Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/TQE.2020.DOI

Mutation Testing of Quantum Programs: A Case Study with QISKit

DANIEL FORTUNATO[1,2,3], JOSÉ CAMPOS[1,4], RUI ABREU[1,2]
[1]Faculty of Engineering of University of Porto, Portugal
[2]INESC-ID, Lisboa, Portugal
[3]LIACC - Artificial Intelligence and Computer Science Laboratory (member of LASI LA), Porto, Portugal
[4]LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

Corresponding author: Daniel Fortunato (email: daniel.b.fortunato@tecnico.ulisboa.pt).

This work was supported in part by FCT/MCTES through projects ref. PTDC/CCI-COM/29300/2017 and CMU/TIC/0064/2019, and the research units LASIGE (ref. UIDB/00408/2020 and UIDP/00408/2020), INESC-ID (ref. UIDB/50021/2020), and LIACC (ref. UIDB/00027/2020).

ABSTRACT As quantum computing is still in its infancy, there is an inherent lack of knowledge and technology to test a quantum program properly. In the classical realm, mutation testing has been successfully used to evaluate how well a program's test suite detects seeded faults (i.e., mutants). In this paper, building on the definition of syntactically equivalent quantum operations, we propose a novel set of mutation operators to generate mutants based on qubit measurements and quantum gates. To ease the adoption of quantum mutation testing, we further propose QMutPy, an extension of the well-known and fully automated open-source mutation tool MutPy. To evaluate QMutPy's performance, we conducted a case study on 24 real quantum programs written in IBM's Qiskit library. Furthermore, we show how better test suite coverage and improvements to test assertions can increase the test suites' mutation score and quality. QMutPy has proven to be an effective quantum mutation tool, providing insight into the current state of quantum tests and how to improve them.

INDEX TERMS Quantum computing, Quantum software engineering, Quantum software testing, Quantum mutation testing

I. INTRODUCTION

QUANTUM computation uses the quantum bit (qubit)—the quantum-mechanical analog of the classical bit—as its fundamental unit instead of the classical computing bit. Whereas classical bits can take on only one of two basic states (e.g., 0 or 1), qubits can take on superpositions of those basic states (e.g., $\alpha \cdot |0\rangle + \beta \cdot |1\rangle$), where α and β are complex scalars such that $|\alpha|^2 + |\beta|^2 = 1$, allowing a number of qubits to theoretically hold exponentially more information than the same number of classical bits. Thus, quantum computers can, in theory, quickly solve problems that would be extremely difficult for classical computers. Such computation is possible because of qubit properties such as superposition of both 0 and 1, entanglement of multiple qubits, and interference [1, 2].

The field of quantum computing is evolving at a pace faster than originally anticipated [3]. For example, in March 2020, Honeywell announced[1] a revolutionary quantum computer based on trapped-ion technology with quantum volume 64—the highest quantum volume ever achieved, twice the state of the art previously accomplished by IBM. Quantum volume is a unit of measure indicating the fidelity of a quantum system. This important achievement shows that the field of quantum computing may reach industrial impact much sooner than initially anticipated.

While the fast approaching universal access to quantum computers is bound to break several computation limitations that have lasted for decades, it is also bound to pose major challenges for many, if not all, computer science disciplines [4], e.g., software testing. Testing is one of the most used techniques in software development to ensure software quality [5, 6]. It refers to the execution of software in *in vitro* environments that replicate (as close as possible) real scenarios to ascertain its correct behavior. Despite the

[1]https://www.honeywell.com/us/en/press/2020/03/honeywell-achieves-breakthrough-that-will-enable-the-worlds-most-powerful-quantum-computer

fact that, in the classical computing realm, testing has been extensively investigated, and several approaches and tools have been proposed [7, 8, 9, 10, 11, 12], such approaches for Quantum Programs (QPs) are still in their infancy [13, 14, 15]. It is worth noting that (i) QPs are much harder to develop than classical programs and therefore, programmers, mostly familiar with the classical world, are more likely to make mistakes in the counter-intuitive quantum programming one [16], and (ii) QPs are necessarily probabilistic and impossible to examine without disrupting execution or without compromising their results [17]. Thus, ensuring a correct implementation of a QP is even harder in the quantum computing realm [18].

