IterTestQ: Assembly-Level, Cross-Platform Testing of Quantum Computing Platforms

MATTEO PALTENGHI, University of Stuttgart, Germany
MICHAEL PRADEL, CISPA Helmholtz Center for Information Security, Germany

Quantum computing platforms are susceptible to quantum-specific bugs, such as incorrectly ordering qubits or incorrectly implementing quantum abstractions. These bugs are difficult to detect and require specialized expertise. The field faces challenges due to a fragmented landscape of platforms and rapid development cycles that often prioritize new features over thorough testing, severely hindering the reliability of quantum software. To address these challenges, we present IterTestQ, a novel cross-platform testing approach for quantum computing platforms. The key technical contribution is our novel ITE process, which generates equivalent quantum programs by iteratively (I)mporting them into platform-specific representations, (T)ransforming the program via optimizations and gate conversions, and (E)xporting the program again. To transfer programs across platforms and test cross-platform consistency, IterTestQ leverages QASM, an assembly-level representation supported by most platforms. The approach uses a crash oracle to detect failures during cross-platform transformations and an equivalence oracle to validate the semantic consistency of the generated assembly programs, which are expected to be equivalent by construction. We evaluate IterTestQ on widely-used quantum computing platforms, including Qiskit, PennyLane, Pytket, BQSKit, and Cirq, revealing 23 bugs, 17 of which are already confirmed or fixed. Our results also demonstrate that IterTestQ complements existing quantum fuzzers (covering tens of thousands of otherwise uncovered lines), is efficient (with 0.00089 seconds per generated program), and that the ITE process is crucial for its effectiveness.

CCS Concepts: • **Software and its engineering → Compilers**; **Software verification and validation**; • **Hardware → Quantum technologies**.

Additional Key Words and Phrases: quantum computing, software testing, cross-platform testing, program equivalence, quantum circuits

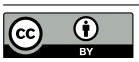
ACM Reference Format:
Matteo Paltenghi and Michael Pradel. 2026. IterTestQ: Assembly-Level, Cross-Platform Testing of Quantum Computing Platforms. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA120 (October 2026), 23 pages. https://doi.org/10.1145/3832211

arXiv:2503.17322v2 [cs.SE] 2 Nov 2026

1 Introduction

Quantum computing promises to solve computationally challenging problems beyond the reach of traditional computing. An essential prerequisite to realize this potential is having reliable *quantum computing platforms*, i.e., software systems that let developers define, manipulate, and execute quantum programs. Growing interest from industry and research highlights the importance of these platforms for advancing quantum computing. However, ensuring their reliability is challenging due to the complexity of quantum mechanics and the quantum software stack. These challenges lead to quantum-specific bugs that require specialized expertise to detect and fix [42]. Platform bugs can cause incorrect program behavior, impeding the development of quantum applications. The

Authors' Contact Information: Matteo Paltenghi, University of Stuttgart, Stuttgart, Germany, mattepalte@live.it; Michael Pradel, CISPA Helmholtz Center for Information Security, Stuttgart, Germany, michael@binaervarianz.de.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/10-ARTISSTA120
https://doi.org/10.1145/3832211

```python
from qiskit import QuantumCircuit,
    QuantumRegister
qr = QuantumRegister(4)
qc = QuantumCircuit(qr)
qc.mcx([qr[0], qr[1], qr[2]], qr[3])
```

(a) Qiskit circuit with a three-control MCX.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q2[4];
c4x q2[0],q2[1],q2[2],q2[3];
```

(b) Incorrect QASM from Pytket (c4x instead of c3x).

Fig. 1. Pytket generates an incorrect c4x gate for a three-control MCX operation (Bug #13).

intricate nature of these bugs can undermine trust in quantum computing platforms, motivating dedicated testing techniques to ensure reliability.

The quantum software landscape is characterized by a diverse ecosystem of platforms, including Qiskit [21], Cirq [1], PennyLane [4], Pytket [52], and BQSKit [63], each providing a unique framework for quantum programming. To ensure interoperability in this diverse ecosystem, many quantum computing platforms share a common representation for quantum programs: Quantum Assembly Language (QASM) [14], which serves as a standard assembly language for representing quantum circuits. Most quantum computing platforms support importing and exporting circuits in QASM format, making it a key enabler of cross-platform compatibility.

While the fragmentation of the quantum software ecosystem drives innovation, it introduces challenges in ensuring interoperability and causes gaps in current testing practices. Existing approaches for testing quantum computing platforms [43, 44] are tailored to specific platforms, lacking the ability to validate the consistency of quantum circuit execution across different platforms. This cross-platform consistency matters when developers exchange programs across specialized quantum computing platforms that target different architectures. Therefore, a need for comprehensive cross-platform testing is emerging to ensure reliable interoperability across this diverse landscape.

To illustrate how bugs in a platform's assembly code generation and import/export functions can introduce subtle compatibility issues, consider Figure 1. When converting and exporting a quantum program with a three-control multiple-controlled NOT operation (MCX), as shown in Listing 1a, Pytket generates incorrect QASM assembly code. The resulting QASM uses a `c4x` gate (for four controls) instead of the required `c3x` gate (for three controls), as shown in Listing 1b. This bug in Pytket's gate definition causes crashes when other platforms import the generated assembly code. Unfortunately, the issue easily goes undetected when testing Pytket in isolation, since it can read back its own incorrect code.

The above bug illustrates a more general challenge: While interoperability across quantum computing platforms is essential, there currently is no systematic approach to test this interoperability layer. On the positive side, having multiple platforms that share a common assembly representation also offers opportunities: Checking for cross-platform consistency can serve as a test oracle, which can help uncover subtle bugs in the assembly-level import/export functionality of quantum computing platforms. Moreover, a cross-platform testing approach can test multiple platforms simultaneously, increasing the breadth of testing way beyond the current single-platform testing approaches [43, 44].

This work introduces IterTestQ, a novel approach for cross-platform testing of quantum computing platforms. IterTestQ harnesses the commonalities in quantum platforms through the use of a common assembly representation and employs an equivalence oracle to identify discrepancies in compiled circuit outputs. IterTestQ starts by generating programs in the common assembly language. For each generated program, IterTestQ then produces an equivalence class of semantically equivalent programs by iteratively importing, transforming, and exporting a program across

different platforms under test. This process, called the *ITE process*, is repeated multiple times to create a pool of semantically equivalent programs that can be used for cross-platform testing. To amplify the number of equivalent programs, IterTestQ employs two transformation strategies: (a) applying platform-specific optimization passes and (b) converting the program gate set to a different but equivalent gate set. Finally, IterTestQ uses two oracles to detect bugs: a *crash oracle* that checks for crashes during all components of IterTestQ and an *equivalence oracle* that compares the programs in the same equivalence class for semantic equivalence. The approach is designed to test several critical components of quantum computing platforms, including the import and export of assembly programs, the internal representation of circuits, and the application of circuit transformations. In contrast, testing the execution of quantum programs on quantum hardware is out of scope for IterTestQ.

Our evaluation shows that IterTestQ is highly effective at finding bugs in quantum computing platforms. Across five platforms, IterTestQ discovered 23 bugs, with 17 already confirmed or fixed by developers. These bugs span a range of issues, from incorrect gate implementations to compiler optimizations that fail to preserve program semantics. Moreover, by focusing on the interoperability layer, IterTestQ reveals unique issues that are typically out of scope for existing quantum fuzzers focused on a single platform, demonstrating strong complementarity with current testing approaches.

Our work advances the state-of-the-art in testing quantum computing platforms. While some prior work, e.g., QDiff [56], MorphQ [43], Fuzz4All [60], and QuteFuzz [19], focuses on testing a single platform across different configurations and optimization levels, we introduce a more general approach that tests interoperability across multiple quantum computing platforms by leveraging the common assembly-level representation. Other existing work, e.g., FuzzQ [26] and work by Blackwell et al. [5], also perform cross-platform differential testing, but compare simulator behavior instead of testing cross-platform interoperability. A key difference to all the above approaches is that IterTestQ is the only one that uses cross-platform interoperability as a means to generate new quantum programs.

The contributions of this work include:

- We introduce IterTestQ, a novel approach for cross-platform testing of quantum computing platforms.
- We present the novel ITE process, which iteratively imports, transforms, and exports programs across different platforms to create a pool of semantically equivalent programs.
- We conduct an extensive evaluation of IterTestQ on five widely-used quantum computing platforms, demonstrating its effectiveness in detecting 23 bugs, with (so far) 17 confirmed and/or fixed by developers, and showing its complementarity to existing quantum fuzzers.
- We provide code and data (Section 8) to reproduce our results, enhancing transparency and inspiring future research.

2 Background

2.1 Quantum Software and QASM

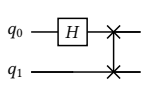
Quantum computing platforms enable developers to define, manipulate, and execute quantum programs. These programs take the form of quantum circuits, where register definitions first specify the available qubits, followed by a sequence of quantum operations (gates) applied to specific qubits or groups of qubits. Through these structured programs, developers implement quantum algorithms, with the platform managing the complex task of preparing the programs for execution on quantum hardware.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
h q[0];
swap q[0],q[1];
```

