QuteFuzz: Fuzzing quantum compilers using randomly generated circuits with control flow and subcircuits

ILAN IWUMBWE, BENNY ZONG LIU, and JOHN WICKERSON, Imperial College London, UK

Recent advancements in quantum hardware have led to an increase in the number of quantum software stacks to allow developers to write, compile and run quantum programs. These quantum software stacks (QSS) are usually comprised of a high-level quantum programming language, a compiler to optimise quantum programs and convert them into quantum gate instructions, and a backend on which the program is run.

Despite these advancements, quantum processors are still in the Noisy Intermediate-Scale Quantum (NISQ) era, characterised by a limited number of logical qubits that are susceptible to noise that corrupts the state of the qubits. Therefore, quantum compilers are heavily relied upon to produce optimised circuits that can produce reliable results, making the correctness even more crucial.

We introduce a tool for detecting quantum compiler bugs. Unlike other similar tools that have been developed over the past few years, notably QDiff, QuteFuzz generates random quantum programs with higher-level abstractions like subroutines, and more complex circuit-level manipulation like control flows (i.e if-else, switch), all with varying depths of nesting and a variety of gates. These new generation strategies combine to allow for a breadth of test cases not achieved before and thus exposing bugs in previously unseen areas.

Seventeen bugs, including unexpected compiler crashes and silent miscompilations, were found in simulators and compilers used by Pytket, Qiskit, and Cirq combined. Some of these bugs have been verified and fixed by their respective developers, although some remain unfixed but acknowledged. These findings suggest that there is great potential in fuzzing quantum compilers, especially through the random generation of quantum circuits using more complex circuit elements.

1 INTRODUCTION

With the advancements seen in the quantum computing space, there is a greater push to allow users to easily write, compile and run quantum circuits on quantum hardware and simulators, led mainly by Pytket [20], Qiskit [11], and Cirq [2].

However, quantum processors are in the Noisy Intermediate-Scale Quantum (NISQ) era, which means that there aren't that many qubits to work with, and their states are easily corrupted by noise and decoherence. Despite this, researchers and enthusiasts get reliable results from their experiments thanks to quantum compilers, whose main role is to optimise the circuit by reducing two-qubit gate count and circuit depth.

Quantum compilers, much like their classical counterparts, are prone to mistakes that could alter the semantics of the circuit i.e., the result obtained is different from what the user expects. This issue, known as miscompilation, was a motivator for tools such as CSmith[4] to fuzz test C compilers. Additionally, work by Wang et al. [21] and a study by Paltenghi and Pradel [18] showed that quantum compilers can also crash, which creates a poor user experience for quantum compilers that should be avoided.

As such, several efforts have been made to fuzz test quantum compilers to detect miscompilations and crashing bugs. QDiff [21] is one such notable example, where quantum compilers were tested by applying semantics-changing and semantics-preserving mutations on 6 hand-written programs, generating 15,000 test cases divided into 730 equivalence classes. These circuits were then differential tested, which led to the discovery of bugs while also showcasing the feasibility of fuzz testing quantum compilers. Subsequent works like MorphQ [19] builds upon this work by generating

Authors' address: Ilan Iwumbwe, ilan.iwumbwe22@imperial.ac.uk; Benny Zong Liu, benny.liu21@imperial.ac.uk; John Wickerson, j.wickerson@imperial.ac.uk, Imperial College London, UK.

a large and diverse set of valid quantum programs with subroutines, onto which metamorphic mutations are made to produce 8000 program pairs.

QuteFuzz is a fully open-source tool that we have developed which specifically focuses on generating quantum circuits control flow (if-else, switch statements) and subroutines. This has led to the discovery of several bugs in widely used quantum compilers that existing tools would have missed. This paper will therefore dive into how QuteFuzz was developed and discuss some of the results that it has achieved.

The repository for QuteFuzz is on Github:

`https://github.com/QuteFuzz/QuteFuzz`

2 QUTEFUZZ

QuteFuzz is a quantum compiler fuzzer, which was built to generate a large number of semantically diverse test cases that test the limits of the compiler. QuteFuzz can generate circuits with subroutines and control flow, which are more complex quantum circuit elements that could increase the likelihood of exposing more bugs.

