: Fuzzing-Based Benchmarking of Quantum Circuits

Building on our formal model (§4) and its initial validation (§5), we now detail our fuzzing-based methodology for evaluating the accuracy and consistency of quantum simulators. Our approach uses the Alloy model as a guided input engine for a rigorous benchmarking strategy that systematically assesses simulator behaviour across a wide range of scenarios. The process consists of four key stages:

(1) *Formal Model Construction and Validation:* Using and refining the formal model as discussed in §4 and §5.
(2) *Test Case Generation:* Generating random, constraint-guided quantum circuits designed to expose subtle behaviours (§6.1).
(3) *Differential Testing:* Executing the generated circuits on multiple simulators and assessing them against various test criteria (§6.2).
(4) *Discrepancy Validation and Classification:* Identifying and classifying discrepancies found across a broad range of circuit types and complexities.

The following subsections detail stages 2-4.

6.1 Test Case Generation via Random Synthesis and Guided Mutation

Our test generation employs a two-pronged, adaptive fuzzing approach: *random synthesis* for broad exploration, complemented by *guided mutation* for targeted refinement of interesting cases.

Random Synthesis. The synthesis phase uses the Alloy Analyser to generate a diverse corpus of circuit seeds. These seeds are not complete circuits but rather structural blueprints, specified as instances of our formal model (§4). Each seed defines a circuit's fundamental topology: the number of qubits, the sequence of gate *types* (e.g. H, CX), and their connectivity. The process is guided by constraints within the Alloy model, which allows for an implicit classification of test cases based on the properties enforced. For example, the symmetry constraint shown in Example 2 directs Alloy to generate a specific class of circuits exhibiting that structure, enabling targeted exploration.

These abstract seeds are then "fleshed out" into simulable circuits by assigning concrete values to numerical gate parameters (e.g. rotation angles). We use three parameterisation strategies:

- *Random Sampling* of values within valid parameter ranges.
- *Edge Cases & Critical Values*, such as standard angles $(0, \pi, 2\pi)$, near-zero perturbations ($\approx 10^{-10}$), extreme values (infinities, bit shifts), Rabi frequency rotations ($\theta = \Omega t$), and QFT angle discretisations ($\theta = 2\pi/2^k$).
- *Small Perturbations* from standard values to identify rounding errors and numerical drift.

These fully parametrised circuits are then executed across multiple simulators to establish a baseline for consistency, with any discrepancies flagged for guided mutation.

Guided Mutation. This phase takes discrepant circuits from synthesis and applies targeted modifications to intensify inconsistencies and pinpoint failure conditions. Instead of generating entirely new cases, it refines existing ones via:

- *Parameter Tweaks:* Incremental adjustments to gate parameters like rotation angles.
- *Structural Modifications:* Minor alterations to circuit topology, such as rearranging gates.
- *Repetition and Scaling:* Expanding the circuit with more qubits or gate layers to test for size-sensitive errors.

Following each mutation, circuits are re-simulated and results are compared to baselines. This creates an iterative feedback loop, shifting the process from broad, random exploration to a structured, investigative one. This overall approach, which can be described as *seeded mutation fuzzing*, enhances test diversity and efficiency by focusing resources on high-impact cases and supporting continuous refinement.

6.2 Test Criteria

To assess quantum simulators effectively, we establish rigorous test criteria that capture essential quantum circuit behaviours. Since determining exact outcomes of generated test circuits is often computationally intractable, we employ *differential testing*. This approach cross-validates simulator outputs by executing identical input circuit seeds across multiple platforms and systematically comparing their results. Rather than relying on external oracles, we evaluate intrinsic quantum properties – such as statevector evolution and probability distributions – to identify inconsistencies. Below, we detail three core aspects of our evaluation methodology.

6.2.1 Gate-by-Gate Fidelity. This method evaluates the consistency of quantum state evolution by tracking statevector deviations after each gate operation, ensuring simulators produce equivalent intermediate states throughout the simulation. For a set of N simulators, let ψ_i denote the statevector produced by simulator i, where $i = 1, 2, \ldots, N$. The difference between any two simulators' statevectors should be bounded by a tolerance ϵ:

$$\|\psi_i - \psi_j\|_2 \leq \epsilon \quad \forall\, i, j \in \{1, 2, \ldots, N\},\ i \neq j. \tag{2}$$

When this condition is met, the simulators demonstrate agreement in correctly reproducing the expected quantum state evolution. However, direct statevector comparison (Equation 2) is sensitive to global phase differences—physically meaningless variations[4] that can introduce artificial discrepancies even when simulators produce equivalent physical states. To address this limitation, we use *fidelity* – a more robust metric that quantifies the "closeness" of quantum states while accounting for small numerical deviations and ignoring global phases. For pure states, the fidelity F between statevectors ψ_i and ψ_j is defined as:

$$F(\psi_i, \psi_j) = |\langle \psi_i | \psi_j \rangle|^2 \quad \forall\, i, j \in \{1, 2, \ldots, N\}. \tag{3}$$

[4]Global phase differences do not affect measurement probabilities or observable properties of quantum states.

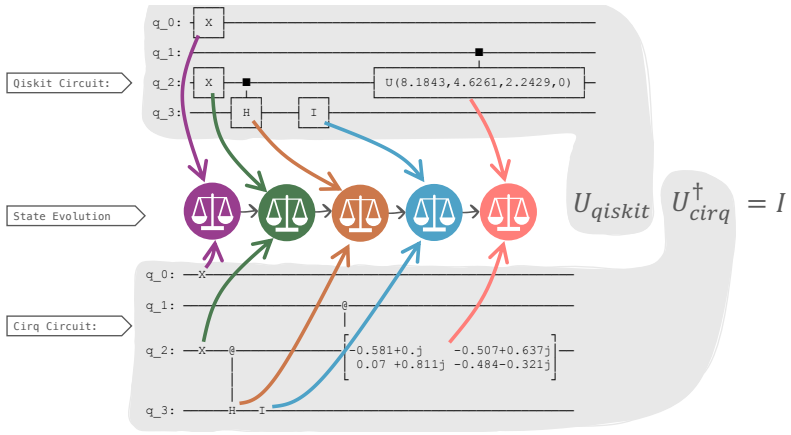


Fig. 5. Gate-by-Gate state fidelity evolution between Qiskit and Cirq simulators, with final Unitary Parity Benchmark application.

Fidelity values range from 0 (completely orthogonal states) to 1 (identical states), providing a more tolerant metric for comparison. By establishing an acceptable threshold (e.g. $F \geq 0.99$), we ensure simulators produce sufficiently similar quantum states despite potential minor numerical discrepancies arising from floating-point limitations.
The gate-by-gate fidelity comparison enables robust validation of quantum state similarity between simulators. Figure 5 illustrates this approach with a gate-by-gate fidelity evaluation between Qiskit and Cirq implementations, tracking a test circuit through five distinct intermediate quantum state evolution points.

6.2.2 Probabilistic Consistency of Measurement Distributions. Our evaluation assesses two critical aspects of simulator performance: alignment with theoretical expectations and cross-simulator consistency in measurement outcomes. To quantify these, we employ two complementary statistical methods:
The *Chi-Square Goodness-of-Fit test* (χ^2) evaluates individual simulator accuracy. It compares observed measurement counts to expected probabilities derived from the final statevector's theoretical distribution (§3.2). The resulting p-value represents the probability of obtaining the observed test statistic (or more extreme) if the null hypothesis – that observed and expected distributions match – is true. A low p-value ($< \alpha$, typically 0.05) indicates a statistically significant deviation between measured outcomes and theoretical predictions, leading us to reject the null hypothesis; larger p-values suggest consistency with theoretical predictions.
The *Jensen-Shannon divergence* (JSD) quantifies cross-simulator consistency by measuring similarity between output distributions. This metric ranges from 0 (identical distributions) to 1 (completely different), allowing us to detect subtle variations in how different simulators process identical quantum circuits. [61, 62].
Together, these metrics provide a comprehensive statistical framework for evaluating both the absolute accuracy and relative consistency of quantum simulators.