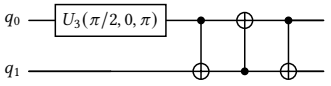
(a) Original QASM.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
u3(pi/2,0,pi) q[0];
cx q[0],q[1];
cx q[1],q[0];
cx q[0],q[1];
```

(b) Transformed QASM.



(c) Original circuit.



(d) Transformed circuit.

Fig. 2. Original quantum circuit and its transformed version using the U3 and CX gate set.

QASM [14] serves as a standard assembly-level representation to specify quantum programs. Like traditional assembly languages, QASM follows a simple structure. Each program starts with a prologue that declares the version and includes standard libraries. The program body consists of register definitions, analogous to variable declarations in classical programs, followed by quantum operations. Each operation specifies a gate name, optional parameters, and the qubits it acts on. Figure 2a illustrates this structure with a simple quantum program. The program applies a Hadamard gate (`h`) to qubit 0 – a quantum operation that puts the qubit in an equal superposition of the states 0 and 1 – followed by a swap operation (`swap`) between qubits 0 and 1 – a quantum operation that swaps the states of the two qubits. Figure 2c shows the corresponding quantum circuit diagram.

2.2 Program Transformations

Program transformations are essential techniques in quantum computing for generating semantically equivalent quantum programs. Two fundamental types of transformations are optimization passes and gate set conversions. Optimization passes reduce circuit complexity by minimizing the number of quantum gates while preserving behavior. Gate set conversions transform circuits to use different sets of quantum gates, which is particularly useful when targeting specific hardware platforms. As illustrated in Figures 2b and 2d, a quantum circuit can be transformed into an equivalent version using only `u3` and `cx` gates, a universal gate set commonly used in quantum computing platforms like Qiskit [21].

3 The IterTestQ Approach

We start by defining the problem this paper addresses, and then present the components of IterTestQ.

3.1 Problem Statement

We address the challenge of cross-platform testing for quantum computing platforms. Given a set of n quantum computing platforms $P = \{p_1, \ldots, p_n\}$, we assume that each platform p provides an import function i and an export function e that support a common assembly language shared by all platforms, e.g., QASM [14]. Let A be the set of assembly programs and R be the set of platform-specific representations. The importer function $i_p : A \rightarrow R$ converts an assembly program into the platform's representation. The exporter function $e_p : R \rightarrow A$ converts the platform's representation back into assembly. Additionally, each platform can offer transformations $t \in T_p$ that produce semantically equivalent programs, i.e., $t : R \rightarrow R$ and for any $r, r' \in R$, if $r' = t(r)$ then $r \sim r'$,

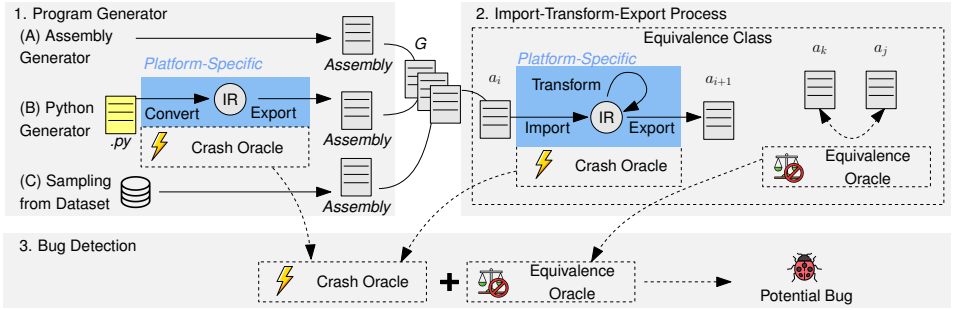


Fig. 3. Overview of IterTestQ. The platform-specific components (blue boxes) are implemented by randomly sampling from the platforms under test.

where $\sim$ denotes semantic equivalence. Note that IterTestQ also supports platforms that do not offer any transformations, i.e., $T_p = \emptyset$.

Our testing goal is to automatically detect two types of bugs: (a) platform crashes that occur while a platform manipulates programs, e.g., while running i_p, e_p, or any $t \in T_p$, and (b) semantic inequivalences $a \nsim a'$ between assembly programs that, by construction, should be equivalent.

3.2 Overview

To address this goal, IterTestQ uses a three-component approach: program generation, our novel ITE process, and bug detection (Figure 3). First, the program generation component (Sec. 3.3) produces quantum assembly programs either by directly generating assembly code, by transforming platform-specific programs via platform converters, or by sampling from a pre-existing corpus of programs. Second, the ITE process (Sec. 3.4) iteratively imports, transforms, and exports programs across platforms. This process creates new programs that form an equivalence class, because importing, transforming, and exporting are supposed to preserve semantic equivalence. An equivalence class E contains programs such that $a_i \sim a_j$ for all $a_i, a_j \in E$. Each conversion and transformation is platform-specific, relying on the internal code of the platform under test. Finally, the bug detection component (Sec. 3.5) uses two oracles: a crash oracle that monitors program generation and the ITE process, and an equivalence oracle that checks semantic equivalence within each equivalence class. If either oracle detects a crash or an inequivalence, IterTestQ reports a bug.

3.3 Program Generation

To create quantum programs, IterTestQ supports three generators that produce assembly code in QASM. The first two generators build upon a template-based approach, which has proven effective in uncovering bugs when testing a single quantum computing platform [43], whereas the third generator samples program from an existing corpus of quantum programs.

The first generator, referred to as the *assembly-based generator*, creates QASM code directly using a grammar-based approach, as shown in Figure 5. The grammar defines the basic structure of quantum programs, including register declarations and quantum operations, ensuring that the generated programs are syntactically valid. One example of possible QASM code generated by this generator is shown in Figure 2a.

The second generator, referred to as the *Python-based generator*, produces high-level quantum programs using Qiskit's Python API [21]. The grammar closely resembles the QASM grammar since the Qiskit API mirrors QASM syntax. An example of a Qiskit program generated by the Python-based generator is shown in Listing 1a. Many quantum platforms provide converters from Qiskit to their

```python
from pytket.extensions.qiskit import qiskit_to_tk
pytket_circ = qiskit_to_tk(qc)
```

(a) Conversion from Qiskit to Pytket representation.