QuteFuzz consists of two main components: the program generator and the differential tester. The program generator, which is written in C++, generates Python code that adheres to the API given by the specific QSS being tested. The differential tester will then put the program through a quantum compiler, testing optimisation levels and specific optimisations passes and trying to find miscompilation bugs or catch crashes.

2.1 Random circuit generation

QuteFuzz's random program generator creates complex, randomized quantum circuits containing control flow and subcircuits to rigorously test compiler capabilities. In order to do that effectively and consistently across all three target QSSes, QuteFuzz was designed with modularity and generality in mind.

It is observed that a quantum circuit follows a general structure:

- Import the quantum circuit and qubit class abstraction
- Define any subcircuits to be used
- Define the top-level circuit that will be tested

The program generator, which follows the following key steps, is written in such a way that its core algorithms can be reused:

(1) Instantiate a quantum circuit object and choose a certain number of resources (qubits or bits) to use. While this is done, a struct `circuit_info` keeps track of all available resources that the circuit can use.
(2) Parameters that will be used in the circuit are defined. These will be used for parametric gates instead of literal constants and bound to constants after the entire circuit is generated.
(3) When a new gate is to be added to the circuit, a valid one is randomly chosen from the gateset stored in `circuit_info`. The gate will only be written into the circuit if there are enough qubit resources for it to use, and if adding it wouldn't go over the expected total gate count. Any random gate chosen isn't necessary a primitive gate, since all subcircuits are converted into gates and added into the top-level circuit's gateset.
(4) Bind all previously defined parameters used within the circuit to constant float values.

```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.circuit import Parameter, ParameterVector
from helpers.qiskit_helpers import run_pass_on_simulator
from pathlib import Path
from math import pi

subcirc0 = QuantumCircuit(0)
# Adding qregs
qreg_0 = QuantumRegister(4)
subcirc0.add_register(qreg_0)
# Adding creg resources
subcirc0.u(pi/2,-0.699000,-0.635000, qreg_0[2])
subcirc0.u(pi/2,0.573000,-0.226000, qreg_0[0])
subcirc0.u(-0.379000,-0.962000,0.418000, qreg_0[1])
subcirc0.ry(0.666000, qreg_0[2])
subcirc0.x(qreg_0[0])

main_circ = QuantumCircuit(2)
# Adding qregs
qreg_0 = QuantumRegister(1)
main_circ.add_register(qreg_0)
qreg_1 = QuantumRegister(3)
main_circ.add_register(qreg_1)
# Adding creg resources
creg_0 = ClassicalRegister(1)
main_circ.add_register(creg_0)
creg_1 = ClassicalRegister(1)
main_circ.add_register(creg_1)
# Adding symbols
param_0 = Parameter("param_0")
param_1 = Parameter("param_1")
param_2 = Parameter("param_2")

main_circ.cx(1, 0)
main_circ.x(2)
main_circ.x(0)
main_circ.cx(2, 0)
main_circ.measure(0, creg_1[0])
with main_circ.if_test((creg_1[0],0)) as else_2:
        main_circ.u(-0.969000,param_2,param_1, qreg_1[1])

with else_2:
        main_circ.measure(1, creg_1[0])
        with main_circ.if_test((creg_1[0],0)) as else_1:
                main_circ.h(0)
                main_circ.id(1)
                main_circ.append(subcirc0,[qreg_1[1],0,1,qreg_0[0]])
        with else_1:
                main_circ.u(param_0,0.076000,-0.803000, 1)
                main_circ.barrier(qreg_1[1])

bindings = {param_0: -0.127000, param_1: 0.233000, param_2: 0.733000,}
main_circ = main_circ.assign_parameters(bindings)

print(Path(__file__).name, " results:")
main_circ.measure_active()
run_pass_on_simulator(main_circ, 5, "HoareOptimizer")
```

Listing 1. Example Qiskit circuit generated by QuteFuzz

To explain the function of subcircuits and control flows, a brief overview summarising their functionalities is shown below:

- Control flow measures the state of a particular qubit and applies a different set of gate operations on another qubit, based on the result state of the qubit measurement.
- Subcircuits are abstractions of more complex quantum circuits, which can be called upon and appended to other quantum circuits without having to rewrite code. This greatly reduces the code length for repeated circuit sections and allows for compartmentalisation of different parts of circuits that may do different things.