Mutation testing [19, 20] has been shown to be an effective technique in improving testing practices, hence helping to guarantee program correctness. Big tech companies, such as Google and Facebook, have conducted several studies [21, 22, 23] advocating for mutation testing and its benefits. The general principle underlying mutation testing is that the bugs considered to generate buggy program versions represent realistic mistakes that programmers often make. Such bugs are deliberately seeded into the original program by simple syntactic changes to create a set of buggy programs called mutants, each containing a different syntactic change. To assess the effectiveness of a test suite at detecting mutants, these mutants are executed against the input test suite. If the result of running a mutant is different from the result of running the original program for at least one test case in the input test suite, the seeded bug denoted by the mutant is considered detected or killed.

Just et al. [24] performed a study on whether mutants are a valid substitute for real bugs in classical software testing, and they concluded that (1) test suites that kill more mutants have a higher real bug detection rate, (2) mutation score is a better predictor of test suites' real bug detection rate than code coverage. We have no reason to believe that it would be any different in quantum computing. Thus, and to shed light on whether manually-written test suites for QPs are effective at detecting mistakes that programmers might often make, in this paper, we aim to investigate the application of mutation testing on real QPs.

In this paper, we focus our investigation on the most popular open-source full-stack library for quantum computing [25], IBM's Quantum Information Software Kit (Qiskit) [26][2]. Qiskit is one of the first software development kits for quantum to be released publicly and provides tools to develop and run QPs on either prototype quantum devices on IBM Quantum Experience infrastructure or simulators on a local computer. In a nutshell, Qiskit translates QPs written in Python into a lower-level language called OpenQASM [27], which is its quantum instruction language. Many famous quantum algorithms such as Shor [28] and Grover [29] have already been implemented using Qiskit's API[3]. Building from our previous work [30], in detail, the main contributions of this paper are:

- A set of five novel mutation operators, leveraging the notion of syntactically equivalent gates, tailored for QPs.
- A novel Python-based toolset named QMutPy that automatically performs mutation testing for QPs written in the Qiskit's [26] full-stack library.
- An empirical evaluation of QMutPy's effectiveness and efficiency on 24 real QPs.
- A detailed discussion on extending test suites for QPs to kill more mutants and therefore detect more bugs.

To the best of our knowledge, the study described and evaluated in this paper is the first comprehensive mutation testing study on real QPs. Our results suggest that QMutPy can generate fault-revealing quantum mutants and surfaced several issues in the test suites of the real QPs used in the experiments. We have discussed two improvements to test suites, viz. increasing code coverage and improving the quality of the test assertions. Such improvements significantly increase the mutation score of the test suites—hence, leading to QPs of higher quality.

The remainder of the paper is organized as follows. We present current available open-source mutation tools and detail the extension done for QMutPy in Section II. We detail how our experiment was conducted and subjects were selected in Section III. We present our results in Section IV. We discuss and execute improvements to current test suites and how they were impacted in Section V. In Section VI, we mention published works about mutation tools and current quantum testing tools. We discuss future work and conclude our paper in Section VII.

II. MUTATION TESTING OF QUANTUM PROGRAMS

In this section, we explain our mutation strategy, including the five novel mutation operators tailored for QPs, and the implementation details of QMutPy —our proposed Python-based toolset to automatically perform mutation testing for QPs written in Qiskit's [26].

A. QUANTUM MUTATION OPERATORS

Similar to classical programs, a QP is fundamentally a circuit in which qubits are initialized and go through a series of operations that change their state. These operations are commonly known as quantum gates. Two of the most used quantum gates are the NOT gate and the Hadamard gate, usually referred to as the `x` gate and the `h` gate, respectively. They are single-qubit operations, i.e., they change the state of one qubit [31]. The `x` gate is analogous to the classical `NOT` gate; it simply inverts the current qubit state. The `h` gate is quantum specific; it puts the qubit in a perfect state of superposition (i.e., equal probability of being 1 or 0 when measured).