6.2.3 Cross-Simulator Unitary Consistency. To detect subtle semantic differences in how simulators realise the same logical gate (a key challenge in cross-platform validation), we developed the Unitary Parity Benchmark. This method verifies that different quantum simulators implement unitary operations consistently. We examine whether the matrix product $U_i U_j^{\dagger}$ is proportional to the identity matrix I, where U_i and U_j represent the unitary transformations from different simulators (e.g. Qiskit and Cirq, respectively). This equivalence relation, invariant to global phase

differences, confirms that both simulators faithfully implement the same quantum transformation. The theoretical foundation for this approach is formalised as follows:

Theorem 1 (Unitary Parity Benchmark in Quantum Simulation)**.** *Let $S_1, S_2, \ldots, S_n$ be a collection of quantum simulators each implementing the same quantum circuit C on identical initial states. Each simulator produces a unitary matrix U_i describing the quantum state evolution under the circuit. The unitaries U_i and U_j from different simulators are equivalent up to a global phase if and only if there exists a phase factor $e^{i\phi}$ such that $U_i U_j^\dagger = e^{i\phi} I$.*

Proof. We prove the equivalence in two directions:
($\Rightarrow$) If U_i and U_j are equivalent up to a global phase, then $U_i = e^{i\phi} U_j$ for some phase ϕ. Multiplying both sides on the right by $U_j^\dagger$:

$$\begin{aligned} U_i U_j^\dagger &= e^{i\phi} U_j U_j^\dagger \\ &= e^{i\phi} I \end{aligned}$$

since U_j is unitary, implying $U_j U_j^\dagger = I$.
($\Leftarrow$) Conversely, suppose $U_i U_j^\dagger = e^{i\phi} I$ for some phase ϕ. Multiplying both sides on the right by U_j, we have:

$$\begin{aligned} U_i U_j^\dagger U_j &= e^{i\phi} I U_j \\ U_i &= e^{i\phi} U_j \end{aligned}$$

This establishes that U_i and U_j are equivalent up to the global phase factor $e^{i\phi}$. Since global phases have no observable effect on quantum measurements, the unitaries U_i and U_j represent physically equivalent quantum operations. □

This theorem underpins our *equivalence oracle* used in cross-simulator validation. The oracle examines whether the product $U_i U_j^\dagger$ is proportional to the identity matrix by checking if all off-diagonal elements are approximately zero and all diagonal elements have approximately equal magnitude. Any significant deviation from this condition reveals meaningful implementation differences between simulators.

In practice, we compute the normalised quantity $\frac{\|U_i U_j^\dagger - e^{i\phi} I\|_F}{\sqrt{d}}$ where d is the dimension of the matrices, $\|\cdot\|_F$ is the Frobenius norm, and ϕ is chosen to minimise this expression. This approach provides a robust measure of unitary equivalence that accounts for both the global phase and numerical precision limitations.

Limitations of Differential Testing. A core assumption of our differential testing approach is that independently developed simulators are unlikely to fail in the exact same way. However, if all simulators under test were to produce an identical, incorrect result for a novel circuit, the differential oracles (fidelity, JSD, Unitary Parity) would not detect the error. To mitigate this risk, our overall methodology does not rely solely on differential testing. We complement it with property-based and invariant-based testing (§5) and validate against circuits with analytically known outcomes (§7), providing an absolute correctness check for certain classes of circuits.

7 Threats to Validity

While our validation approach was carefully designed for robustness, several concerns must be acknowledged.

The accuracy and completeness of our tool are crucial; inaccuracies in representing quantum circuits can lead to discrepancies between simulated and expected outcomes. Additionally, inherent simulator limitations (differences in implementation, numerical precision, and performance) can create discrepancies potentially misattributed to our framework. Another concern is overfitting

risk, where models performing well on specific circuits may struggle to generalise across a wider range of quantum operations, leading to potentially misleading conclusions. Test case selection plays a vital role; limited test sets may overlook significant issues in both model and simulators. As circuit complexity increases, scalability becomes a pressing concern, making comprehensive testing impractical and allowing errors to persist undetected. Furthermore, variations in noise and error rates across simulators can complicate evaluation, potentially affecting simulation accuracy. It is important to note that our current study primarily evaluates simulators in the context of pure-state evolution, where noise and decoherence effects are ideally absent; a detailed discussion of the implications for mixed states and noisy simulations is presented in §10. Finally, the probabilistic nature of quantum measurements requires careful statistical analysis to avoid misinterpreting discrepancies between expected and observed outcomes.

These challenges underscore the need for rigorous validation and raise a fundamental question: *What if the framework itself is flawed?*

We implement four strategies to mitigate this risk and strengthen internal validity.

Dual-layered Simulator Alignment Checks. We combine gate-by-gate fidelity (see §6.2.1) with final unitary equivalence (see §6.2.3) to establish a robust validation framework. To motivate the necessity of the latter, more rigorous check, the following example and lemma illustrate a subtle discrepancy where naive statevector comparison can be misleading, underscoring a potential flaw that our Unitary Parity Benchmark is designed to avoid.

Example 3. Consider a quantum circuit where the *CX* gate's implementation varies between Qiskit and Cirq. In the Qiskit implementation, the standard *CX* gate operates with a control qubit determining the target qubit's flip. Conversely, the Cirq implementation inadvertently swaps the control and target qubit roles.

Lemma 1 (Deceptive Equivalence)**.** *Two quantum circuits may produce identical output for a specific input state while fundamentally differing in their underlying transformation.*

Proof. Matrix analysis reveals the implementation asymmetry:

$$U_{\text{Qiskit}} \cdot U_{\text{Cirq}}^{\dagger} = \underbrace{\begin{bmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{bmatrix}}_{CX} \cdot \underbrace{\begin{bmatrix} 1&0&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \\ 0&1&0&0 \end{bmatrix}^{\dagger}}_{CX_{reversed}} = \begin{bmatrix} 1&0&0&0 \\ 0&0&0&1 \\ 0&1&0&0 \\ 0&0&1&0 \end{bmatrix} \tag{4}$$

For the initial state $|00\rangle$, both implementations leave $|00\rangle$ unchanged, trivially passing a naive statevector comparison. However, the unitary product test exposes a non-trivial transformation that differs from the identity matrix. □

Many quantum algorithms initialise from $|00\ldots0\rangle$, making such subtle implementation discrepancies particularly elusive.

Grounding in Known and Real-World Circuits. To further ground our model and testing, we use two complementary approaches. First, we validate against fundamental circuits with well-established theoretical outcomes, such as Bell and GHZ states, as detailed in our invariant testing (§5). Second, we encode externally sourced, real-world quantum circuits as predicates in our Alloy model. Verifying that the model can find satisfying instances for these predicates ensures our framework remains grounded in practical applications.

Iterative Model Validation and Refinement. We do not assume our initial Alloy model is perfect. Instead, we gain confidence in its correctness and systematic nature through a multi-faceted validation process inspired by CEGAR-style abstraction refinement.

- *Specification-Driven Modelling:* The model's hierarchy and signatures (§4) are systematically derived from the OpenQASM 3.0 specification [15], ensuring it is grounded in an established standard.
- *Internal Consistency Checking:* The Alloy Analyser itself helps find logical inconsistencies or underspecified aspects within our model during development.
- *Invariant-Based Validation:* As detailed in §5, we use the model to generate circuits that must satisfy known physical invariants. If a simulator correctly adheres to an invariant, but the model had difficulty expressing it, it signals a need for model refinement.
- *Feedback Loop with Differential Testing:* The core process (Figure 1) serves as a continuous validation loop. Discrepancies traced back to an oversimplification in our model act as counterexamples to refine its specification, increasing its fidelity over time.

This iterative process strengthens our confidence in the model's accuracy and its systematic representation of quantum circuits.

8 Evaluation

We aim to address the following research questions in our evaluation:

RQ_1 *How efficiently and comprehensively can* `FuzzQ` *generate test cases?* We investigate Alloy's generation throughput and then analyse the structural and code-level coverage of the generated test suite to assess its thoroughness.

RQ_2 *Is this comprehensive test suite effective in detecting bugs?* Using the diverse set of circuits generated, we conduct large-scale experiments on a CloudLab machine [19] to assess the bug detection capability of our approach. Priorly, we needed to establish a clear definition of what constitutes a bug. To achieve this, we further ask RQ3 and RQ4.

RQ_3 *How many shots are required to get a reliable quantum output?* We determine the optimal number of shots for simulating the quantum circuit by measuring the closeness of probability distributions produced by Qiskit and Cirq simulators using the Jensen-Shannon Divergence (JSD).

RQ_4 *What is the accuracy of the simulator in reproducing expected results?* We assess the accuracy of Qiskit and Cirq simulators using χ^2 tests and JSD metrics to determine precision sensitivity in benchmarking, ensuring that only meaningful bugs are reported to developers.

RQ_5 *What are the scalability and stability limitations of quantum test case execution?* We examine the scalability of quantum simulation with increasing qubit count, focusing on memory consumption and numerical stability.