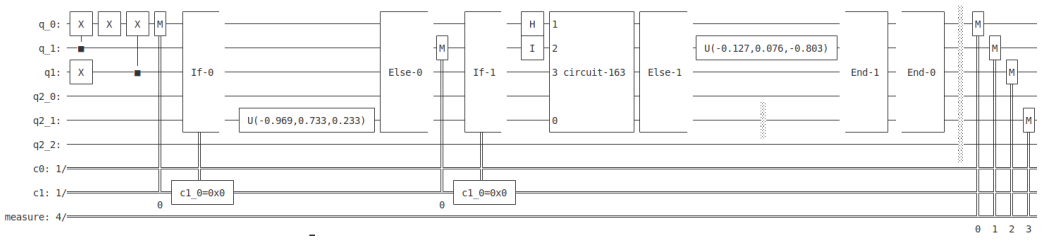


Fig. 1. Printed Qiskit circuit shown in Listing 1, containing control flow and subcircuits

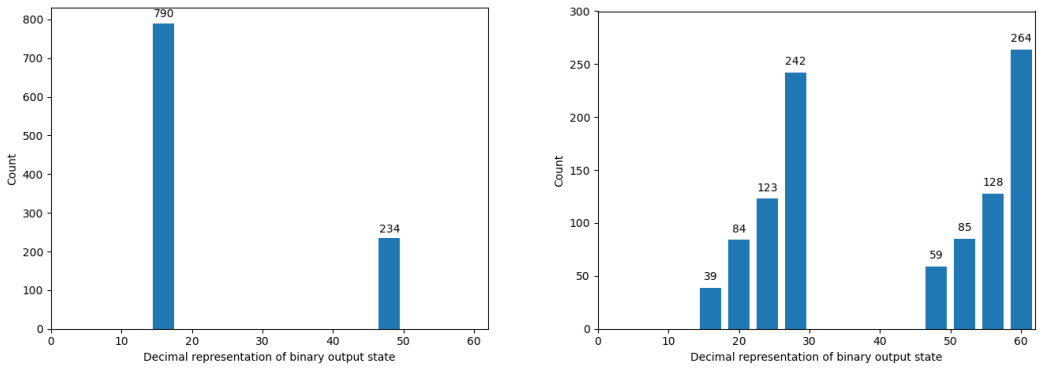
Shown in figure 1 is the Qiskit circuit in Listing 1. The circuit contains both control flow and subcircuits working together in a complex quantum circuit. It can be seen that there is an `If-1` block in the circuit, which checks if the classical bit `c0` is equal to `0x0`, after being measured with the `M` block located before it. If that is evaluated to be true, a subcircuit named `circuit-160`, which is an abstracted element of a more complex circuit section, will be applied to the qubits `q10` and `q11`, with qubits `q_0`, `q_1` and `q_2` controlling it. If not, the circuit elements enveloped between the `Else-1` and `End-1` blocks will be applied, which is simply a Hadamard gate `H` and an identity gate `I` applied to qubits `q_0` and `q_1` respectively.

Some circuit generation exceptions had to be made for some QSSes, due to severe limitations imposed by bugs or a lack of features. For example, control flow in Pytket was omitted because it was found to be buggy, making it impossible to generate circuits with this construct [9]. Since this bug was not fixed before the conclusion of this project, it was decided that control flow generation for Pytket would be removed.