At the time of writing this paper, Qiskit v0.29.0 provides support to more than 50 quantum gates[4]. This includes

[2]https://qiskit.org

[3]https://github.com/Qiskit/qiskit-aqua/tree/stable/0.9/qiskit/aqua/algorithms

[4]https://qiskit.org/documentation/apidoc/circuit_library.html

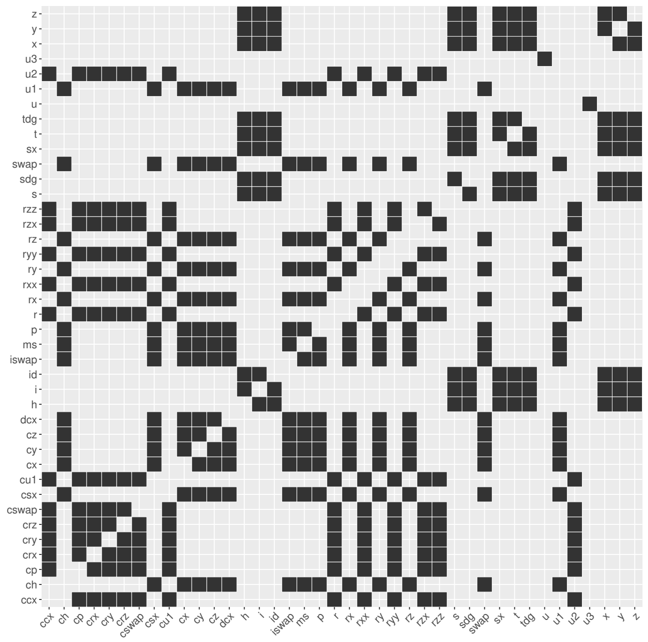


FIGURE 1. Equivalent gates in the Qiskit full-stack library reported horizontally and vertically. For example, the set of equivalent gates of gate `x` is `y`, `z`, `h`, `i`, `id`, `s`, `sdg`, `sx`, `t`, and `tdg`.

single-qubit gates (e.g., `h` gate), multiple-qubit gates (e.g., `cx` gate), and composed gates, also known as circuits (e.g., `QFT` circuit). Given their importance on the execution and result of a QP, as a simple typo on the name of the gate could cause bugs that developers may not be aware of, our set of mutation operators to generate faulty versions of QPs is based on single- and multi-qubit quantum gates, in particular, syntactically equivalent gates. We argue that our quantum mutants match real world bugs as (1) Liu et al. [32] described quantum mutation to be helpful to assess the correct behavior of QPs, and (2) 3 out of the 8 common bug patterns in Qiskit programs described by Zhao et al. [33] are related to quantum gates as so are the majority of our mutation operators. Nevertheless, and as part of our future work, we will investigate and develop novel quantum mutation operators based on conceptual mistakes a developer might make when developing QPs.

Formally, a gate g is considered syntactically equivalent to gate j if and only if the number and the type of arguments[5] required by both g and j are the same. At the time when we performed our experiment, we had identified 40 gates that had syntactical equivalents. Figure 1 lists all gates and their syntactically equivalent ones. For instance, the `h` gate has 10 syntactically equivalent gates: `i`, `id`, `s`, `sdg`, `sx`, `t`, `tdg`, `x`, `y`, and `z`. Note that these gates do not perform or compute the same operation; they are simply used in the same manner and require the same number and type of arguments.

The following subsections briefly describe the five quantum mutation operators proposed in this paper. Our examples are based on the implementation of Shor's [28] algorithm available in the Qiskit-Aqua's repository[6].

[5]Optional arguments are not taken into consideration.

1) Quantum Gate Replacement (QGR)

This mutation operator first identifies each call to a quantum gate function (e.g., `circuit.x()`[7]), and then replaces it with all syntactically equivalent gates, e.g., `circuit.h()`[8], one at a time. For instance, for the `x` quantum gate, 10 mutants are generated as there are 10 syntactically equivalent gates (see Figure 1). Listing 1 exemplifies the QGR operator.