8.1 RQ_1: Test Case Generation and Coverage

Throughput. The Alloy Analyser generates quantum circuit instances through a two-phase process: **1) Constraint Processing** – translating the model into a propositional formula with necessary variables and clauses; and **2) Solving** – using a SAT solver to find satisfying assignments that meet constraints. For a 12-qubit, 10-gate circuit, translation took 139 ms and solving required 226 ms. To assess throughput, we measured generation rates using Alloy in command-line mode, which significantly outperforms the GUI. This approach produced ~41,000 12-qubit, 10-gate circuits in 5 minutes (140 circuits/second). Throughput varies with model complexity (qubit count, circuit depth, constraints), SAT solver selection, and hardware specifications.

Quantum Platforms Code Coverage. To evaluate the quantum platforms' code coverage, we instrumented Qiskit `2.2.0.dev0+6b4477f` libraries under Python version `3.11.13+` and `rust-1.79.0stable`, and Cirq `1.6.0.dev0` libraries under Python version `3.11.13+`.

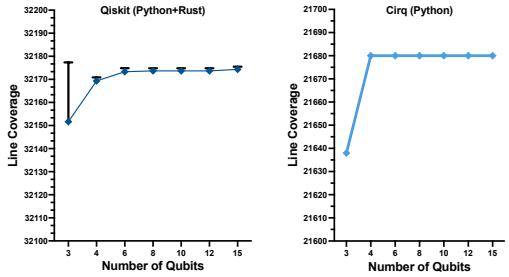


Fig. 6. Line code coverage across qubit sizes on Cirq and Qiskit backends using FuzzQ-generated circuits. Black error bars: standard deviation. Cirq's standard deviation is minimal and not visible in the plot.

We sampled 500 XML-based circuit seeds uniformly at random from the full generated population (as in the throughput paragraph above). Each XML seed was fuzzed 5 times. Coverage was measured using `coverage.py 7.9.2` for Python and for Rust components `llvm-cov 18.1.7-rust-1.79.0-stable`. This process was repeated independently 5 times, resulting in 5×500×5 simulable quantum circuits per platform. Reported results reflect the average across the five runs.

We measured coverage by progressively accumulating qubit sizes, starting from size 3, then including sizes 4, 6, 8, 10, 12, and 15, with each step incorporating all smaller sizes. Figure 6 presents the line code coverage for Qiskit (left) and Cirq (right). Qiskit achieved a higher average code coverage of **32,174.6 lines** when combining Python (23,453 lines) and Rust (8,721.6 lines) components. Cirq achieved an average code coverage of 21,680 lines.

The line coverage results in Figure 6 show a rapid early increase followed by saturation across both platforms. For Cirq, coverage rises from 21,638 lines at qubit size 3 to 21,680 at size 4, and then remains flat through size 15, indicating that small circuits are sufficient to fully activate relevant code paths. Qiskit displays a similar trend but with higher overall coverage, saturating around 6 qubits. The black error bars highlight higher standard deviation at low qubit sizes in Qiskit, likely because smaller circuits exercise less code than 8+ qubit circuits, while Cirq shows negligible variation. This trend suggests that FuzzQ is highly effective at achieving structural and parametric diversity early, generating compact circuits that already trigger much of the platform's behaviour. To further investigate this, we propose a second coverage metric focused on circuit diversity.

Circuit Coverage. To assess the effectiveness of our generated test cases in detecting bugs, we first analysed the comprehensiveness of our test suite by evaluating a set of quantum-specific structural coverage metrics. Unlike classical code coverage, which is inadequate for this domain, our metrics assess the diversity of quantum operations and circuit topologies generated by FuzzQ. Our analysis was performed on a set of 875 circuits randomly selected from the larger corpus of over 800,000 generated for our full evaluation. This sample, scaled from 3 to 15 qubits and comprising over 10,750 gate operations, confirms that our overall approach achieves high coverage:

- *Gate Type Diversity:* We achieved 100% coverage of the 30 distinct gate types defined in our Alloy model, including a sophisticated mix of advanced multi-qubit gates (e.g. 19.3% controlled-U, 16.5% SWAP, 7.6% controlled-SWAP).
- *Structural Uniqueness:* Of the 875 circuits, 758 (86.6%) possessed unique structural topologies, exploring a broad set of computational structures and 128 unique qubit interaction patterns.
- *Two-Level Architecture Validation:* Our separation-of-concerns approach was empirically validated: Alloy-generated XML templates provide comprehensive structural coverage while runtime parameter generation ensures computational diversity, achieving superior coverage compared to monolithic approaches.

Table 1. Summary of reported issues in Qiskit and Cirq simulators

#	Issue Description
1	**SWAP Gate Ineffectiveness due to Transpiler Remapping (Qiskit):** The logical effect of a SWAP gate can be nullified in the final statevector under specific circuit structures due to dynamic qubit remapping during the transpiler's layout pass, which is not correctly reflected in all simulation paths [52].
2	**Unpredictable Handling of `seed_simulator` (Qiskit Aer):** Qiskit Aer accepts undocumented input types (e.g. strings, booleans, floats) for `seed_simulator`[1], resulting in inconsistent randomisation behaviour [53].
3	**Inconsistent Statevectors for Fixed Seeds (Qiskit Aer):** Using the same seed value does not always yield identical statevectors in Qiskit Aer's statevector simulator, contradicting expected deterministic behaviour [54].
4	**Misleading Errors for Invalid Qubit Indices (Qiskit):** Assigning extreme values (e.g. -1, 10^9) to `initial_layout`[2] produces vague error messages, making debugging difficult [51].
5	**`MatrixGate` Does Not Support Symbolic Parameters (Cirq):** Unlike other Cirq gates, `MatrixGate` does not support symbolic parameters (e.g. `sympy.Symbol`), limiting its use in variational algorithms [12].
6	**Numerical Instability with Large Angles (Cirq):** Using extremely large or small angles (e.g. π^{100}) in Cirq's unitary gates can result in non-unitary matrices, violating quantum gate constraints [13].
7	**Transpilation Failure for Specific Unitaries (Qiskit):** Qiskit's transpiler fails with a compilation error when the `TwoQubitWeylDecomposition` is applied to certain classes of 2-qubit unitary matrices, particularly those with specific complex phase relationships between their elements, preventing circuit execution [55].
8	**NumPy's `einsum` Index Overflow in Large Quantum Circuits:** Simulating 16+ qubits circuits may exceed NumPy's limit of 32 unique indices in `einsum` operations, leading to failures [44, 45, 47]. Using `opt_einsum`[4] resolves this.

[1] `seed_simulator` is a parameter in Qiskit's Aer simulator backend that sets the seed for the internal random number generator. This ensures reproducibility in simulations involving probabilistic processes, such as measurement sampling and noise models.
[2] The `initial_layout` parameter in Qiskit allows users to specify a mapping of logical qubits to physical qubits on a quantum device.
[3] The `TwoQubitWeylDecomposition` is an internal Qiskit function used for decomposing two-qubit gates into a sequence of simpler gates based on the Weyl decomposition.
[4] `opt_einsum` is a more efficient implementation of the `einsum` function in NumPy (Numerical Python), optimised for high-dimensional tensor contraction.

This extensive structural and parametric coverage, validated through actual simulator execution rather than theoretical analysis, provides a robust foundation for bug finding, giving high confidence that the discrepancies we find are genuine simulator issues, not artifacts of a biased test suite.

Answer to *RQ_1:* FuzzQ can efficiently and comprehensively generate test cases. Alloy provides high-throughput circuit generation on standard hardware, while the generated test cases exhibit extensive structural and parametric coverage. For code coverage, saturation was reached early (between 4 and 6 qubits), demonstrating the effectiveness of our formal-methods-driven approach in exploring a wide range of simulator behaviours with compact circuits.

8.2 RQ_2: Bug Finding and Testing

Throughout our validation campaign, we identified and reported several key issues within both Qiskit and Cirq simulators, summarised in Table 1. While some of these issues may appear straightforward in retrospect, their persistence in mature, widely-used simulation platforms highlights the limitations of conventional testing. FuzzQ's strength lies in its systematic, constraint-guided exploration of circuit structures and parameter spaces, which can uncover subtle interaction bugs that ad-hoc testing might miss.

For instance, *Issue 1 (Unexpected Behaviour in Qiskit's SWAP Gate)* exposes a complex interaction with the transpiler's dynamic qubit remapping for layout optimisation. Under certain circuit structures generated by FuzzQ, the transpiler would alter the physical qubit mapping in a way that effectively nullified a subsequent SWAP gate's intended logical effect on the final statevector. This type of bug, where the outcome depends on the interplay between circuit structure, transpilation

passes, and the final simulation method, is particularly well-suited for discovery through the systematic and differential nature of our fuzzing framework. We note that 6 of the 8 bug types discovered were previously undocumented. We have provided a description and reproducer for this illustrative issue in our accompanying artifact (see §12).