During circuit generation, circuits also pick a subset of gates from a pool of available gates to use throughout the generation loop – an idea reminiscent of swarm-testing [5]. Additionally, the number of qubits, gates, subroutines and amount of nesting for Qiskit control flow is varied. These techniques were used together to allow for the exploration of various complex structural combinations of circuits while minimising the risk of generating overly average circuits, which would increase the likelihood of exposing more bugs.

2.2 Differential testing

QuteFuzz needs to find bugs in quantum compilers, and this is done through differential testing. Differential testing is where a single program is put through different compiler configurations, generating two differing programs that should produce the same output. The output of two programs is then compared, and a difference would indicate semantic difference between the two programs. When used on quantum programs, differential testing can help detect subtle miscompilation bugs that do not crash.



(a) No optimisation

(b) Optimisation level 3

Fig. 2. Example of differing Output distributions of two optimization levels due to miscompilation in Qiskit

These different compiler configurations could be applying a particular compiler pass or optimization level. In that case, a quantum circuit without any changes is used as ground truth, and its results are compared against its copy that has been optimised using the compiler configuration being tested. The comparison of their outputs can then be done in one of two ways:

(1) Run the original circuit and the optimised on a state-vector simulator, which gives two vectors that denote the probability distributions of results from the circuits. A dot product is taken between them, whose value should be 1 if the circuits semantics were unchanged. Any deviation would suggest possible miscompilations.
(2) Run the circuit on a shot-based simulator, which generate two sample distributions describing the resultant state of the circuits' qubits. Then by using the Kolmogorov–Smirnov test (KS test) [16], it can be determined if the circuits' output are equivalent and by extension, are produced by semantically identical circuits.

Shown in figure 2 is an example of differential testing using a shot-based simulator in Qiskit, showing the effect of miscompilation. These results are obtained by running the circuit in Listing 1. The graph labelled (a) shows the output distribution of a circuit without any changes, while (b) shows the output distribution of a modified circuit with optimisation level 3 applied onto it by the compiler. The resulting graphs differs greatly, which shows how the compiler had completely altered the semantics of the circuit. This discrepancy can be detected by QuteFuzz's differential tester and logged into a results file and analysed later.

3 RESULTS

In total, seventeen bugs have been found by QuteFuzz between the three tested QSSes, including bugs from simulators. These bugs include compiler crashes, miscompilations as well as simulator errors. Figure 3 shows a breakdown of the nature of the bugs found in all the QSSes tested. All the tests were run on an Intel-Xeon based server, over the period of up to 48 hours over multiple runs ranging from 50,000 to 100,000 circuits each. It is estimated that an excess of 500,000 circuits were run across all three QSSes for these bugs to be found. Since these bugs are rare, it was determined that manually searching through the output result file is sufficient for finding crashes and miscompilations. This is typically done by searching for keywords such as "error", "exception", or small decimal numbers with "e-" for KS tests.

As of this writing, 3 of the 17 bugs reported have been fixed, while the rest are acknowledged.

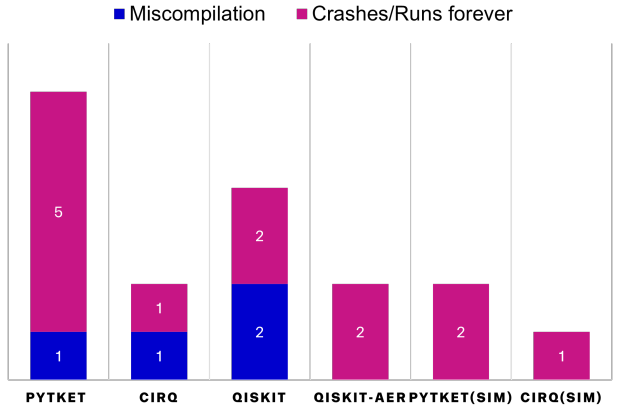


Fig. 3. Graph breakdown of bugs found

Some of the bugs that were found included bugs in the simulators. These are indicated in figure 3 by the three rightmost columns, which only contain crashing bugs.