LISTING 1. Example of the QGR operator.

```
153 -  circuit.x(qubits[0])
153 +  circuit.h(qubits[0])
```

2) Quantum Gate Deletion (QGD)

Adding and removing quantum gates from a QP can significantly impact its output. The QGD operation deletes an invocation to a quantum gate. Listing 2 exemplifies the QGD operator.

LISTING 2. Example of the QGD operator. In Python, a `pass` statement is a `nop` that when executed nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed [34].

```
153 -  circuit.x(qubits[0])
153 +  pass
```

3) Quantum Gate Insertion (QGI)

This quantum mutation operator performs the opposite action of the QGD operator. Instead of deleting a call to a quantum gate, it inserts a call to a syntactically equivalent gate. For each quantum gate in the source code, this mutation operator creates as many mutants as the number of syntactically equivalent gates. For example, for the `x` gate, which has 10 syntactically equivalent gates, it creates 11 mutants, one per equivalent gate. Note that the `x` gate itself can be inserted in the source code, counting as a valid mutant. Listing 3 shows an example of the QGI operator.

LISTING 3. Example of the QGI operator.

```
153    circuit.x(qubits[0])
154 +  circuit.y(qubits[0])
```

4) Quantum Measurement Insertion (QMI)

In quantum computing, measuring a qubit breaks the state of superposition and the qubit's value becomes either 1 or 0 (as in classical computing), which can be considered a mutation by design. This operator adds a call to the `measure` func-

[6]https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/qiskit/aqua/algorithms/factorizers/shor.py
[7]https://qiskit.org/documentation/stubs/qiskit.circuit.library.XGate.html
[8]https://qiskit.org/documentation/stubs/qiskit.circuit.library.HGate.html

TABLE 1. MutPy [35] vs. mutmut [36] vs. Cosmic Ray [37] vs. Mutatest [38]. Regarding testing frameworks mutmut supports all test runners (because mutmut only needs an exit code from the test command)

	MutPy [35]	mutmut [36]	Cosmic Ray [37]	Mutatest [38]
Open-Source	✓	✓	✓	✓
Language	Python	Python	Python	Python
Installing/Setup	▯	◧	■	▯
Mutation operators	AOD, AOR, ASR, BCR, COD, COI, CRP, DDL, EHD, EXS, IHD, IOD, IOP, LCR, LOD, LOR, ROR, SCD, SCI, SIR	value mutations, decision mutations, statement mutations	Binary Operator Replacement, Boolean Replacer, Break/-Continue, Comparison Operator Replacement, Exception Replacer, Keyword Replacer, Number Replacer, Remove Decorator, Unary Operator Replacement, Zero Iteration For Loop	AugAssign, BinOp, BinOp Bitwise Comparison, BinOp Bitwise Shift, BoolOp, Compare, Compare In, Compare Is, If, Index, NameConstant, Slice
Can select operators	✓	✗	✓	✓
Test framework	unittest, pytest	any	unittest, pytest	pytest
Report	yaml, html	xml	html	—
Fully automated	✓	✓	✓	✓

tion[9] for each quantum gate call. Listing 4 shows an example of the QMI operator.

LISTING 4. Example of the QMI operator.

```
      circuit.x(qubits[0])
    + measurement_cr =
    ClassicalRegister(circuit.num_qubits)
    + circuit.add_register(measurement_cr)
    + circuit.measure(qubits[0], measurement_cr)
```

5) Quantum Measurement Deletion (QMD)

Contrary to QMI, the QMD operator removes each measurement from a QP, one at a time. Without a `measure` call, the QP keeps the superposition state and as a consequence does not converge the qubit to either 1 or 0. Listing 5 shows an example of the QMD operator.