The issues found stemmed from a range of causes, including inconsistencies in statevector outputs, unexpected behaviours with quantum gate operations, and challenges with input handling. In some cases, simulators exhibited errors due to extreme or unconventional qubit index values, as well as limitations in handling symbolic parameters or large angle values. These findings underscore the potential impact of such bugs on the reliability of quantum simulations.

Answer to RQ_2***:*** Yes, our approach proved highly effective, leading to the discovery of eight distinct bug types across Qiskit and Cirq, six of which were previously undocumented.

8.3 RQ_3: Determining the Optimal Number of Shots

To determine an optimal number of shots for reliable measurement outcomes, we analysed the Jensen-Shannon Divergence (JSD) between the output probability distributions of Qiskit and Cirq. Our analysis used a 10-qubit circuit, with shot counts increasing from 10 to 10^6, and each data point repeated 1,000 times for statistical robustness. We observed consistent trends across circuits of varying sizes, underscoring the generality of our findings.

As shown in Figure 7a, and in line with the law of large numbers, the mean JSD and its variance both decrease as the number of shots increases. This reflects a reduction in statistical noise and more stable, reliable measurement outcomes. Upon visual and statistical inspection, it is evident that a clear plateau in variance reduction is reached at 1,000 shots (represented by the dash-dot segment). To formalise this, we define the stabilisation point as the shot count where the JSD variance first drops below a threshold of 10^{-3}, which corresponds to this plateau.

Based on this analysis, we identified 1,000 shots as the point of stabilisation, where further increases yield only marginal improvements in variance. However, to ensure maximum robustness and minimise any residual fluctuation in our subsequent experiments, we conservatively chose to use **10,000 shots** for all validation tasks.

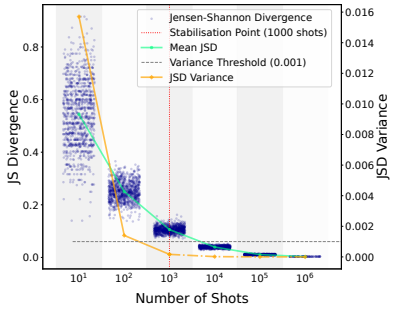
Answer to RQ_3***:*** Our analysis shows that the JSD between simulator outputs stabilises with a minimum of 1,000 shots. To ensure robustness, we conservatively selected 10,000 shots for all subsequent experiments and recommend this threshold for developers conducting testing in simulation, as it balances statistical reliability with computational overhead.

8.4 RQ_4: Assessing Simulator Accuracy

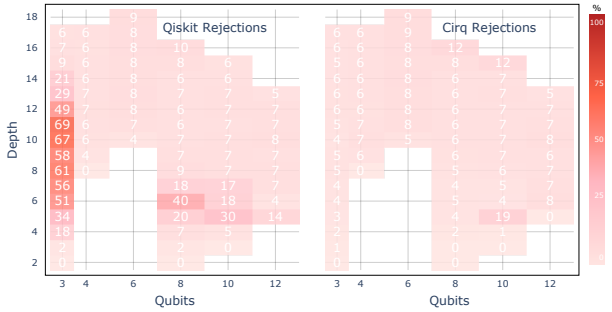
(I) To evaluate simulator accuracy against theoretical predictions, we analysed χ^2 test results. These tests generate p-values indicating whether observed probabilities significantly differ from expected ones. When $p < \alpha$ (typically 0.05), the null hypothesis is rejected. We computed rejection rates across different qubit counts and circuit depths, with shots fixed at 10,000.

As shown in Figure 7b, we observe no significant trends. Qiskit and Cirq demonstrate comparable behaviour overall, with one notable exception: Qiskit exhibits a distinct spike in p-values between

[5]Jittering is applied to spread the points horizontally within each shots-region while preserving the original JSD (y-values). This technique helps visualise the distribution of the data by reducing point overlap, allowing for clearer differentiation between measurements. The primary focus remains on the vertical values (the JSD values), as they are the key data of interest. The shaded regions serve solely to visually segment and clearly differentiate between various shot ranges.



(a) Jensen-Shannon Divergence between Qiskit and Cirq (with horizontal jittering[5])



(b) χ^2 Test Rejections: Coloured areas indicate tested qubit-depth configurations, with colour intensity indicating null hypothesis rejection rates

Fig. 7. Comparison of Jensen-Shannon Divergence (JSD) and chi-squared (χ^2) Test Rejections between Qiskit and Cirq simulations

circuit depths 6 and 12 with 3 qubits, reaching values as high as 69% – significantly deviating from the otherwise consistent pattern.

(II) To provide further insight, we introduced a new metric, the *Simulated Performance Score* (SPS). SPS aggregates the proximity to 1 across all experiments, providing a single value that reflects the simulator's performance based on p-values. The SPS for each simulator was defined as the mean proximity to 1 across all experiments: SPS $= \frac{1}{N}\sum_{i=1}^{N}(1-p_i)$, where: N was the number of experiments, p_i was the p-value for experiment i (for either Qiskit or Cirq), and $1-p_i$ was the proximity to 1 for each experiment. The closer the SPS was to 0, the better the simulator performed in terms of aligning with the expected results. After computing the SPS for a set of approximately 600k experiments, Qiskit had an SPS of 0.4657, while Cirq performed marginally better with an SPS of 0.4463.

(III) We applied polynomial regression to explore the relationship between SPS, the number of qubits, and circuit depth for Qiskit and Cirq. The results showed limited predictive power, with R^2 (coefficient of determination) values of 0.063 for Qiskit and 0.081 for Cirq. R^2 indicates the proportion of variance in SPS explained by the regression model, with these values suggesting it explains only a small fraction of the variance. Additionally, the Mean Squared Error (MSE) was 0.106 for Qiskit and 0.096 for Cirq, with Cirq showing slightly lower prediction errors.

Figure 8 complements the regression analysis by visualising SPS trends across the number of qubits and circuit depth. Both factors exhibit weak correlations with SPS, though circuit depth has a marginally stronger effect. As expectedly, greater depth introduces complexities such as increased entanglement, interference effects, and numerical instability, all of which amplify the computational effort needed for accurate simulation, especially as the state space grows exponentially. It highlights the increasingly demanding and less predictable nature of simulations with greater depth.

(IV) We analysed the *Jensen-Shannon Divergence* (JSD) between the Qiskit and Cirq simulators, finding a mean value of 0.0109 across 600k simulations, indicating reasonable alignment. The influence of the number of qubits and circuit depth on the JSD is minimal, as shown in Figure 8. Polynomial regression confirms this, with a low R^2 value of 0.1488, suggesting that the model explains only 14.88% of the variance. Although there is a slight positive relationship between the number of qubits, depth, and JSD, the overall fit is weak, and the regression coefficients and interaction terms suggest a minimal, nonlinear influence.

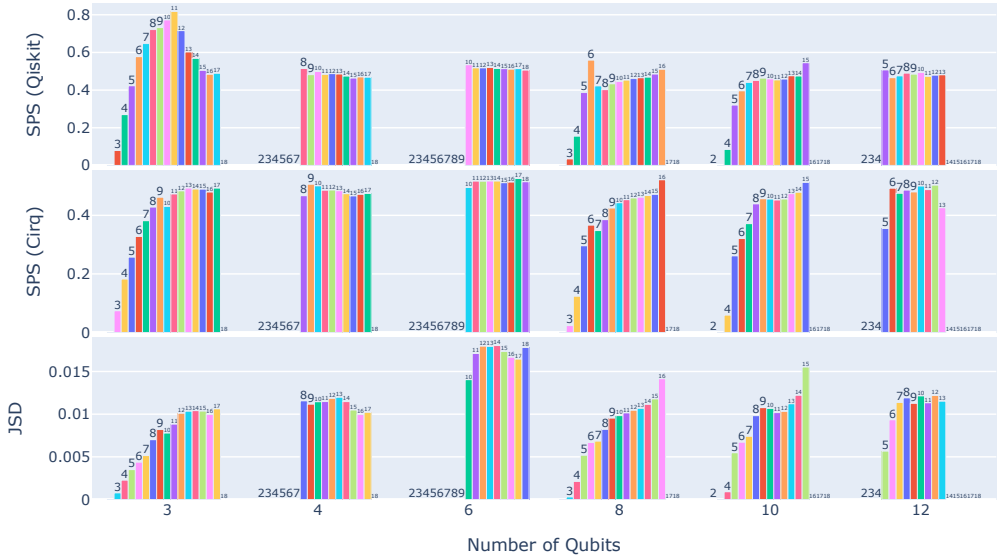


Fig. 8. Simulated Performance Scores (SPS) for Qiskit (top) and Cirq (middle), along with the Mean JS Divergence between Qiskit and Cirq (bottom). The data is grouped by the Number of Qubits, with Circuit Depth annotated above each bar.