One of the bugs we found was with the `HoareOptimizer` pass in Qiskit Terra. The root cause was identified to be a simplification step in the compiler; if the control qubit of a `CX` gate is 1, then the gate would be replaced with an `X` gate, but keep track of this mutation incorrectly [6]. The effect of this bug is shown Figure 2.

Another bug that changed the circuit semantics was introduced while refactoring code for the `ConsolidateBlocks` transpiler pass in Qiskit Terra, which is often used at higher optimization level configurations. Specifically, it was because two parameters had been swapped around accidentally, which wasn't caught during unit testing since they both had the same value.

Existance of the latter points to the need for tools in the same vein as work by Fortunato et al.[3] that can test the coverage of test suites used during development of these QSSes.

4 DISCUSSION

4.1 Analysis of new circuit generation methods

Our novel approach we use generates circuits with complex control flow and subroutines. These are some bugs that would not have been found otherwise.

4.1.1 Generating circuits with subroutines. The bugs below are examples that would not have been found if QuteFuzz did not generate circuits with subroutines.

- Tket's `KAKDecomposition` compiler pass: The compiler crashes when it applies this pass on a circuit that has 2 subroutines next to each other [13].
- Tket's `GlobalisePhasedX` compiler pass: This changed the circuit semantics when a subroutine was applied before a single-qubit gate on the same qubit [12].
- Cirq simulator crashes: When the circuit contains an empty subroutine, and is later applied with a classical control bit, the simulator would crash. [7]
- Qiskit's `ConsolidateBlocks` compiler pass: This bug was due to the fact that two parameters had been swapped around accidentally while refactoring code for this pass, which wasn't caught during unit testing since they both had the same value. [14]

These bugs are varied and seem to come from various parts of the compiler, suggesting the effectiveness of the testing strategies employed in hitting different parts of the compiler. Moreover,

the universality of subroutines across different QSSes means that subroutines can and should be an essential part of future quantum fuzzing tools.

4.1.2 Generating circuits with control flow. Circuits with control flow proved to be an important testing outlet as shown by the number of bugs that were found. These are a few examples:

- Generating nested control flow blocks allowed QuteFuzz to expose a transpiler bug in Qiskit [10] which would not have been caught otherwise.
- QuteFuzz utilizing classical operators in Pytket circuits helped expose a runtime bug stemming from incorrect type conversion[8] in the `symbol_substitution` construct.

The proportionally large number of bugs that are found from generating random circuits using these two features alone, suggests that this is an area for further exploration that could be fruitful. It is also crucial that these new features are tested, since the existence of tools like QuteFuzz could potentially increase the confidence of developers and help lead to more optimisations in these new areas.

4.2 Related Works

Unsurprisingly, this is a critical area of interest that has seen many works, which we attempt to expand and improve upon. Works like QDiff [21] and MorphQ[19] typically use semantics-changing mutations to generate their test cases. QuteFuzz extends upon the idea and greatly increases the diversity of the programs by introducing random circuit generation and introducing new circuit elements.

Inspired by several other works characterising bugs found typically in quantum software stacks, including a study [18] into the sources of bugs in quantum software stacks, QuteFuzz significantly extends upon some limitations of past attempts at fuzzing. Specifically, the diversity of the programs generated by QuteFuzz and the increase in circuit generation/execution speeds differentiates QuteFuzz from other similar works.

Measuring the effectiveness of QuteFuzz could be of an area of interest, as explored before by Fortunato [3]. Their work applied semantics-changing mutations to Qiskit programs in a test-suite, then checks are made to see if programs still pass tests. This would help to indicate how good the test-suite is, since semantics-changes should always be caught. Indeed, some of the bugs found by QuteFuzz point to the need for work like this that focuses on testing robustness of test suites.

4.3 Future Work

There are areas which QuteFuzz can improve. For example, a set of arbitrarily chosen ranges and values were used to control the circuit generation, such as the number of gates added in any subroutine and nested command depth, among others. Providing users with command-line options to modify these values could unlock ways of testing the scalability and robustness of quantum compilers.