```python
from pytket.qasm import circuit_to_qasm_str
qasm_str = circuit_to_qasm_str(pytket_circ)
```

(b) Exporter to QASM in Pytket.

Fig. 4. Pytket's platform-specific converter and exporter APIs.

internal representations. This generator leverages these platform-specific converters to transform the generated Qiskit code into other platforms' representations. Figure 4a shows an example of conversion from a Qiskit circuit to Pytket's internal representation using platform-specific APIs. The internal representation is then exported to QASM using another platform-specific API, e.g., as shown in Figure 4b. IterTestQ supports multiple platforms, each with a different converter and specific process to convert the Qiskit representation to the platform's internal representation and back to QASM. Both the converter code and the subsequent export to QASM are implemented by the platform maintainers, ensuring that we test only the platform's code during this translation process. We select Qiskit to generate the initial Python program because it was developed by the same company that proposed the QASM specification, IBM. This alignment ensures that Qiskit offers the most robust support for QASM features during translation. Our evaluation confirms this, as Qiskit's implementation of the QASM specification stands out as being particularly robust.

The third generator, referred to as *sampling from dataset*, retrieves programs from a pre-existing corpus of diverse quantum programs [48]. This corpus contains implementations of a variety of quantum algorithms, and hence, complements the other two generators by offering realistic programs.

IterTestQ can be configured to use any subset of these generators, allowing for flexibility in program generation. By default, the approach uses all three generators equally, alternating between them to produce batches of programs. For the third generator, IterTestQ samples each program at most once to maximize diversity, falling back to the other generators once the pre-existing corpus is exhausted. Our evaluation studies the impact of the generators in detail (Section 4.5).

3.4 Import–Transform–Export Loop

The ITE process is the central component of IterTestQ and our key technical contribution. Algorithm 1 shows the pseudocode for this iterative process. In each iteration, the algorithm imports the generated assembly code into a specific platform, transforms it, and then exports it back to assembly. The algorithm picks an initial assembly program a from the set of generated programs G (line 8) and initializes an equivalence class E with a (line 9). An example of an input assembly program is shown in Figure 2a. Then, the algorithm iterates m times (line 10). Within this loop, it samples a platform p from the set of platforms P (line 11). The algorithm imports the current assembly program a into the platform p (line 12) converting it to the platform-specific representation r. The ITE process then samples a transformation t (line 13) and applies it to r to obtain r' (line 14), which is a semantically equivalent representation of the program a. Next, the algorithm exports r' back to assembly, yielding the transformed assembly program a' (line 15). An example of an output assembly program, after applying a transformation that changes its gate set, is shown in Figure 2b. The transformed assembly program a' is added to the equivalence class E (line 16), and the current

Program	::=	Header RegisterDecl StatementList
Header	::=	`"OPENQASM 2.0;"`
RegisterDecl	::=	`"qreg q["` QubitCount `"];"`
		`"creg c["` QubitCount `"];"`
StatementList	::=	Statement*
Statement	::=	GateStatement `";"`
GateStatement	::=	SingleQubitGate | TwoQubitGate | ...
SingleQubitGate	::=	`"h q["` QIx `"]"` | `"rx("` Param `") q["` QIx `"]"` | ...
TwoQubitGate	::=	`"cx q["` QIx `"],q["` QIx `"]"` | `"crz("` Param `") q["` QIx `"],q["` QIx `"]"` | ...

Constraints:

- QubitCount, QIx $\in \mathbb{N} \wedge$ Param $\in \mathbb{R}$
- QIx $<$ QubitCount $\wedge$ QubitCount, QIx ≥ 0

Fig. 5. Grammar of our quantum assembly generator, defining the structure of quantum programs with register declarations and quantum operations. The `h` and `rx` gates act on single qubits, while `cx` and `crz` gates act on pairs of qubits. More gates are supported but omitted for brevity.

Algorithm 1 Import-Transform-Export (ITE) process

Require:

1: P: Quantum platforms $\{p_1, \ldots, p_n\}$
2: G: Set of generated programs
3: T_p: Transformations for platform $p \in P$
4: m: Number of ITE iterations

Ensure:

5: EQ: Set of equivalence classes of assembly programs
6: $EQ \leftarrow \emptyset$ ▷ Initialize the set of equivalence classes
7: **while** testing budget remaining **do**
8: $a \leftarrow \text{Sample}(G)$ ▷ Pick initial assembly program
9: $E \leftarrow \{a\}$ ▷ Start a new equivalence class
10: **for** $i \leftarrow 1$ **to** m **do** ▷ Iterate through ITE iterations
11: $p \leftarrow \text{Sample}(P)$ ▷ Select a platform
12: $r \leftarrow i_p(a)$ ▷ Import assembly into platform p
13: $t \leftarrow \text{Sample}(T_p)$ ▷ Select a transform for platform p
14: $r' \leftarrow t(r)$ ▷ Apply transformation t
15: $a' \leftarrow e_p(r')$ ▷ Export to assembly
16: $E.\text{append}(a')$ ▷ Add to equivalence class
17: $a \leftarrow a'$ ▷ Update current program
18: **end for**
19: $EQ.\text{append}(E)$ ▷ Add equivalence class to list
20: **end while**
21: **return** EQ ▷ Return list of equivalence classes

program a is updated to a' (line 17) for the next iteration. After completing all m ITE iterations, the equivalence class E is added to the set of equivalence classes EQ (line 19). Finally, the algorithm returns the set of equivalence classes EQ (line 21), where each equivalence class contains a set of semantically equivalent assembly programs. The default value for m is set to 5, but our evaluation (Section 4.5) explores the impact of varying this parameter.

Table 1. Optimization and gate set change APIs for different platforms.

Platform	Optimizations	Change Gate Set

Qiskit	`transpile(opt_level=0)`
	`transpile(opt_level=1)`
	`transpile(opt_level=2)`
	`transpile(opt_level=3)`
		`transpile(basis_gates=..)`
		`['u1', 'u2', 'u3', 'cx']`
		`['u3', 'cx']`
		`['u3', 'cx']`
		`['rz', 'sx', 'x', 'cx']`
		`['rx', 'ry', 'rz', 'cz']`

Pytket	`FullPeepholeOptimise`
	`PeepholeOptimise2Q`
	`RemoveRedundancies`
	`OptimisePhaseGadgets`
		`AutoRebase(gateset=..)`
		`[OpType.U1, U2, U3, CX]`
		`[OpType.U3, CX]`
		`[OpType.RZ, SX, X, CX]`
		`[OpType.RX, RY, RZ, CZ]`

PennyLane	`cancel_inverses`
	`commute_controlled`
	`merge_rotations`
	`single_qubit_fusion`
	`unitary_to_rot`
	`remove_barrier`
	`undo_swaps`
	`combine_global_phases`
	`clifford_t_decomposition`
	`merge_amplitude_embedding`
	`defer_measurements`
		`decompose`

BQSKit	`compile( opt_level=0 )`
	`compile( opt_level=1 )`
	`compile( opt_level=2 )`
	`compile( opt_level=3 )`
		N/A

Cirq	`drop_empty_moments`
	`eject_z`
	`eject_phased_paulis`
	`align_left`
	`align_right`
	`drop_negligible_operations`
		N/A

Table 1 summarizes the transformations T_p available for the five platforms supported by IterTestQ: Qiskit, Pytket, PennyLane, BQSKit, and Cirq. To ensure semantic equivalence, the ITE process employs two main types of transformations: optimization passes and gate set conversions. Each transformation takes a circuit in the platform's internal representation and returns a semantically equivalent circuit in the same representation. Optimization passes aim to reduce the number of gates in the circuit, while gate set conversions adapt the circuit to a different gate set. As there are (theoretically) infinitely many universal gate sets, we select a few commonly used ones for gate set conversions, such as `['u3', 'cx']` and `['rz', 'sx', 'x', 'cx']`. Using commonly used gate sets means that IterTestQ tests widely used transformations and ensures that a program transformed by one platform can be imported and transformed by other platforms. Importantly, all these transformations are supposed to be semantics-preserving, i.e., they do not change the behavior of the program.

For Qiskit [21], IterTestQ uses the `transpile` API to apply transformations such as optimizing the circuit at different levels (e.g., `opt_level=1`) or changing the gate set to a universal gate set composed of `['u3', 'cx']` gates, an example of the effect of which is shown in Figure 2b. For PennyLane [4],

IterTestQ uses circuit transformations such as `cancel_inverses` to simplify the circuit by canceling inverse gates and `merge_rotations` to combine consecutive rotation gates. For Pytket [52], IterTestQ uses optimization techniques such as `FullPeepholeOptimise` to perform peephole optimization and `AutoRebase` to perform gate set transformations. For BQSKit, IterTestQ leverages the predefined optimization levels offered by the `compile` function, e.g., `compile(opt_level=2)`. For Cirq [1], IterTestQ wraps built-in circuit transformers such as `drop_empty_moments`, `eject_z`, `eject_phased_paulis`, `align_left`, `align_right`, and `drop_negligible_operations`. Since Cirq's QASM importer has stricter dialect assumptions than some other platforms, the Cirq processor also applies lightweight QASM normalization before import, including rewriting Qiskit-style `u(...)` operations to `u3(...)`.

IterTestQ's design allows for easy extensibility to new quantum computing platforms. To apply the approach to a new platform, the platform must provide three key components: an importer, an exporter, and optionally, one or more transformations.

3.5 Bug Detection

The bug detection component is active during all the previous stages of IterTestQ. It employs two complementary oracles to detect potential bugs. The *crash oracle* is always active, monitoring the platform code for crashes. Specifically, crashes may occur when the generator uses a platform's converter and when the ITE process invokes a platform's importer, transformations, and exporter. The *equivalence oracle* is periodically triggered when the ITE process completes a given number of ITE iterations on a given set of initial assembly programs. The oracle is based on QCEC [46], an existing equivalence checker for quantum circuits. The checker gets the set of equivalence classes EQ generated by the ITE process and compares pairs of programs within each equivalence class E to ensure semantic equivalence. Since all programs in E are constructed to be semantically equivalent, any discrepancies detected by the equivalence checker signal a bug. We call each pair of programs flagged as not semantically equivalent by the equivalence oracle an *inequivalence*. To address the scalability issue of comparing all pairs of programs within an equivalence class, IterTestQ randomly samples a small subset of program pairs for comparison (by default, five pairs per equivalence class).

Our equivalence oracle differs from differential testing oracles that compare program behavior across different optimization levels or simulator backends [19, 26, 56] because it does not execute the programs. Instead, it uses the symbolic equivalence-checking method of QCEC, which translates the two quantum circuits into ZX-diagrams and symbolically checks whether the composition of one circuit's inverse with the other simplifies to the identity [46]. The advantage of this approach is that it can detect semantic discrepancies without relying on specific input states or measurement outcomes, which are inherently probabilistic in quantum computing and can lead to false positives or negatives in differential testing.

To streamline the inspection of detected bugs, the approach heuristically reduces bug-triggering programs. When IterTestQ detects a potential issue, it records the exception or the specific discrepancy reported by the equivalence checker. Then, IterTestQ uses delta debugging [64] to reduce the program to the smallest test case that still triggers the issue. Specifically, IterTestQ uses the exception message as the signal for crashes and the returned string `not equivalent` reported by the equivalence checker as the signal for inequivalences. To ensure reproducibility and allow for automated delta debugging, IterTestQ's metadata tracks the provenance of each generated program, including the sequence of transformations applied during the ITE process. This allows us to replay the exact steps leading to a bug. While the heuristic reduction cannot guarantee that the final reduced program (or pair of reduced programs) preserves the original root cause, a minimal program (or pair of programs) that triggers a crash or inequivalence remains valuable to developers.

To prioritize bug inspection and reduce redundancy, IterTestQ groups warnings based on their error messages, assuming that identical messages indicate the same underlying bug. Then, we sample and inspect one warning per cluster, representing a unique error message. In the case of equivalence discrepancies, where no explicit error message is available, we sample and inspect a subset of the generated programs.

4 Evaluation

We investigate the following research questions:

- **RQ1: Bug detection**. How effective is IterTestQ at finding bugs in quantum computing platforms?
- **RQ2: Impact of ITE iterations**. What is the impact of the number m of ITE iterations on the generated programs and the number of detected crashes and inconsistencies?
- **RQ3: Efficiency**. How efficient are the different components of IterTestQ?
- **RQ4: Ablation study**. How does the choice of program generators and transformations affect the effectiveness of IterTestQ?
- **RQ5: Comparison with prior work**. How does IterTestQ compare with prior tools for testing quantum computing platforms?

4.1 Experimental Setup

We evaluate IterTestQ on five widely-used quantum computing platforms: Qiskit [21] (Qiskit version 1.2.4, Qiskit Aer version 0.15.1, and Qiskit IBM Runtime version 0.29.0), PennyLane [4] (PennyLane version 0.40.0 and PennyLane-qiskit version 0.40.0), Pytket [52] (Pytket version 1.33.1 and Pytket-qiskit version 0.56.0), BQSKit [63] (version 1.2.0), and Cirq [1] (version 1.5.0). While we consider the BQSKit platform for RQ1 to assess the bug-finding effectiveness of IterTestQ, we exclude it from subsequent research questions because its exporter produces many invalid assembly programs due to a single bug, which disrupts the continuous application of ITE iterations. Moreover, its execution often reaches timeout thresholds, likely due to its pure Python implementation without performance optimizations in systems languages, unlike Qiskit's Rust or Pytket's C++ components. For the equivalence oracle, we use MQT QCEC [46] version 2.7.1. We conduct all experiments on a dedicated machine equipped with 48 CPU cores (Intel Xeon Silver, 2.20GHz), two NVIDIA Tesla T4 GPUs with 16GB of memory each, and 252GB of RAM, running Ubuntu 22.04.5 LTS.

4.2 RQ1: Bug Detection

We evaluate the effectiveness of IterTestQ in detecting crashes and inconsistencies across five quantum computing platforms: Qiskit, PennyLane, Pytket, BQSKit, and Cirq. We measure IterTestQ's effectiveness by the number of unique bugs found, the number confirmed and fixed by developers, and developer feedback on GitHub issues we create. The results for this RQ are from several runs of IterTestQ performed over a period of three months.

Table 2 summarizes the bugs found across the different platforms. For each bug, we provide the platform and GitHub issue number, a brief description, its current status, and the oracle (crash or equivalence) that detected it. The results show that IterTestQ effectively identifies bugs, uncovering 23 issues across the evaluated platforms, with 17 already confirmed or fixed, demonstrating its practical impact and the interest by platform maintainers to address these issues.

In addition to the bug illustrated in Listings 1a and 1b, which has been confirmed by the Pytket developers, we describe two more representative bugs discovered by IterTestQ. Figure 6 demonstrates bug #5 in BQSKit, where the platform omits the definition of a controlled phase gate (`cs`) when exporting to QASM. The figure contrasts the original QASM code containing the gate definition with BQSKit's exported version that lacks this crucial definition. When reimporting with

Table 2. Summary of bugs found in different platforms.

Id	Issue	Platform	Short description	Status	Oracle
#1	#6323	PennyLane	Unexpected reallocation of gates to different qubits	Confirmed	Equiv.
#2	#12124	Qiskit	Missing standard `rxx` gate definition	Confirmed	Crash
#3	#307	BQSKit	Unrecognized delay gate	Confirmed	Crash
#4	#1629	Pytket	Error converting decomposed circuit (`PhasedX` op)	Confirmed	Crash
#5	#306	BQSKit	Missing controlled phase `cs` gate definition	Confirmed	Crash
#6	#6900	PennyLane	Error exporting QASM file: `MidMeasureMP` op	Fixed	Crash
#7	#6942	PennyLane	Unitary matrix mismatch in `cu` gate implementation	Confirmed	Equiv.
#8	#1771	Pytket	Incompat. of `PhasedX` gate with `GreedyPauliSimp` opt	Confirmed	Equiv.
#9	#1652	Pytket	QASM register reordering causes semantic changes	Confirmed	Equiv.
#10	#1605	Pytket	Unsupported `ISwap` gate	Fixed	Crash
#11	#1747	Pytket	`ZXGraphlikeOpt` pass creates non-equivalent circuit	Fixed	Equiv.
#12	#1750	Pytket	`FullPeepholeOpt` pass fails with `mcrz` gate	Fixed	Crash
#13	#1751	Pytket	Incorrect `MCX` gate generation (`c4x` instead of `c3x`)	Fixed	Crash
#14	#1768	Pytket	Incorrect `u0` gate conversion to `u3`	Fixed	Crash
#15	#1770	Pytket	`FullPeepholeOpt` pass fails with `u0` and `c4x` gates	Fixed	Crash
#16	#302	BQSKit	Duplicate classical register definition	Reported	Crash
#17	#298	BQSKit	Crash with DaggerGate	Reported	Crash
#18	#299	BQSKit	Missing Ryy gate definition	Reported	Crash
#19	#8087	Cirq	QASM importer rejects lowercase `u` gate	Fixed	Crash
#20	#9499	PennyLane	`merge_amplitude_embedding` inequivalence	Reported	Equiv.
#21	#2167	Pytket	`GreedyPauliSimp` changes circuit semantics	Confirmed	Equiv.
#22	#2169	Pytket	`PeepholeOptimise2Q` changes circuit semantics	Reported	Equiv.
#23	#9510	PennyLane	`decompose` pass on a `CSX` gate	Reported	Equiv.