Although the regression models for both the χ^2 and JSD metrics capture some trends, the results imply that incorporating additional features could improve predictive accuracy. As discussed in §8.3, the *number of shots* appears to be a key factor influencing the divergence between simulators.

Answer to RQ_4**:** We assessed the accuracy of Qiskit and Cirq simulators using χ^2 tests and JSD. The χ^2 test results revealed no significant trends, except for an anomaly in Qiskit at specific depths for circuits with the lowest qubit count. The SPS indicated similar accuracy levels, with Cirq performing slightly better (SPS = 0.4463 vs. 0.4657 for Qiskit). Polynomial regression on SPS and JSD yielded low predictive power, suggesting weak dependence on qubit count and circuit depth. Overall, both simulators show reasonable alignment, with accuracy influenced by depth and potential additional factors like shot count.

8.5 RQ_5: Scalability and Numerical Stability in Quantum Circuit Simulations

We examine scalability and stability through three lenses: overall runtime cost, memory usage under increasing qubit count, and numerical stability over deep circuits.

Overall Runtime Cost. The total runtime of our methodology is composed of two distinct phases. The first, circuit seed generation with Alloy (as detailed in our answer to RQ1, §8.1), is highly efficient and does not constitute a performance bottleneck. The second and dominant cost is the subsequent quantum circuit simulation, determined by the simulator under test and the circuit's complexity, not our framework. For our evaluations on the CloudLab node, simulating a single ~12-qubit circuit with 10,000 shots typically took from under a second to a few seconds per circuit. Therefore, executing the entire corpus of over 800,000 circuits for our accuracy studies represented a significant computational task that necessitated parallel execution on the high-performance computing resources.

Memory Usage and Qubit Scalability. Our experiments focused on quantum systems with up to 12 qubits, as simulating larger systems rapidly becomes memory-intensive due to the exponential

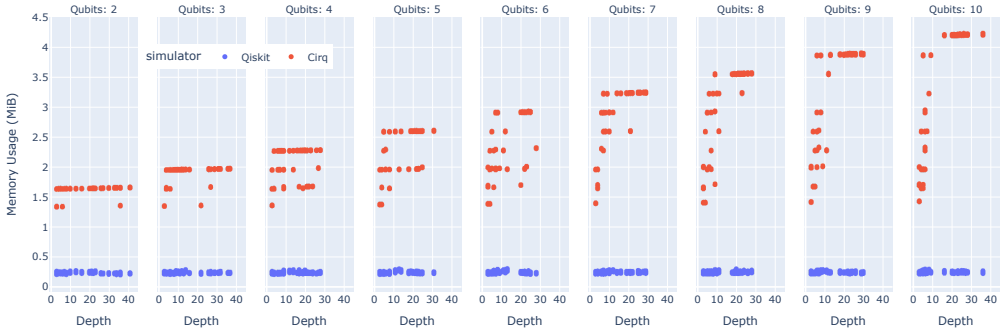


Fig. 9. Memory Usage in Qiskit and Cirq Simulators as a Function of Circuit Depth and Number of Qubits, with shaded regions highlighting different qubit ranges.

growth in the quantum state size and the unitary matrix governing its evolution. Our approach is not inherently limited to this range and can scale to larger systems using GPUs or more powerful hardware.

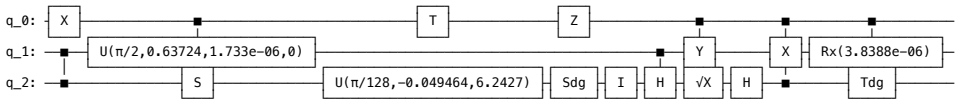
For a system with n qubits, the unitary matrix representing the quantum state's evolution is of size $2^n \times 2^n$, with each entry being a complex number. This leads to an exponential increase in both the statevector (of size 2^n) and the unitary matrix (with 2^{2n} elements). For example, a 15-qubit system requires a unitary matrix of size $2^{15} \times 2^{15}$, containing approximately $2^{30} \approx 10^9$ complex numbers, which requires ~16 GB of memory (assuming each complex number takes 16 bytes). Simulating quantum evolution further escalates memory and computational costs, as applying the unitary matrix to the statevector requires significant resources. This rapid increase in memory and processing requirements limits the scalability of quantum simulators on conventional hardware.

To evaluate how memory usage impacts quantum simulators and hence our ability to validate them, we analysed the efficiency of Qiskit and Cirq in *measurement-based simulations*, tracking only the memory consumption during the measurement step[6]. We selected 345 unique circuits with varying depths and qubit counts, simulating each circuit 1,000 times on both simulators, with 10,000 shots per simulation. We then plotted memory usage against circuit depth and the number of qubits for each simulator, providing a comparative view of their performance and resource utilisation. As shown in Figure 9, Qiskit demonstrated a consistent memory usage of around 0.23 MiB across all circuits, regardless of depth or the number of qubits. In contrast, Cirq's memory usage fluctuated more significantly, ranging from 1.3 MiB to 4.3 MiB, with higher memory demands observed for circuits involving more qubits.

These findings indicate that Qiskit employs a more aggressive approach to memory management. This observation aligns with feedback that we received from Qiskit engineers in response to a prior issue we raised. They explained that Qiskit prioritises the optimisation of "happy paths" – the most commonly used workflows where errors are rare. To catch type-related issues, the recommendation was to rely on static code analysis tools like mypy [65], rather than burdening runtime with checks that could penalise performance for the "happy paths", which are the most common and least error-prone. While this design philosophy ensures excellent memory efficiency and minimal performance overhead in typical scenarios, it comes at a cost. By focusing on "happy paths", Qiskit, as we have highlighted in several reported issues, becomes more exposed to unconventional inputs or edge cases, potentially compromising its robustness in less predictable or atypical situations.

[6]Qiskit: `Aer.get_backend('qasm_simulator').run(transpile(circuit), shots).result();`
Cirq: `cirq.Simulator().run(circuit, repetitions)`.

Numerical Stability and Error Accumulation. We observed that numerical precision becomes a limiting factor when performing iterative computations. Starting with the following circuit:



which consists of three qubits and 16 gates, we repeatedly applied the same set of gates with varying angles, simulating the circuit in Qiskit and Cirq while comparing the statevectors at each step. After 805 concatenations (i.e. constructing a circuit with 16×805 gates), we found that the statevectors produced by Qiskit and Cirq began to diverge beyond our chosen tolerance of 10^{-5}. This tolerance was deliberately set to be not overly strict, as our initial, more conservative choice of 10^{-12} was violated even sooner.

Below are the statevectors at the point where the tolerance was first exceeded.

Basis State	Qiskit	Cirq
$\|000\rangle$	$-2.3258 \times 10^{-6} + 1.8058 \times 10^{-6}\,j$	$0 + 0\,j$
$\|001\rangle$	$1.2789 \times 10^{-6} - 2.2542 \times 10^{-6}\,j$	$0 + 0\,j$
$\|010\rangle$	$1.9366 \times 10^{-6} + 3.5916 \times 10^{-6}\,j$	$0 + 0\,j$
$\|011\rangle$	$-1.6184 \times 10^{-6} - 3.4847 \times 10^{-6}\,j$	$0 + 0\,j$
$\|100\rangle$	$0.6968819 + 0.0320744\,j$	$0.6968802 + 0.0320724\,j$
$\|101\rangle$	$0.5666165 - 0.0979956\,j$	$0.5666154 - 0.0980026\,j$
$\|110\rangle$	$0.1348269 - 0.1214029\,j$	$0.1348258 - 0.1214149\,j$
$\|111\rangle$	$0.2030027 + 0.3294576\,j$	$0.2030029 + 0.3294574\,j$

The divergence observed between the statevectors produced by Qiskit and Cirq arises primarily from differences in how each simulator handles floating-point precision during iterative computations. Both simulators introduce small rounding errors with each quantum gate, but they handle these errors differently. Qiskit retains very small non-zero values that are close to zero, which, though seemingly negligible, still contribute to the overall statevector. Meanwhile, Cirq rounds these small values down to exactly zero, ignoring the residuals. This discrepancy is due to the distinct numerical precision and rounding methods employed by each simulator. While the differences in statevectors may seem minor at first, they become more noticeable in larger circuits, where the accumulation of rounding errors has a more significant impact over many iterations. This difference in behaviour underscores the varying approaches each simulator takes toward numerical accuracy and memory management, making differential benchmarking a challenge as the circuit depth increases in simulation.