Additionally, QuteFuzz's modularity which allows for the support of multiple QSS APIs, could be further improved. This could pave way to cross-QSS differential testing across different quantum compilers [17], serving as an open-source cross-platform benchmark.

Potentially more control over the critical path of the program could also be gained by generating the graph representation of a circuit before generating the circuit. This idea could expand even more for higher level languages like Q# and Guppy [15], where it might be a better idea to generate the Abstract Syntax Tree (AST) of the program instead.

Alternatively, QuteFuzz can be combined with concepts explored in work done by Daniel Blackwell et al. [1], which used a grammar-aware AFL fuzzer to generate syntactically valid quantum programs. This presents another approach to testing quantum compilers, which removes the need

to manually provide arbitrary parameters to generate random circuits and pave the way to highly automated and effective test-suites.

5 ACKNOWLEDGMENTS

This project was supported by the Undergraduate Research Opportunities Programme (UROP) at Imperial College London. We thank Michalis Pardalos for access to servers for running the circuits. We also thank George Constantinides for his support.

REFERENCES

[1] Daniel Blackwell, Justyna Petke, Yazhuo Cao, and Avner Bensoussan. 2024. Fuzzing-Based Differential Testing For Quantum Simulators. https://kclpure.kcl.ac.uk/ws/portalfiles/portal/270709916/SSBSE_24_Challenge_Track_Fuzzing_quantum_programs.pdf
[2] Cirq Developers. 2020. https://doi.org/10.5281/zenodo.4064322
[3] Daniel Fortunato et al. 2022. Mutation Testing of Quantum Programs: A Case Study With Qiskit. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9844849
[4] Xuejun Yang et al. 2011. Finding and Understanding bugs in C Compilers. https://users.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf
[5] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. [n. d.]. Swarm Testing. https://users.cs.utah.edu/~regehr/papers/swarm12.pdf
[6] Alexander Ivrii. 2024. Bug fix in HoareOptimizer. https://github.com/Qiskit/qiskit/pull/13083
[7] Ilan Iwumbwe. 2024. Controlled empty subroutine causes ValueError on simulator. https://github.com/quantumlib/Cirq/issues/6730
[8] Ilan Iwumbwe. 2024. Quantum circuit with conditionals can cause RunTime error. https://github.com/CQCL/tket/issues/1536
[9] Ilan Iwumbwe. 2024. Running circuit with classical condition can cause KeyError. https://github.com/CQCL/pytket-qiskit/issues/375
[10] Ilan Iwumbwe. 2024. Transpiler sometimes produces circuit that causes AerError. https://github.com/Qiskit/qiskit/issues/13162
[11] 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. https://doi.org/10.48550/arXiv.2405.08810 arXiv:2405.08810 [quant-ph]
[12] Benny Zong Liu. 2024. Circuit semantics changed when applying GlobalisePhasedX. https://github.com/CQCL/tket/issues/1554
[13] Benny Zong Liu. 2024. Runtime error when using KAKDecomposition on Circboxes. https://github.com/CQCL/tket/issues/1553
[14] Benny Zong Liu. 2024. Transpiler changes circuit semantics. https://github.com/Qiskit/qiskit/issues/13118
[15] Quantinuum Ltd. [n. d.]. https://github.com/CQCL/guppylang
[16] National Institute of Standards and Technology. [n. d.]. Kolmogorov-Smirnov Goodness-of-Fit Test. https://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm
[17] Matteo Paltenghi. 2022. Cross-Platform Testing of Quantum Computing Platforms. https://dl.acm.org/doi/abs/10.1145/3510454.3517061#:~:text=The%20final%20approach%20for%20cross,returned%20as%20the%20output%20of
[18] Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum Computing Platforms: An Empirical Study. https://arxiv.org/abs/2110.14560
[19] Matteo Paltenghi and Michael Pradel. 2023. MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform. https://arxiv.org/pdf/2206.01111
[20] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. A Retargetable Compiler for NISQ Devices. *IOPScience* (2020).
[21] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks p. https://web.cs.ucla.edu/~miryung/Publications/ase2021-qdiff.pdf