```
OPENQASM 2.0;
include "qelib1.inc";
gate cs q0,q1 {
p(pi/4) q0; cx q0,q1;
p(-pi/4) q1; cx q0,q1;
p(pi/4) q1; }
qreg q[5];
cs q[4],q[0];
```

(a) Original QASM with `cs` gate definition.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
cs q[4], q[0];
# Error: 'cs' is not defined
# in this scope
```

(b) BQSKit exported QASM with missing `cs` gate definition.

Fig. 6. BQSKit omits the controlled phase gate (`cs`) definition when exporting to QASM (Bug #5).

BQSKit itself, this omission causes no issues, masking the bug in single-platform testing. However, when the QASM file is used with other platforms that require explicit gate definitions, the missing definition leads to compilation errors. The BQSKit developers acknowledged this as a complex issue that involves balancing standard gate definitions with compilation efficiency.

Figure 7 shows bug #2, where a valid QASM generated by Pytket includes an `rxx` gate. Figure 7b shows how Qiskit fails to recognize the `rxx` gate when importing the QASM code, raising a parse error. This bug represents a known limitation of Qiskit's importer, which becomes apparent through cross-platform testing. While the developers have provided a workaround using `qasm2.loads` with

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[11];
rxx(1.1761910836010856*pi) q[3],q[5];
```

(a) Valid QASM generated by Pytket with `rxx` gate.