Answer to *RQ_5:* Our evaluation revealed key scalability and stability differences between Qiskit and Cirq. Qiskit maintained a stable memory footprint (0.23 MiB) across varying circuit depths and qubit counts, whereas Cirq's memory usage fluctuated significantly (1.3 MiB–4.3 MiB), increasing with the number of qubits. Numerical stability: Qiskit retained near-zero values, while Cirq rounds them to zero, leading to divergence in statevector representations over multiple iterations and potentially affecting long-term simulation accuracy. This suggests that Qiskit optimises memory efficiency for standard workflows but may be less robust to unconventional inputs.

8.6 Experimental Setup

Our evaluation was conducted across a high-performance computing environment provided by CloudLab [19] and a standard laptop (Apple M2, 16GB RAM). Circuit generation (RQ1), post-processing, and local analysis were performed on the laptop.

All large-scale quantum circuit simulations (RQ2–RQ5) were executed on a CloudLab node featuring dual Intel Xeon E5-2660 v3 processors (40 logical cores, 256GB RAM). These experiments utilised a corpus of over 800,000 unique circuits generated by FuzzQ, with qubit counts ranging from 3 to 12 and depths from 2 to 18 gates. The circuit structures included a comprehensive mix of single-qubit (rotations, Hadamard, Pauli) and multi-qubit (CNOT) gates. For analyses based on measurement distributions, measurements were applied to all qubits at the end of each circuit. The quantum platform code coverage analysis (RQ1) was run on a separate CloudLab node with an Intel Xeon Gold 5512U processor.

9 Related Work

Testing with Alloy. Alloy [27] enables rigorous system validation through the automated generation of model instances and counterexamples. By systematically exploring model specifications, Alloy can identify under-specified or ambiguous system properties [63]. Its versatility has been demonstrated across diverse research domains, including empirical testing of GPU compilers and specification validation [32], memory model verification [11, 56, 57, 71], model transformation verification [38], as well as model synthesis, repair and type system analysis [28, 60, 67]. Despite its broad adoption, to our knowledge, this is the first application of the Alloy Analyser to model and analyse quantum simulator behaviour. Our work demonstrates Alloy's potential in this domain, laying the groundwork for future quantum computing research.

Quantum Program-Level Testing. Quantum software testing is challenging, with multiple studies highlighting the critical need for specialised testing and debugging tools for quantum programs [22, 40, 42, 46, 47]. Several testing approaches have been explored, including property-based testing [21], mutation-based techniques [20, 39, 70], verification methods [35, 74], statistical assertion techniques [25, 29, 34], search-based approaches [69] and metamorphic testing [1]. Complementing these efforts, researchers have developed quantum bug benchmarks [10, 37, 76] and identified specific bug patterns, notably in Qiskit programs [75]. The QASMBench suite [33] provides a low-level benchmark focusing on NISQ evaluation and simulation. Despite these advances, quantum program testing remains a nascent field, with existing research primarily limited to small-scale circuits of only a few qubits and dozens of gates.

Quantum Platform-Level Testing. Validating quantum simulators is critical given their potential for unintended behaviours [22, 23]. Two recent approaches illustrate the landscape: MorphQ [30] mutates valid programs via metamorphic relations and compares original vs. mutated behaviours, while Blackwell et al. [7] use a Grammar Mutator to generate random (often invalid) QASM programs, running them on Braket, Quantastica, and Qiskit for comparison. MorphQ offers sophisticated mutation but with low throughput; Blackwell et al. favour speed, but at the cost of many invalid circuits. Our method, though bounded by the same state-vector scalability limit (~15 qubits), guarantees structural validity through formal methods, enabling FuzzQ to reach 140 valid circuits per second and systematically test deeper, more complex structures.

10 Limitations and Future Work

This section clarifies the practical limits of the FuzzQ framework and outlines our future research, which is shaped by two primary factors: our initial focus on pure-state semantics and the inherent scalability challenges of quantum simulation.

Our pure-state approach provides a well-defined validation baseline, but it also means our evaluation on circuits up to 12 qubits is limited by the exponential cost of the post-simulation analysis phase. This bottleneck is imposed by our oracles, which require full statevectors for comparison, and not

by FuzzQ's scalable Alloy-based generator. While other fuzzers manage this cost by imposing short timeouts on arbitrarily large circuits [7, 14, 73], our methodology prioritises generating tractable, structurally-valid circuits that can run to completion for deeper analysis. Our own coverage analysis (Figure 6) supports this strategy, showing that code coverage on simulators saturates with small circuits (~6-qubits). This suggests diminishing returns from simply generating larger pure-state circuits and motivates our primary future direction: extending FuzzQ to support mixed states and noisy simulations – an open challenge in quantum computing [5, 6, 18, 49, 68] – which we detail in the remainder of this section.

10.1 Scope of Pure-State Validation & Path Towards Mixed-State Analysis with FuzzQ

Motivation for Initial Pure-State Focus. The FuzzQ framework, as presented, primarily validates simulators within the pure state formalism. This initial scope was chosen for three key reasons: (1) *Semantic Clarity*: Pure-state evolution via unitary transformations (U) on statevectors ($|\psi\rangle$) provides an unambiguous baseline for validating core functional semantics. (2) *Foundational Correctness*: It allows us to isolate fundamental bugs in gate logic and transpilation before introducing the complexities of noise. (3) *Tractability*: The $O(2^N)$ memory complexity of statevectors, versus $O(2^{2N})$ for density matrices, makes systematic fuzzing over large circuit spaces tractable in the pure-state paradigm.

Implications of Mixed States: Semantics, Completeness, and Oracles. A comprehensive validation strategy must eventually address mixed states, which are indispensable for describing realistic quantum computation arising from noise or partial measurements. This transition entails a fundamental semantic shift from unitary evolution on statevectors to Completely Positive Trace-Preserving (CPTP) maps, or quantum channels ($\mathcal{E}$), acting on density matrices (ρ). FuzzQ's Alloy model (§4) can be extended to represent these noisy operations by introducing new signatures (e.g. `DepolarisingChannel`), enabling the generation of test cases for these more complex scenarios.

Whilst extending the formal model is achievable, integrating full mixed-state validation directly impacts efficiency. This extension is nevertheless vital for FuzzQ's *completeness*, as it enables validation of realistic NISQ-era scenarios and detection of bugs specific to noise models. Consequently, our testing oracles (§6.2) require significant adaptation. For instance, statevector fidelity ($|\langle\psi_i|\psi_j\rangle|^2$) must be replaced by computationally more demanding density matrix fidelity measures (e.g. Uhlmann-Jozsa fidelity, $F(\rho_1, \rho_2) = \left(\mathrm{Tr}\left[\sqrt{\sqrt{\rho_1}\rho_2\sqrt{\rho_1}}\right]\right)^2$). Similarly, the Unitary Parity Benchmark would need to be generalised to compare entire quantum channels. Each of these adapted oracles incurs a greater computational burden, directly affecting validation throughput. For example, applying a Hadamard gate (H) then a depolarising channel ($D(p)$) to $|0\rangle$ yields the mixed state $\rho_{\text{out}} = (1-p)|+\rangle\langle+| + \frac{p}{2}I_2$, not the pure state $|+\rangle = H|0\rangle$. Validating simulator agreement on ρ_{out} requires density matrix metrics and is inherently more demanding than pure-state vector comparison.

Scalability Challenges and FuzzQ's Path Forward for Mixed States. The primary impediment to extending FuzzQ with direct support for full density matrix evolution is scalability. As mentioned, density matrix storage scales as $O(2^{2N})$. Computationally, applying a quantum channel or calculating fidelities often scales as $O(2^{3N})$. These factors make systematic fuzzing that relies on full density matrix simulation intractable for systems beyond a small number of qubits, directly impacting FuzzQ's ability to cover such scenarios at scale.

Recognising these severe constraints, our future work will focus on scalable oracles, such as those based on *classical shadow tomography* [9], to compare statistical properties of mixed states without reconstructing the full density matrix. However, the core FuzzQ framework is already

capable of handling the generation and processing of noisy circuits that produce mixed states. As demonstrated in our accompanying artifact, FuzzQ can systematically inject common noise channels (e.g., depolarising, amplitude damping) with varying probabilities into its Alloy-generated circuit seeds. The resulting circuits, when executed on simulators supporting density matrix evolution, produce verifiable mixed states. For instance, our artifact includes a test case that prepares a Bell state, applies partial tracing to deterministically create a mixed state, and correctly computes its purity ($\mathrm{Tr}(\rho^2) \approx 0.58$), confirming the framework's capability to handle mixed-state semantics. This provides a solid, implemented foundation for our planned future work on developing more advanced, scalable oracles for these noisy scenarios.