LISTING 5. Example of the QMD operator.

```
      up_cqreg = ClassicalRegister(2 * self._n, name='m')
      circuit.add_register(up_cqreg)
    - circuit.measure(self._up_qreg, up_cqreg)
    + pass
```

B. QMutPy TOOLSET

QPs written in Python and using Qiskit library are a mix of classical operations (e.g., initialization of variables, loops), and quantum operations (e.g., initialization of quantum circuits, measuring qubits). Thus, we foresee that the most suitable mutation tool for QPs would be one that

- Supports Python programs and the two widespread testing frameworks for Python: `unittest` and `pytest`.
- Supports various classical mutation operators (e.g., Assignment Operator Replacement, Conditional Operator Insertion).
- Supports the creation of a report that could be shown to a developer or easily parsed by an experimental infrastructure (as the one described in Section III).
- Fosters wide adoption, the learning curve to install, configure and use the tool ought to be low.

In this section, we first describe the most relevant mutation testing tools out there and which requirements they fulfil. Then we selected a tool to build upon, and describe its workflow and added features.

1) Python-based Mutation Testing Tools

Mutatest [38], mutmut [36], MutPy [35] and CosmicRay [37] are the most popular mutation testing tools for Python that are available through pip[10] (the package installer for Python). Table 1 reports the most relevant features of each mutation tool. In the following subsections, we describe their advantages and disadvantages. Albeit being open-source, fully automated, and supporting classical mutation operators, not all tools fulfil all our requirements.

Mutatest [38] only supports `pytests` whereas, e.g., the programs in the Qiskit-Aqua's repository[11] require `unittest`. It neither produces a report of a mutation testing session. Thus, any post-mortem analysis (e.g., statistical analysis) could not be easily performed.

mutmut [36] does not allow one to instantiate the tool with a single mutation operator or a defined set of mutation operators. Thus, a developer that decides to use it would have to wait for all mutants to be analyzed. This can be severely time-consuming as a program could have thousands of mutants and, more importantly, a developer would not be able to, e.g., only select quantum mutation operators. Thus, using mutmut would be unproductive.

MutPy [35] and Cosmic Ray [37] are similar in nature. Both provide a reporting system, support `unittest` and `pytest`, and allow one to select a subset of mutation operators. However, from our own experience installing and running the tools, MutPy's learning curve is more gradual than Cosmic Ray's.

The tool that better fulfils all requirements we aimed for in a mutation tool is MutPy [35] which we extended and named QMutPy (details in Section II-B3).

2) MutPy Flow

MutPy's workflow is composed by four main steps. Given a Python program P, its test suite T, and a set of mutation operators M, MutPy's workflow is as follows: (1) MutPy firstly loads P's source code and test suite; (2) Executes T on the original (unmutated) source code; (3) Applies M and generates all mutant versions of P; (4) Executes T on each

[9] https://qiskit.org/documentation/stubs/qiskit.circuit.library.Measure.html

[10] https://pypi.org/project/pip

[11] https://github.com/Qiskit/qiskit-aqua/tree/stable/0.9

mutant and provides a summary of the results either as a yaml or html report.

Since steps one and two are self-explanatory, we will focus on steps three and four. In step three, MutPy parses the code and for each mutation operator[12] checks if there are mutants to be generated. Mutants in MutPy are generated through the Python Abstract Syntax Tree (AST). When a possible mutation is found, the corresponding node from the AST is removed and a mutated node is created and injected into the unmutated source code.

In step four, MutPy executes T on the mutated version and produces a report. Each report includes information such as the number of mutants, whether each mutant was either killed, survived, incompetent (e.g., mutants that make the source code uncompilable.), or timeout, the time it took to run the T on P, the time it took to run T on each mutant.

3) QMutPy

QMutPy[13] is built on top of the open-source Python mutation testing tool MutPy. Installing and using QMutPy is simple and straightforward. One only needs to clone QMutPy's repository and follow the installation and usage instructions available in the README[14] file.