```
from qiskit.qasm2 import load
new_qc = load("valid_pytket_with_rxx.qasm")
# Error when importing with other platform:
# QASM2ParseError: "<input>:3,27: 'rxx' is not defined in this scope"
```

(b) Crash when importing with Qiskit importer.

Fig. 7. Pytket generates valid QASM with `rxx` gate, but Qiskit's importer fails to recognize it (Bug #2).

argument `custom_instructions=LEGACY_CUSTOM_INSTRUCTIONS` rather than fixing the issue directly, this case demonstrates how IterTestQ effectively identifies gaps in platform interoperability.

Overall, IterTestQ finds both inconsistencies in the translation process from platform-specific representations to and from QASM and inequivalences between the resulting program pairs. While the primary goal of IterTestQ is to detect bugs in the tested platforms, it also identified three bugs in the QCEC equivalence checking tool upon which our approach builds. We reported these bugs to the QCEC developers, who confirmed the issues and provided fixes.

To avoid redundancy, we group multiple bugs pointing to the same root cause under a single GitHub issue. We report bugs only if they have not been reported previously; only one bug found by IterTestQ had been reported before. In a few cases (5), after reporting a bug, we had to make configuration changes to continue with the evaluation without repeatedly triggering the same bug. These changes included disabling specific gates, such as the `delay gate` (#3) or `iswap` (#10), avoiding the use of classical registers (#16), or adding workarounds suggested by developers to prevent recurring bugs (#2).

Answer to RQ1: IterTestQ effectively identifies bugs, uncovering 23 issues across the tested platforms, with 17 bugs already confirmed or fixed, demonstrating its practical impact.

4.3 RQ2: Impact of ITE Iterations

We investigate how iteratively translating quantum circuits across different platforms affects the characteristics of the generated test programs and their effectiveness in detecting bugs. One ITE iteration consists of creating an assembly program, importing it into a platform, applying a transformation, exporting to assembly, and adding the new assembly program to the equivalence class. The number of ITE iterations is represented by m in Algorithm 1.

To quantify the impact of ITE iterations, we measure:

- *Number of total gates per program:* This metric measures program complexity, as more complex programs are more likely to trigger subtle bugs.
- *Number of unique gates per program:* This metric captures the diversity of gates in the generated programs by counting the number of unique gate types. For example, a `cx` gate is counted once, even if a program applies this kind of gate to different qubits.
- *Entropy of n-grams of instructions per program:* We compute the entropy of pairs and triplets of consecutive instructions in the generated programs to measure program diversity. Higher entropy indicates more diverse programs. We consider instruction sequences as they appear in the assembly code since this is the input processed by the platforms.

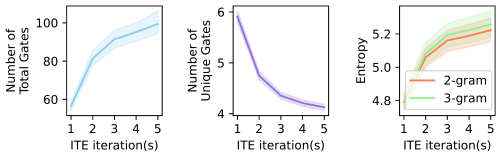


Fig. 8. Program diversity versus number of ITE iterations, showing the average number of total gates (left), the number of unique gates (center), and the entropy of instruction n-grams (right) per generated program.

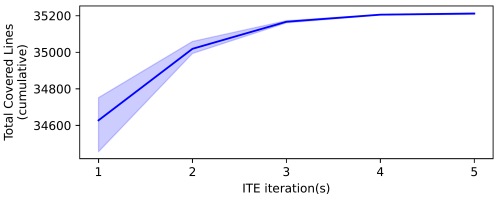


Fig. 9. Cumulative line coverage versus number of ITE iterations.

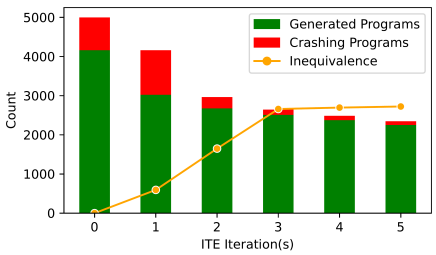


Fig. 10. Number of programs and warnings detected over ITE iterations. The number of warnings is not cumulative and zero iterations correspond to crashes during generation, including the Python-to-QASM conversion.

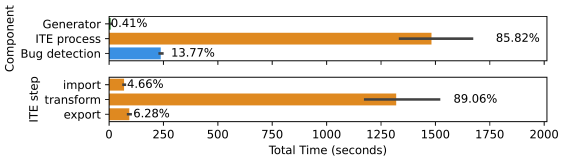


Fig. 11. Time analysis of IterTestQ in terms of components (top) and steps of the ITE process (bottom). Note the log scale.

- *Line coverage:* This metric assesses how much platform code is exercised by the generated programs.

We generate an initial set of 1,000 programs and perform five ITE iterations. We repeat this experiment five times, consistent with prior work [60], and then report the average and the confidence intervals. Note that, for any experiment run, IterTestQ randomly selects a platform for each single program transformation, thus distributing the testing effort across the tested platforms in a balanced manner.

Figure 8 shows how program diversity changes with the number of ITE iterations. Both the average number of total gates (left) and the entropy of instruction sequences (right) increase over iterations, with 3-grams showing slightly higher entropy levels but following similar trends. This indicates that the ITE process effectively generates more complex and diverse programs over time. The average number of unique gates (middle), however, follows the opposite trend, decreasing over iterations. This suggests that the platforms converge towards a common set of gates, which is expected due to the use of transformations that change the gate set to a small, universal gate set. Indeed, the target gate set size of the transformations contains between two to five gates, depending on the platform, significantly reducing the gate set size average shown in Figure 8.

Figure 9 shows the cumulative line coverage achieved on all platforms together across ITE iterations. As the number of iterations increases, the cumulative line coverage also increases, indicating that the ITE process continues to explore new code paths and potentially uncover additional bugs.

To better understand the impact of ITE iterations on program generation and bug detection, Figure 10 shows both the total number of programs and warnings detected over ITE iterations. At ITE iteration zero (initial generation), the process starts with direct program generation, including the Python-to-QASM conversion. While our target was to generate 5,000 programs (1,000 programs × 5 runs), the actual number is lower due to crashes during the conversion process - though these crashes themselves lead to the discovery of interesting bugs. The vertical axis of Figure 10 shows (i) the number of generated programs per iteration, distinguishing between those successfully stored as assembly programs (green bar) and those that cause a crash (red bar); and (ii) the number of inconsistent pairs detected by the equivalence checker (yellow line). Although IterTestQ usually runs the equivalence checker only at the end of the ITE process, for this experiment we run it after each ITE iteration to understand its behavior over time. The number of detected inconsistencies initially grows but then stabilizes and slightly decreases in later ITE iterations. This pattern emerges because the size of equivalence classes grows over time.

Answer to RQ2: The iterations of the ITE process are instrumental to effectively generate diverse test programs while increasing both program complexity and coverage.

4.4 RQ3: Efficiency of IterTestQ

We study the efficiency of IterTestQ in terms of time spent in each component of IterTestQ, i.e., in the generator, the ITE process, and bug detection, to generate and process a predefined number of programs. We also measure the time spent in each of the steps of the ITE process, where we consider the `import`, `transform`, `export` steps separately. These metrics are relevant because they provide insights into the computational cost of each phase and the overall rate at which IterTestQ can generate test cases. We use the same setup as used to study the ITE process in Section 4.3, consisting of generating 1,000 initial programs followed by $m = 5$ ITE iterations.

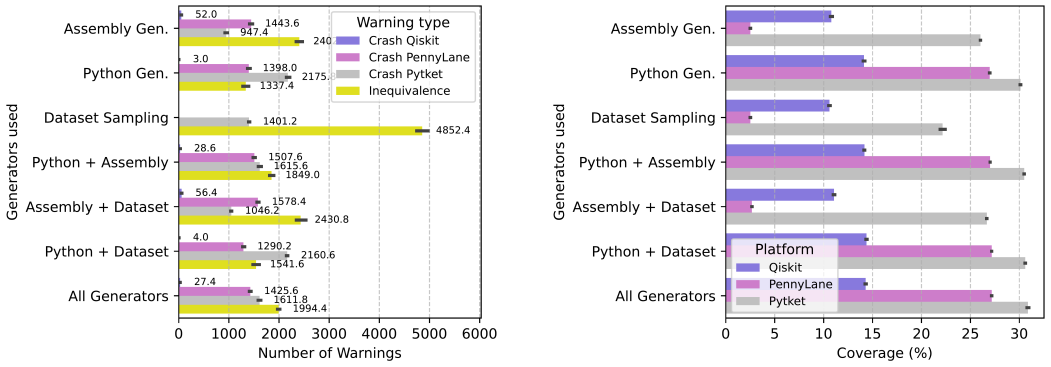
Figure 11 shows the time analysis of IterTestQ components on top, and the steps of the ITE process on the bottom. The results indicate that the ITE process consumes the most time, accounting for 85.82% of the total time of a typical run. The bug detection component, which includes the equivalence checker, consumes 13.77% of the total time, whereas the generator component is the fastest, accounting for only 0.41%. This is understandable since the ITE process component is the most complex and involves multiple steps and ITE iterations, including optimizations, while the generator component is based on a lightweight grammar-based approach.

In terms of steps of the ITE process, the `transform` stage consumes the most time, accounting for 88.19% of the total time, followed by the `export` stage (6.27%) and the `import` stage (4.66%). This is expected since the `transform` stage often involves complex transformations, the `export` stage involves occasional conversion and serialization of gates that are not directly supported in QASM, and the `import` stage involves a more straightforward parsing of QASM code, thus being the fastest.

Answer to RQ3: IterTestQ exhibits high efficiency, generating programs in just a fraction of a second (0.00089 seconds per program; generating 39,990 programs over five runs in 35.61 seconds), allowing computational resources to be focused on the ITE process and bug detection, which constitute 85.82% and 13.77% of the total time, respectively.

4.5 RQ4: Ablation Study

4.5.1 Impact of Program Generators. First, we evaluate the impact of the three program generators (Section 3.3) on the effectiveness of the approach. We study seven setups: each generator individually, all pairs of generators, and all three generators combined. For each setup, we generate 5,000 initial QASM programs and perform $m = 5$ ITE iterations. For a fair comparison, we configure all



(a) Bug detection by generator and platform.

(b) Coverage percentage by generator and platform.

Fig. 12. Ablation study results for different generators: (a) number of warnings detected and (b) coverage achieved across platforms.