10.2 Other Avenues for Future Research and Development

Beyond mixed-state simulations, future work will focus on three key avenues. *First*, we will broaden FuzzQ's evaluation to more quantum simulators beyond Qiskit and Cirq to affirm its platform agnosticism. FuzzQ's modular architecture facilitates this via platform-specific backend adapters, a process detailed and exemplified with PennyLane in our artifact (see §12). *Second*, a significant long-term direction is adapting FuzzQ for direct validation against physical quantum hardware, which introduces distinct challenges in noise characterisation and statistical analysis. *Finally*, we will enhance FuzzQ's core methodology and performance by: (1) developing more sophisticated, quantum-specific coverage metrics; (2) empirically evaluating its bug-finding efficacy by assessing false negative rates against known bug benchmarks (e.g., Bugs4Q [76]); and (3) investigating strategies for parallelising quantum circuit simulations to improve performance.

11 Conclusion

This work introduced FuzzQ, an implementation-agnostic methodology for detecting and analysing discrepancies across industrial quantum simulation frameworks. By combining lightweight formal methods (Alloy) for structured test case generation with differential benchmarking and invariant checking, we have developed a robust validation framework that transcends simulator-specific architectural constraints, enabling comprehensive inter-simulator comparison.

The core contribution of our research is the *unitary parity benchmark*, a novel and rigorous technique for systematically identifying and diagnosing inconsistencies such as miscompilations and numerical inaccuracies among quantum simulators. Our extensive evaluation on Qiskit and Cirq, involving over 800,000 quantum circuits, demonstrated FuzzQ's efficiency in circuit generation, its effectiveness in uncovering previously unknown bugs, and provided insights into simulator performance characteristics.

Our research represents a unique interdisciplinary effort, merging theoretical insights from physics with practical software engineering methodologies to advance the modelling and analysis of quantum simulators. By offering a rigorous and automatable methodology for scrutinising these critical tools, this work sets the stage for more reliable and precise computational approaches in quantum computing research. The FuzzQ framework not only uncovers potential implementation flaws but also establishes a solid foundation for continued investigations into quantum system validation and the development of more trustworthy quantum software ecosystems.

12 Data Availability Statement

All research artifacts associated with this paper, including FuzzQ's source code, experimental data, and scripts for reproducability, are publicly available on Zenodo DOI 10.5281/zenodo.16918102 [31].

Author Contributions. After the first author, all subsequent authors contributed equally to the conceptualisation, experimental design, and manuscript revision, and are listed in alphabetical order.

Acknowledgements

We thank CloudLab [19] for providing the platform and infrastructure support that was instrumental for the large-scale experiments presented in our evaluation. This work was supported by EPSRC projects VSL-Q (EP/Y005244/1), RoaRQ (Investigation 009), and ModeMCQ (EP/W032635/1), the QAssure project from Innovate UK, and King's Quantum grants from King's College London.

References

[1] Rui Abreu, João Paulo Fernandes, Luis Llana, and Guilherme Tavares. 2022. Metamorphic Testing of Oracle Quantum Programs. In *2022 IEEE/ACM 3rd International Workshop on Quantum Software Engineering (Q-SE)* (2022-05). IEEE, Pittsburgh, PA, USA, 16–23. doi:10.1145/3528230.3529189

[2] Giovanni Acampora, Ferdinando Di Martino, Alfredo Massa, Roberto Schiattarella, and Autilia Vitiello. 2023. D-NISQ: A reference model for Distributed Noisy Intermediate-Scale Quantum computers. *Information Fusion* 89 (2023), 16–28. doi:10.1016/j.inffus.2022.08.003

[3] Amazon Web Services (AWS). Accessed: 2025-01-17. Amazon Braket. https://aws.amazon.com/braket/.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[5] Johannes Bausch, Andrew W Senior, Francisco JH Heras, Thomas Edlich, Alex Davies, Michael Newman, Cody Jones, Kevin Satzinger, Murphy Yuezhen Niu, Sam Blackwell, et al. 2024. Learning high-accuracy error decoding for quantum processors. *Nature* 635, 8040 (2024), 834–840.

[6] Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S. Kottmann, Tim Menke, Wai-Keong Mok, Sukin Sim, Leong-Chuan Kwek, and Alán Aspuru-Guzik. 2022. Noisy intermediate-scale quantum (NISQ) algorithms. *Rev. Mod. Phys.* 94 (Feb 2022), 015004. Issue 1. doi:10.1103/RevModPhys.94.015004

[7] Daniel Blackwell, Justyna Petke, Yazhuo Cao, and Avner Bensoussan. 2024. Fuzzing-Based Differential Testing for Quantum Simulators. In *Search-Based Software Engineering* (Cham, 2024). Springer Nature Switzerland, Cham, 63–69. doi:10.1007/978-3-031-64573-0_6

[8] Nicolas Brunner, Daniel Cavalcanti, Stefano Pironio, Valerio Scarani, and Stephanie Wehner. 2014. Bell Nonlocality. *Rev. Mod. Phys.* 86, 2 (April 2014), 419–478. doi:10.1103/RevModPhys.86.419

[9] Zhenyu Cai, Ryan Babbush, Simon C. Benjamin, Suguru Endo, William J. Huggins, Ying Li, Jarrod R. McClean, and Thomas E. O'Brien. 2023. Quantum error mitigation. *Rev. Mod. Phys.* 95 (Dec 2023), 045005. Issue 4. doi:10.1103/RevModPhys.95.045005

[10] José Campos and André Souto. 2021. QBugs: A Collection of Reproducible Bugs in Quantum Algorithms and a Supporting Infrastructure to Enable Controlled Quantum Software Testing and Debugging Experiments. http://arxiv.org/abs/2103.16968 arXiv:2103.16968 [cs].

[11] Nathan Chong, Tyler Sorensen, and John Wickerson. 2018. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, USA) *(PLDI)*. Association for Computing Machinery, Philadelphia, PA, USA, 211–225.

[12] Cirq GitHub Issue Tracker. 2024. GitHub Issue #6810. https://github.com/quantumlib/Cirq/issues/6810. GitHub accessed August 2025.

[13] Cirq GitHub Issue Tracker. 2024. GitHub Issue #6811. https://github.com/quantumlib/Cirq/issues/6811. GitHub accessed August 2025.

[14] Fuzz4All Contributors. 2024. Qiskit Target Runner for Fuzz4All. https://github.com/fuzz4all/fuzz4all/blob/main/Fuzz4All/target/QISKIT/QISKIT.py. Accessed: 2025-07-21.

[15] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3 (Sept. 2022), 1–50. doi:10.1145/3505636

[16] D-Wave Systems. Accessed Jan. 2025. D-Wave Documentation. https://docs.ocean.dwavesys.com/en/stable/.

[17] Cirq Developers. 2024. Cirq. doi:10.5281/zenodo.11398048

[18] Giovanni Di Bartolomeo, Michele Vischi, Francesco Cesa, Roman Wixinger, Michele Grossi, Sandro Donadi, and Angelo Bassi. 2023. Noisy gates for simulating quantum computers. *Phys. Rev. Res.* 5 (Dec 2023), 043210. Issue 4. doi:10.1103/PhysRevResearch.5.043210

[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The design and operation of cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[20] Daniel Fortunato, José Campos, and Rui Abreu. 2022. Mutation testing of quantum programs written in QISKit. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (New York, NY, USA, 2022-10-19) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 358–359. doi:10.1145/3510454.3528649

[21] Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and Mario Piattini. 2023. Quantum software testing: State of the art. *Journal of Software: Evolution and Process* 35, 4 (2023), e2419. doi:10.1002/smr.2419 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2419.

[22] Sukhpal Singh Gill, Adarsh Kumar, Harvinder Singh, Manmeet Singh, Kamalpreet Kaur, Muhammad Usman, and Rajkumar Buyya. 2022. Quantum computing: A taxonomy, systematic review and future directions. *Software: Practice and Experience* 52, 1 (2022), 66–114. doi:10.1002/spe.3039

[23] Philipp Hauke, Fernando M. Cucchietti, Luca Tagliacozzo, Ivan Deutsch, and Maciej Lewenstein. 2012. Can one trust quantum simulators? *Reports on Progress in Physics* 75, 8 (2012), 082401. doi:10.1088/0034-4885/75/8/082401 Publisher: IOP Publishing.

[24] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for Quantum circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (Jan. 2021), 29 pages. doi:10.1145/3434318

[25] 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* (2019-06-22). Association for Computing Machinery, New York, NY, USA, 541–553. arXiv:1905.09721 [quant-ph] doi:10.1145/3307650.3322213

[26] IBM Quantum Computing. Accessed: 2025-01-17. Qiskit. https://www.ibm.com/quantum/qiskit.

[27] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76. doi:10.1145/3338843

[28] Ana Jovanovic and Allison Sullivan. 2022. Towards automated input generation for sketching alloy models. In *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering (FormaliSE)*. IEEE, Pittsburgh, PA, USA, 58–68.