We extended MutPy by implementing the quantum mutation operators described in Section II-A which developers can freely use to perform mutation testing on their QPs written in Qiskit. Notwithstanding, addressing the technical challenges of implementing the quantum operators, we added support to MutPy to mutate AST calls[15], which is not possible in its original version. Interested readers can find more information on the technical challenges we faced to implement the quantum mutation operators described in Section II-A and how we addressed them in Fortunato et al. [39]'s recent work.

III. EMPIRICAL STUDY

We have conducted an empirical study to evaluate QMutPy's effectiveness and efficiency at performing mutation testing on QPs. In particular, in this study, we aim to answer the following research questions:

RQ1: How efficient is QMutPy at creating quantum mutants?
RQ2: How many quantum mutants are generated by QMutPy?
RQ3: How do test suites for QPs perform at killing quantum mutants?
RQ4: How many test cases are required to kill or timeout a quantum mutant?
RQ5: How are quantum mutants killed?

TABLE 2. Details of QPs used in the empirical evaluation. The test suite of each QP was identified and selected based on each program's name. In Qiskit, a QP is named after the algorithm it implements and to its test suite is given the prefix "test". For example, the test suite `test_shor.py` corresponds to the program `shor.py`. Code coverage was measured using the `Coverage.py` tool.

Algorithm	LOC	# Tests	Time (seconds)	% Coverage
adapt_vqe	151	5	85.66	82.78
bernstein_vazirani	80	33	4.28	98.75
bopes_sampler	91	2	320.51	81.32
classical_cplex	210	1	0.04	81.43
cobyla_optimizer	75	4	1.60	94.67
cplex_optimizer	60	3	0.70	81.67
deutsch_jozsa	85	64	4.18	98.82
eoh	70	2	34.71	100.00
grover	381	593	153.77	95.54
grover_optimizer	197	6	21.14	96.45
hhl	341	21	630.65	93.26
iqpe	231	3	20.38	93.51
numpy_eigen_solver	220	5	0.10	76.36
numpy_ls_solver	56	1	0.00	92.86
numpy_minimum_eigen_solver	73	5	0.24	94.52
qaoa	96	18	49.45	95.83
qgan	226	11	349.72	84.51
qpe	197	3	21.27	94.92
qsvm	303	8	266.19	78.22
shor	265	13	251.76	93.21
simon	89	48	17.21	98.88
sklearn_svm	88	4	0.13	76.14
vqc	443	13	1626.38	85.55
vqe	386	19	811.27	85.49
Average	183.92	36.88	194.64	89.78

As a baseline, we have compared the results achieved by QMutPy's quantum mutation operators with MutPy's classical mutation operators[16]. Note that works [40, 41, 42, 43] on quantum mutation are very preliminary and no other classical or quantum mutation tool could have been used in our empirical study as a baseline (see Sections II-B1 and VI for more information).

We show our commitment to open science [44] by making QMutPy and our experimental infrastructure (data and scripts) available to the research community to assist in future research. The QMutPy tool is available at https://github.com/danielfobooss/mutpy and all data and scripts are available at https://github.com/jose/qmutpy-experiments.

A. EXPERIMENTAL SUBJECTS

To conduct our empirical study, we require (1) real QPs written in the Qiskit's framework [26] (as, currently, QMutPy only supports Qiskit's quantum operations), (2) QPs written in Python[17], (3) an open-source implementation of each QP, and (4) a test suite of each QP. To the best of our knowledge, there are four primary candidate sources of QPs that fulfil (1): the Qiskit-Aqua's repository[18] itself, the "Programming Quantum Computers" book's repository[19] from O'Reilly, the "Qiskit Textbook Source Code"'s repository[20] from the

[12]MutPy supports 20 classical mutation operators and seven experimental mutation operators. If a user does not specify any mutation operator, MutPy applies all of them in alphabetical order.
[13]QMutPy is publicly available at https://github.com/danielfobooss/mutpy.
[14]https://github.com/danielfobooss/mutpy/blob/master/README.rst
[15]https://docs.python.org/3/library/ast.html#ast.Call
[16]https://github.com/mutpy/mutpy#mutation-operators
[17]Although Jupyter notebooks include Python source code, they are not supported by QMutPy.
[18]https://github.com/Qiskit/qiskit-aqua/tree/stable/0.9/qiskit/aqua/algorithms
[19]https://github.com/oreilly-qc/oreilly-qc.github.io/tree/1b9f4c1/samples
[20]https://github.com/qiskit-community/qiskit-textbook/tree/3ffedf9

Qiskit Community, and the official "Qiskit tutorials"'s repository[21].