generators to produce programs with a maximum of 11 qubits and 15 gates. Note that the "sampling from dataset" generator has a limited number of programs by design because it draws from a fixed dataset; once they are fully used, the remaining quota is filled by the other generators. For example in our setup, the MQTBench dataset is fully used after about 500 programs that satisfy the size constraints mentioned above. We measure the number of crash warnings, inconsistent pairs, and the code coverage achieved by each setup to assess their effectiveness. We repeat the experiments five times and report the average and confidence intervals of our measurements.

Figure 12 shows the results of this ablation study. In terms of triggering crashes, any combination involving the Python-based generator or the assembly-based generator is effective across all platforms, whereas the sampling from dataset strategy triggers crashes exclusively in Pytket. This might be due to the fact that the programs in MQTBench are known and could have been tested previously on Qiskit and PennyLane, thus reducing the likelihood of uncovering new crashes in those platforms. In terms of inconsistencies, the sampling from dataset strategy produces the highest number of inconsistencies. Based on inspecting ten inconsistent pairs, eight are linked to PennyLane's exporter, which fails to preserve the target classical register of a measurement operation. The remaining two inconsistencies are related to Pytket reordering classical registers in the QASM code. We report this as a bug (#9), which the Pytket developers have confirmed. Finally, in terms of code coverage, setups involving the Python-based generator achieve higher coverage because they exercise the converter code, covering more of the platform's code. The assembly-based and sampling from dataset strategy achieve similar coverage, with the assembly-based generator slightly outperforming the sampling from dataset strategy for Pytket. Based on these results, we conclude that all three generators contribute to the effectiveness of IterTestQ in complementary ways, justifying their combined use in our approach.

4.5.2 Impact of Transformations. Second, we evaluate the impact of the transformations applied by the ITE process on the number of crashes, inconsistencies, and on the code coverage. To this end, we compare IterTestQ, as described in Section 3, with a version where all transformations are disabled. We repeat the experiment five times and report the average percentage change. In terms of code coverage, the transformations lead to an increase in all platforms: 19.93% in Pytket, 0.91% in PennyLane, and 3.58% in Qiskit. We also quantify the impact of optimizers and gate set changes on code coverage. Using only gate set changes yields a modest coverage increase (4.4%), while

optimizers alone significantly boost coverage (42.8%). Combining both achieves a 43.7% increase, showing their complementary contributions.

4.5.3 Impact of Equivalence-Comparison Strategy. The equivalence oracle compares a subset of all pairs of programs within the same equivalence class to identify inconsistencies (Section 3.5) We compare two strategies for selecting these program pairs. For this experiment, we run IterTestQ for five ITE iterations on 1,000 starting programs to obtain 614 equivalence classes with at least two programs. For each class, the approach constructs a transformation chain of the initial program and intermediate programs produced after each iteration. The *first-last* strategy compares only chain endpoints, while the *random* strategy samples five pairs from each chain. The first-last strategy performs fewer comparisons (614 vs. 2,752) and detects fewer inconsistencies (367 vs. 826), as it can miss faults involving intermediate programs. We therefore use random pairing as the default in IterTestQ, and we keep first-last as a complementary configurable alternative.

Answer to RQ4: All three program generators contribute complementarily to IterTestQ's effectiveness. The transformations in the ITE process are helpful for increasing coverage. For equivalence checking, random pair selection is a better tradeoff than checking only the first and last programs in each transformation chain because endpoint-only checking misses some warnings.

4.6 RQ5: Comparison with Prior Work

We compare IterTestQ with prior approaches by running each approach for a total of 24 hours, using the same hardware setup for all approaches to ensure a fair comparison. The comparison includes two kinds of baselines: single-platform approaches and multi-platform approaches. *Single-platform approaches* target one platform under test. MorphQ [43] tests Qiskit with metamorphic transformations and output-based oracles, so we run it for 24 hours on Qiskit. Fuzz4All [60], currently available only for Qiskit, uses an LLM-based program generator with two separate models: a stronger model for automatically refining generation prompts through autoprompting, and a cheaper model for the large-scale fuzzing loop. To match the computational budget of the other approaches, we run Fuzz4All for 24 hours using local models, specifically Gemma 4 8B for autoprompting and StarCoder2 3B for fuzzing. The same-platform differential testing oracle of Fuzz4All uses dynamic analysis to identify circuits created during program execution, and then checks QASM self-roundtrips and optimization-level consistency.

Multi-platform approaches test multiple platforms, either sequentially or simultaneously. QuteFuzz [19] generates quantum programs with features such as control flow, subcircuits, and measurements to fuzz quantum libraries, so we run it for eight hours each on Qiskit, Pytket, and Cirq. QDiff [56] mutates related circuits and compares execution-output distributions, so we update it for our environment and run it for 12 hours each on Qiskit and Cirq. Because these baselines target Cirq, we added support for testing Cirq to IterTestQ. IterTestQ runs once for 24 hours while testing Qiskit, PennyLane, Pytket, and Cirq simultaneously. Note that the released baseline implementations were not designed for a Qiskit version as recent as the one used by IterTestQ, so we update their software stacks before this comparison to keep the platform versions aligned.

Table 3 summarizes the results. We report the total number of warnings and the kind of oracle that has revealed them. As inspecting all warnings is infeasible, we inspect a statistically relevant sample for each warning set, following prior bug-detection work that samples large warning populations and manually validates representative warnings [16]. A single author annotates warnings until reaching the sample size required for a 90% confidence level and a 10 percentage-point margin of error for each warning set, and clusters the sampled warnings by reduced program, warning

Table 3. RQ5 comparison with prior quantum testing tools under a 24-hour budget. "Other warnings" denotes non-crash anomalies, including inequivalences for IterTestQ and statistical divergences for tools that use output-distribution oracles. The inspected warning samples use a 90% confidence level and a 10 percentage-point margin of error.

Tool	Platforms	Budget	Programs	Warnings		Inspected Sample		
				Crash	Other	Sample Size	Unique Warnings	Bugs
Multi-platform								
IterTestQ	Qiskit, PennyLane, Pytket, Cirq	24h joint	243,886	61,014	46,050	68	33	8
QuteFuzz [19]	Qiskit, Pytket, Cirq	8h/plat.	14,460	84	30	43	5	3
QDiff [56]	Qiskit, Cirq	12h/plat.	6637	0	0	0	0	0
Single-platform								
MorphQ [43]	Qiskit	24h	62,378	11,967	16	68	6	2
Fuzz4All [60]	Qiskit	24h	2,413	3	0	3	1	0

message, and apparent root cause. The table also records confirmed or known bugs when the warning classes can be mapped to developer-confirmed issues or existing bug reports.

The fixed time budget also enables a direct comparative time analysis. Normalizing by the total 24-hour budget, IterTestQ processes 10,162 programs per hour, compared with 2,599 for MorphQ and 101 for Fuzz4All in the single-platform runs. For the multi-platform baselines, the total rates across their platform-specific runs are 603 programs per hour for QuteFuzz and 277 for QDiff. These rates show that IterTestQ generates the largest number of tested programs per hour even while distributing its ITE workflow across four platforms.

The last column of Table 3 compares the approaches in terms of the number of unique bugs revealed while inspecting warnings. It shows that IterTestQ identifies eight bugs, QuteFuzz three, MorphQ two, and neither QDiff nor Fuzz4All any in this comparison. QuteFuzz and IterTestQ both test multiple platforms, but use different strategies. QuteFuzz allocates its budget to fuzz each platform independently, while IterTestQ uses the same 24-hour budget to repeatedly pass programs across platform importers, transformations, and exporters to check equivalence classes. The bugs they expose in Cirq reveal this complementarity: QuteFuzz discovers an optimization bug where gates before measurements are folded into that measurement, causing later measurements of the same qubit to observe incorrect state. IterTestQ discovers a different interoperability bug: Qiskit emits lowercase `u` in QASM gate definitions, which Cirq's importer rejects. MorphQ identifies two pre-existing Qiskit bugs. We conjecture QDiff's lack of success in our setup stems from simplistic mutations, such as canceling gates already handled by recent platform versions. For Fuzz4All, using local models matched to our GPU capacity rather than frontier models may explain its lack of success, as larger LLMs are more effective.

Answer to RQ5: Under the same 24-hour budget, IterTestQ complements prior single- and multi-platform quantum testing tools by jointly exercising Qiskit, PennyLane, Pytket, and Cirq through ITE, processing the most programs per hour, and producing the largest number of unique warnings in the inspected samples.

5 Threats to Validity

We identify five main threats to the validity of our work. First, IterTestQ's applicability to new quantum computing platforms relies on their support for importing and exporting assembly code, particularly QASM, one of several quantum program representations. Although this is common in