[29] Chan Gu Kang, Joonghoon Lee, and Hakjoo Oh. 2024. Statistical Testing of Quantum Programs via Fixed-Point Amplitude Amplification. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 276 (Oct. 2024), 25 pages. doi:10.1145/3689716

[30] Linsey J. Kitt and Myra B. Cohen. 2024. MorphQ++: A Reproducibility Study of Metamorphic Testing on Quantum Compilers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops* (Sacramento, CA, USA) *(ASEW '24)*. Association for Computing Machinery, New York, NY, USA, 8–14. doi:10.1145/3691621.3694959

[31] Vasileios Klimis, Avner Bensoussan, Elena Chachkarova, Karine Even Mendoza, Sophie Fortz, and Connor Lenihan. 2025. *FuzzQ Artifact: Shaking Up Quantum Simulators with Fuzzing and Rigour*. doi:10.5281/zenodo.16918102 Dataset.

[32] Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. 2023. Taking Back Control in an Intermediate Representation for GPU Computing. *Proc. ACM Program. Lang.* 7, POPL (2023), 30 pages.

[33] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. QASMBench: A Low-Level Quantum Benchmark Suite for NISQ Evaluation and Simulation. *ACM Transactions on Quantum Computing* 4, 2 (2023), 10:1–10:26. doi:10.1145/3550488

[34] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Proq: Projection-based Runtime Assertions for Debugging on a Quantum Computer. http://arxiv.org/abs/1911.12855 arXiv:1911.12855 [quant-ph].

[35] Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2022. Verified compilation of Quantum oracles. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 146 (Oct. 2022), 27 pages. doi:10.1145/3563309

[36] Mohamed Messaoud Louamri, Nacer eddine Belaloui, Abdellah Tounsi, and Mohamed Taha Rouabah. 2024. Comparative Study of Quantum Transpilers: Evaluating the Performance of Qiskit-Braket-Provider, qBraid-SDK, and Pytket Extensions. In *2024 1st International Conference on Innovative and Intelligent Information Technologies (IC3IT)*. IEEE, New York, NY, USA, 1–6. doi:10.1109/IC3IT63743.2024.10869429

[37] Junjie Luo, Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. 2022. A Comprehensive Study of Bug Fixes in Quantum Programs. https://www.computer.org/csdl/proceedings-article/saner/2022/378600b239/1FbT6n3hGaA

[38] Nuno Macedo and Alcino Cunha. 2013. Implementing QVT-R bidirectional model transformations using Alloy. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer,

Rome, Italy, 1–15.
[39] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. 2021. Muskit: A Mutation Analysis Tool for Quantum Software Testing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021-11). IEEE, Los Alamitos, CA, USA, 1266–1270. doi:10.1109/ASE51524.2021.9678563 ISSN: 2643-1572.
[40] Sara Ayman Metwalli and Rodney Van Meter. 2022. A Tool For Debugging Quantum Circuits. https://www.computer.org/csdl/proceedings-article/qce/2022/911300a624/1IvLZRlty80
[41] Microsoft. Accessed: 2025-01-17. Microsoft Azure Quantum. https://azure.microsoft.com/en-us/products/quantum/.
[42] Andriy Miranskyy and Lei Zhang. 2019. On testing quantum programs. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '19)*. IEEE Press, Montreal, Quebec, Canada, 57–60. doi:10.1109/ICSE-NIER.2019.00023
[43] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information (10th Anniversary edition)*. Cambridge University Press, Cambridge.
[44] NumPy Developers. 2018. Einsum symbol limitations. https://github.com/numpy/numpy/issues/11938. GitHub issue #11938, accessed July 2025.
[45] NumPy Developers. 2022. How is NumPy's einsum so much slower than other libraries? https://github.com/numpy/numpy/issues/22604. GitHub issue #22604, accessed July 2025.
[46] Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum computing platforms: an empirical study. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 1–27. doi:10.1145/3527330
[47] Matteo Paltenghi and Michael Pradel. 2023. MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform . In *IEEE/ACM 45th International Conference on Software Engineering (ICSE) (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 2413–2424. doi:10.1109/ICSE48619.2023.00202
[48] Matteo Paltenghi and Michael Pradel. 2024. A Survey on Testing and Analysis of Quantum Software. *ArXiv* abs/2410.00650 (2024), 1–10. https://api.semanticscholar.org/CorpusID:273023238
[49] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018), 79. doi:10.1145/237814.237866
[50] QIR Alliance. Accessed: 2025-01-03. QIR Alliance Projects. https://www.qir-alliance.org/projects/
[51] Qiskit GitHub Issue Tracker. 2024. GitHub Issue #13536. https://github.com/Qiskit/qiskit/issues/13536. GitHub accessed August 2025.
[52] Qiskit GitHub Issue Tracker. 2024. GitHub Issue #13583. https://github.com/Qiskit/qiskit/issues/13583. GitHub accessed August 2025.
[53] Qiskit GitHub Issue Tracker. 2024. GitHub Issue #2274. https://github.com/Qiskit/qiskit-aer/issues/2274. GitHub accessed August 2025.
[54] Qiskit GitHub Issue Tracker. 2024. GitHub Issue #2276. https://github.com/Qiskit/qiskit-aer/issues/2276. GitHub accessed August 2025.
[55] Qiskit GitHub Issue Tracker. 2024. GitHub Issue #4159 and #7120. https://github.com/Qiskit/qiskit/issues/4159, https://github.com/Qiskit/qiskit/issues/7120. Known bugs, GitHub accessed August 2025.
[56] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.* 4, POPL (2020), 1–31.
[57] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 1–27.
[58] Neilson C. L. Ramalho, Higor Amario de Souza, and Marcos Lordello Chaim. 2025. Testing and Debugging Quantum Programs: The Road to 2030. *ACM Trans. Softw. Eng. Methodol.* 34 (Jan. 2025), 1–45. doi:10.1145/3715106
[59] Rigetti Computing. Accessed: 2025-01-17. Rigetti Forest SDK. https://www.rigetti.com/forest.
[60] Michael Roberson, Melanie Harries, Paul T. Darga, and Chandrasekhar Boyapati. 2008. Efficient software model checking of soundness of type systems. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. Association for Computing Machinery, Nashville, Tennessee, USA, 1–10.
[61] Hinrich Schütze and Christopher D. Manning. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA. 304 pages.
[62] George W. Snedecor and William G. Cochran. 1989. *Statistical Methods* (8th ed.). Iowa State University Press, Ames, Iowa.
[63] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2017. Automated Test Generation and Mutation Testing for Alloy. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Tokyo, Japan, 264–275.
[64] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2022. Giallar: push-button verification for the qiskit Quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*.

Association for Computing Machinery, New York, NY, USA, 641–656. doi:10.1145/3519939.3523431
[65] Mypy Team. Accessed: 2025-03-20. Mypy: A static type checker for Python. https://github.com/python/mypy.
[66] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021-11) *(ASE '21)*. IEEE Press, Melbourne, Australia, 692–704. doi:10.1109/ASE51524.2021.9678792 ISSN: 2643-1572.
[67] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. ASketch: a sketching framework for Alloy. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 916–919. doi:10.1145/3236024.3264594
[68] Meng Wang, Swamit Tannu, and Prashant J Nair. 2025. Accelerating Simulation of Quantum Circuits under Noise via Computational Reuse. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1539–1553. doi:10.1145/3695053.3730992
[69] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. QuSBT: search-based testing of quantum programs. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 173–177. doi:10.1145/3510454.3516839
[70] Xinyi Wang, Tongxuan Yu, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. Mutation-based test generation for quantum programs with multi-objective search. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 1345–1353. doi:10.1145/3512290.3528869
[71] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery, Paris, France, 1–15.
[72] Xanadu Quantum Technologies. Accessed: 2025-01-17. PennyLane. https://pennylane.ai/.
[73] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 126, 13 pages. doi:10.1145/3597503.3639121
[74] Peng Yan, Hanru Jiang, and Nengkun Yu. 2022. On incorrectness logic for Quantum programs. *Proc. ACM Program. Lang.* 6, OOPSLA1 (April 2022), 72:1–72:28. doi:10.1145/3527316
[75] Pengzhan Zhao, Jianjun Zhao, and Lei Ma. 2021. Identifying Bug Patterns in Quantum Programs . 16-21 pages. doi:10.1109/Q-SE52541.2021.00011
[76] Pengzhan Zhao, Jianjun Zhao, Zhongtao Miao, and Shuhan Lan. 2021. Bugs4Q: A Benchmark of Real Bugs for Quantum Programs. http://arxiv.org/abs/2108.09744 arXiv:2108.09744 [cs].

Received 2025-03-22; accepted 2025-08-12