major platforms, not all quantum computing platforms support it, limiting IterTestQ's generalizability. Cirq illustrates a related compatibility threat: even when a platform supports QASM import and export, small dialect differences may require local normalization, such as preserving unused qubits or translating `u` to `u3`. Second, IterTestQ depends on semantics-preserving transformations provided by the platforms. During our evaluation, we discovered that some platform-provided transformations documented as semantics-preserving actually change program semantics, so we manually check each transformation type. Third, if a platform is significantly buggier than others, it can affect our evaluation by prematurely terminating the ITE loop. For example, BQSKit's reliability issues led us to exclude it from RQ2–RQ5, as many transformation attempts failed due to platform-specific bugs. This suggests that IterTestQ is most effective when testing platforms of comparable reliability. Fourth, our implementation and experimental scripts may contain bugs that affect the reported results. We mitigate this risk through manual inspection of reduced failures and repeated experiments that report averages and confidence intervals. Fifth, our generated and sampled benchmark programs may not represent all quantum programs used in practice; we mitigate this risk by combining three different program generators, including one that samples from a real-world dataset.

6 Related Work

Quantum Testing and Debugging. Quantum software engineering [41, 66] and quantum software testing and analysis [45] are rapidly growing fields. Empirical studies highlight the prevalence of quantum-specific bugs in quantum software [33, 42]. To address these issues, researchers have proposed static analyses [24, 44, 67], dynamic analyses [7], delta debugging [47], mutation testing [37], and metamorphic testing [3]. Other approaches leverage runtime information from executions on quantum computers, using runtime projections and statistical assertions [18, 23, 28, 30]. Debugging quantum programs is particularly challenging [39], mainly due to the difficulty of measuring quantum states without disrupting computation. To address these challenges, researchers have proposed runtime assertions [28, 30], statistical assertions [18, 23], non-destructive discrimination methods [32], and interactive debugging tools [38]. While these approaches focus on quantum applications, IterTestQ targets quantum computing platforms.

Testing Quantum Computing Platforms. Several fuzzers address the challenges of testing quantum computing platforms. QDiff [56] performs differential testing [36] across optimization levels and backend configurations within a single platform, or across simulators [5]. MorphQ [43] uses metamorphic testing [8] to compare outputs across backend configurations in Qiskit. Fuzz4All [60] also targets Qiskit, employing a generator based on large language models and code examples. Unlike these single-platform approaches, IterTestQ is a multi-platform testing technique for quantum software. UPBEAT [17] targets language-level input validation bugs in Q# libraries. Our core contribution, the novel ITE process, can be combined with existing program generators, including those of MorphQ [43] and Fuzz4All [60]. FuzzQ [26] and work by Blackwell et al. [5] perform cross-platform differential testing, but unlike IterTestQ, they compare simulator behavior instead of testing cross-platform interoperability. A difference to all the above is that IterTestQ uses cross-platform interoperability as a means to generate new quantum programs.

Quantum Program Equivalence. Generating equivalent quantum programs is central to our equivalence oracle. Prior work has explored this, often at a smaller scale [61]. Mori et al. [40] propose unoptimization techniques to benchmark optimization tools. QDiff [56] applies gate equivalence transformations to create equivalent programs for cross-backend comparison within a single platform. While these approaches focus on single-platform equivalence, their transformations could be integrated into our ITE process. MorphQ [43] uses metamorphic transformations to generate

pairs of semantically equivalent programs, but not all are semantics-preserving, making them unsuitable for direct use in our approach. Other methods for creating equivalent circuits include circuit mapping techniques [12, 50, 54, 58, 65, 68], which could also be incorporated to transform circuits. Unlike prior work that relies on external code, our approach focuses on built-in platform transformations, making them part of the code under test. Equivalence checking is essential for verifying that program transformations preserve semantics, including QCEC [46], which is based on the ZX-calculus [11, 25], and the automata-based AutoQ [9, 10]. Giallar [55] verifies that compiler passes preserve the semantics of circuits, but unlike IterTestQ, focuses on a single optimization pass at a time.

Quantum Representation and Interoperability. A common intermediate representation (IR) is essential for cross-platform testing. While emerging IRs such as OpenQASM3 [13], Quil [53], QIR [2], and QASM-HL [22] exist, QASM remains the most widely adopted. Our approach leverages QASM2 for iterative import and export, combined with platform-specific transformations. The value of a common IR has also been demonstrated in deep learning testing [31], though it is typically used for one-way translation, unlike our iterative ITE process. Even with standardized IRs, interoperability challenges persist. For example, a recent study reports that 75% of defects in deep learning occur during operator conversion, with 33% leading to semantically inequivalent models [20]. Our work addresses similar challenges for quantum computing and QASM.

Compiler and Runtime Engine Testing. Our work relates to compiler testing [6], as quantum computing platforms conceptually resemble traditional compilers. Grammar-based generators have been used to test compilers and similar systems, including the Java Virtual Machine [51], C compilers [15, 62], SMT solvers [59], database engines [49], and deep learning infrastructure [31, 34, 35, 57]. However, prior compiler testing techniques do not address the unique challenges of cross-platform testing in quantum computing. Equivalence-modulo-input (EMI) [27] and work on testing the MLIR compiler infrastructure [29] shares the idea of testing transformation systems by producing equivalent programs. IterTestQ differs by deriving the equivalent programs via cross-platform interoperability. Unlike EMI, which produces programs that behave equivalently on a specific input, IterTestQ generates programs that are semantically equivalent across all inputs. This design is more suitable for testing quantum computing platforms where checking for input-specific equivalence is challenging due to the non-determinism of executing quantum programs.

7 Conclusion

We introduce IterTestQ, a novel cross-platform testing technique for quantum platforms. IterTestQ uses quantum assembly code for program generation and equivalence checking, enabling cross-platform comparisons. Evaluation on five platforms revealed 23 bugs, with 17 confirmed or fixed. The core component, our novel ITE process, generates diverse programs while testing multiple platforms simultaneously. Given the growing importance and diversity of quantum software platforms, IterTestQ provides an effective and efficient approach for quantum platform testing.

8 Data Availability

We make all data, source code, and experimental results publicly available at https://github.com/sola-st/IterTestQ-quantum-platform-testing/ and https://doi.org/10.6084/m9.figshare.33016415.

9 Acknowledgments

This work was supported by the German Research Foundation (DFG; projects 492507603, 516334526, and 526259073).

References

[1] 2021. Quantumlib/Cirq. quantumlib.

[2] 2025. Qir-Alliance/Qir-Spec. QIR Alliance.

[3] 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)*. 16–23. doi:10.1145/3528230.3529189

[4] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, M. Sohaib Alam, Shahnawaz Ahmed, Juan Miguel Arrazola, Carsten Blank, Alain Delgado, Soran Jahangiri, Keri McKiernan, Johannes Jakob Meyer, Zeyue Niu, Antal Száva, and Nathan Killoran. 2020. PennyLane: Automatic Differentiation of Hybrid Quantum-Classical Computations. *arXiv:1811.04968 [physics, physics:quant-ph]* (Feb. 2020). arXiv:1811.04968 [physics, physics:quant-ph]

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

[6] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surveys* 53, 1 (May 2020), 1–36. doi:10.1145/3363562

[7] Qihong Chen, Rúben Câmara, José Campos, André Souto, and Iftekhar Ahmed. 2023. The Smelly Eight: An Empirical Study on the Prevalence of Code Smells in Quantum Computing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 358–370. doi:10.1109/ICSE48619.2023.00041

[8] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. arXiv:2002.12543 [cs] doi:10.48550/arXiv.2002.12543

[9] Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, and Wei-Lun Tsai. 2023. AutoQ: An Automata-Based Quantum Circuit Verifier. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 139–153. doi:10.1007/978-3-031-37709-9_7

[10] Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023. An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 156:1218–156:1243. doi:10.1145/3591270

[11] Bob Coecke and Ross Duncan. 2011. Interacting Quantum Observables: Categorical Algebra and Diagrammatics. *New Journal of Physics* 13, 4 (April 2011), 043016. doi:10.1088/1367-2630/13/4/043016

[12] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. 2019. On the Qubit Routing Problem. In *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 135)*, Wim van Dam and Laura Mančinska (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:32. doi:10.4230/LIPIcs.TQC.2019.5

[13] 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. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3 (Sept. 2022), 12:1–12:50. doi:10.1145/3505636

[14] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. *arXiv:1707.03429 [quant-ph]* (July 2017). arXiv:1707.03429 [quant-ph]

[15] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. doi:10.1145/3597926.3598130

[16] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 945–956. doi:10.1109/ICSE48619.2023.00087

[17] Tianmin Hu, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Huanting Wang, Meng Li, and Zheng Wang. 2024. UPBEAT: Test Input Checks of Q# Quantum Libraries. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 186–198. doi:10.1145/3650212.3652120

[18] Yipeng Huang and Margaret Martonosi. 2019. Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 541–553. doi:10.1145/3307650.3322213

[19] Ilan Iwumbwe, Benny Zong Liu, and John Wickerson. 2025. QuteFuzz: Fuzzing Quantum Compilers Using Randomly Generated Circuits with Control Flow and Subcircuits. *Workshop Programming Languages for Quantum Computing (PLanQC)* (2025).

[20] Purvish Jajal, Wenxin Jiang, Arav Tewari, Erik Kocinare, Joseph Woo, Anusha Sarraf, Yung-Hsiang Lu, George K. Thiruvathukal, and James C. Davis. 2024. Interoperability in Deep Learning: A User Survey and Failure Analysis of

ONNX Model Converters. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1466–1478. doi:10.1145/3650212.3680374

[21] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum Computing with Qiskit. arXiv:2405.08810 [quant-ph] doi:10.48550/arXiv.2405.08810

[22] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2014. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/2597917.2597939

[23] 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 (2024), 140–164. doi:10.1145/3689716

[24] Maximilian Kaul, Alexander Küchler, and Christian Banse. 2023. A Uniform Representation of Classical and Quantum Source Code for Static Code Analysis. arXiv:2308.06113 [cs] doi:10.48550/arXiv.2308.06113

[25] Aleks Kissinger and John van de Wetering. 2020. PyZX: Large Scale Automated Diagrammatic Reasoning. *Electronic Proceedings in Theoretical Computer Science* 318 (May 2020), 229–241. arXiv:1904.04735 doi:10.4204/EPTCS.318.14

[26] Vasileios Klimis, Avner Bensoussan, Elena Chachkarova, Karine Even-Mendoza, Sophie Fortz, and Connor Lenihan. 2025. Shaking Up Quantum Simulators with Fuzzing and Rigour. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 1400–1428.

[27] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. *SIGPLAN Not.* 49, 6 (June 2014), 216–226. doi:10.1145/2666356.2594334

[28] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 150:1–150:29. doi:10.1145/3428218

[29] Jingjing Liang, Shan Huang, and Ting Su. 2025. Finding Bugs in MLIR Compiler Infrastructure via Lowering Space Exploration. In *ASE*.

[30] Ji Liu, Gregory T. Byrd, and Huiyang Zhou. 2020. Quantum Circuits for Dynamic Runtime Assertions in Quantum Computation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1017–1030. doi:10.1145/3373376.3378488

[31] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. doi:10.1145/3575693.3575707

[32] Ji Liu and Huiyang Zhou. 2021. Systematic Approaches for Precise and Approximate Quantum State Runtime Assertion. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 179–193. doi:10.1109/HPCA51647.2021.00025

[33] Junjie Luo, Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. 2022. A Comprehensive Study of Bug Fixes in Quantum Programs. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1239–1246. doi:10.1109/SANER53432.2022.00147

[34] Weisi Luo, Dong Chai, Xiaoyue Ruan, Jiang Wang, Chunrong Fang, and Zhenyu Chen. 2021. Graph-based fuzz testing for deep learning inference engines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 288–299.

[35] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing Deep Learning Compilers with HirGen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 248–260. doi:10.1145/3597926.3598053

[36] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[37] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. 2022. Muskit: A Mutation Analysis Tool for Quantum Software Testing. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE '21)*. IEEE Press, Melbourne, Australia, 1266–1270. doi:10.1109/ASE51524.2021.9678563

[38] Sara Ayman Metwalli and Rodney Van Meter. 2023. Cirquo: A Suite For Testing and Debugging Quantum Programs. arXiv:2311.18202 [quant-ph] doi:10.48550/arXiv.2311.18202

[39] Andriy Miranskyy, Lei Zhang, and Javad Doliskani. 2021. On Testing and Debugging Quantum Software. *arXiv:2103.09172 [quant-ph]* (March 2021). arXiv:2103.09172 [quant-ph]

[40] Yusei Mori, Hideaki Hakoshima, Kyohei Sudo, Toshio Mori, Kosuke Mitarai, and Keisuke Fujii. 2024. Quantum Circuit Unoptimization. arXiv:2311.03805 [quant-ph] doi:10.48550/arXiv.2311.03805

[41] Juan M. Murillo, Jose Garcia-Alonso, Enrique Moguel, Johanna Barzen, Frank Leymann, Shaukat Ali, Tao Yue, Paolo Arcaini, Ricardo Pérez-Castillo, Ignacio García Rodríguez de Guzmán, Mario Piattini, Antonio Ruiz-Cortés, Antonio Brogi, Jianjun Zhao, Andriy Miranskyy, and Manuel Wimmer. 2025. Quantum Software Engineering: Roadmap and Challenges Ahead. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025). doi:10.1145/3712002

[42] 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), 86:1–86:27. doi:10.1145/3527330

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

[44] Matteo Paltenghi and Michael Pradel. 2024. Analyzing Quantum Programs with LintQ: A Static Analysis Framework for Qiskit. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 95:2144–95:2166. doi:10.1145/3660802

[45] Matteo Paltenghi and Michael Pradel. 2024. A Survey on Testing and Analysis of Quantum Software. arXiv:2410.00650 doi:10.48550/arXiv.2410.00650

[46] Tom Peham, Lukas Burgholzer, and Robert Wille. 2022. Equivalence Checking of Quantum Circuits With the ZX-Calculus. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12, 3 (Sept. 2022), 662–675. doi:10.1109/JETCAS.2022.3202204

[47] Gabriel Joseph Pontolillo and Mohammad Reza Mousavi. 2024. Delta Debugging for Property-Based Regression Testing of Quantum Programs. In *Proceedings of the 5th ACM/IEEE International Workshop on Quantum Software Engineering (Q-SE 2024)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3643667.3648219

[48] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum* 7 (July 2023), 1062. doi:10.22331/q-2023-07-20-1062

[49] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.

[50] Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram. 2014. Qubit Placement to Minimize Communication Overhead in 2D Quantum Architectures. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 495–500. doi:10.1109/ASPDAC.2014.6742940

[51] Emin Gün Sirer and Brian N. Bershad. 2000. Using Production Grammars in Software Testing. *SIGPLAN Not.* 35, 1 (Dec. 2000), 1–13. doi:10.1145/331963.331965

[52] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. T|ket⟩: A Retargetable Compiler for NISQ Devices. *Quantum Science and Technology* 6, 1 (Nov. 2020), 014003. doi:10.1088/2058-9565/ab8e92

[53] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2017. A Practical Quantum Instruction Set Architecture. *arXiv:1608.03355 [quant-ph]* (Feb. 2017). arXiv:1608.03355 [quant-ph]

[54] Swamit S. Tannu and Moinuddin K. Qureshi. 2019. Not All Qubits Are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 987–999. doi:10.1145/3297858.3304007

[55] 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*. 641–656.

[56] 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)*. 692–704. doi:10.1109/ASE51524.2021.9678792

[57] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 788–799. doi:10.1145/3368089.3409761

[58] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. 2019. Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Number of SWAP and H Operations. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3316781.3317859

[59] Dominik Winterer and Zhendong Su. 2024. Validating SMT Solvers for Correctness and Performance via Grammar-Based Enumeration. *Proc. ACM Program. Lang.* 8, OOPSLA2 (Oct. 2024), 355:2378–355:2401. doi:10.1145/3689795

[60] 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 (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3597503.3639121

[61] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022. Quartz: Superoptimization of Quantum Circuits. In *Proceedings of the 43rd ACM*

SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022). Association for Computing Machinery, New York, NY, USA, 625–640. doi:10.1145/3519939.3523433

[62] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *ACM SIGPLAN Notices* 46, 6 (June 2011), 283–294. doi:10.1145/1993316.1993532

[63] Ed Younis, Costin C. Iancu, Wim Lavrijsen, Marc Davis, and Ethan Smith. 2021. *Berkeley Quantum Synthesis Toolkit (BQSKit) V1.* Technical Report Berkeley Quantum Synthesis Toolkit. Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA (United States). doi:10.11578/dc.20210603.2

[64] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. *ACM SIGSOFT Software Engineering Notes* 27, 6 (Nov. 2002), 1–10. doi:10.1145/605466.605468

[65] Chi Zhang, Ari B. Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z. Zhang. 2021. Time-Optimal Qubit Mapping. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 360–374. doi:10.1145/3445814.3446706

[66] Jianjun Zhao. 2021. Quantum Software Engineering: Landscapes and Horizons. arXiv:2007.07047 doi:10.48550/arXiv.2007.07047

[67] Pengzhan Zhao, Xiongfei Wu, Zhuo Li, and Jianjun Zhao. 2023. QChecker: Detecting Bugs in Quantum Programs via Static Analysis. arXiv:2304.04387 [cs] doi:10.48550/arXiv.2304.04387

[68] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2018. Efficient Mapping of Quantum Circuits to the IBM QX Architectures. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1135–1138. doi:10.23919/DATE.2018.8342181

Received 2026-01-27; accepted 2026-06-25