Qiskit-Aqua's[22] repository provides the implementation of 24 QPs in Python, including the successful Shor [28], Grover [29], and HHL [45], and a fully automated test suite for each program. Hence, it fulfils all our requirements.

O'Reilly's book provides the implementation of 182 QPs, 29 written using the Qiskit's framework. However, no test suite is provided for any of the 182 programs. Hence, it does not fulfil (4). "Qiskit Textbook Source Code"'s and "Qiskit tutorials"'s repositories provide Jupyter Python notebooks with examples on how to interact with the Qiskit's framework. No test suite is available for any of the examples. Hence, it does not fulfil (2) nor (4).

Table 2 lists all QPs used in our empirical evaluation. For each program it provides the number of Lines of Codes (LOCs), the number of correspondent test cases, the time required to run the tests, and the code coverage at line level of the tests.

In the Qiskit-Aqua's repository, 24 QPs meet our criteria. We argue that including all QPs, i.e., purely classical (e.g., `classical_cplex`, `cplex_optimizer`), hybrid (e.g., `vqe`, `qaoa`), and purely quantum (e.g., `shor`, `bernstein_vazirani`) was relevant to evaluate the effectiveness of (i) our tool at generating classical and quantum mutants and (ii) tests designed for QPs at killing classical and quantum mutants.

On average, the considered QPs have 184 LOC, where the smallest program has 56 LOC (`numpy_ls_solver`) and the largest has 443 (`vqc`). The number of tests and the time required to run all tests differs significantly. The number of tests ranges from 1 test (`classical_cplex` and `numpy_ls_solver`) to 593 tests (`grover`), and the runtime ranges from nearly 0 seconds (`numpy_ls_solver`) to 1627 seconds (`vqc`).

Regarding code coverage, on average, QPs' test suites cover 90% of all LOCs. This is in accordance with best practices [46] and also with a previous study conducted by Fingerhuth et al. [25], where the ratio of code exercised by QPs' tests was slightly above the industry-expected standard.

B. EXPERIMENTAL SETUP

All experiments were executed on a machine with an AMD Opteron 6376 CPU (64 cores) and 64 GB of RAM. The operating system installed on this machine was CentOS Linux 7. We used Python version 3.7.0 in our experiments because it is the version supported by QMutPy and one of the required versions of Qiskit. We used the GNU Parallel tool [47] to run all experiments in parallel.

[21] https://github.com/Qiskit/qiskit-tutorials/tree/eb189a6

[22] Although Qiskit-Aqua's repository has been deprecated as of April 2021, all its functionalities "are not going away" and have been migrated to either new packages or to other Qiskit packages. For example, core algorithms and operators' functions have been moved to the Qiskit-Terra's repository. More info in https://github.com/Qiskit/qiskit-aqua/#migration-guide.

In our experiments, we ran QMutPy with two configurations: with classical mutation operators only and with quantum mutation operators. For both configurations, we used MutPy's default parameters.

For each QP / test suite, we collected the number of generated mutants, the number of mutated LOC, and the ratio of mutants per LOC, the number of mutants killed, the number of mutants that survived and were exercised as well as that survived and were not exercised by the test suite, the number of incompetent mutants, the number of timeout mutants, the mutation score calculated with the number of survived mutants exercised and not exercised by the test suite and finally, the time it took to run all mutants.

C. EXPERIMENTAL METRICS

To be able to compare the effectiveness of each test suite at killing mutants, we first compute its mutation score [19], i.e., ratio of killed mutants to total number of mutants (excluding incompetent mutants, e.g., mutants that introduce non-compiling changes). Formally, the mutation score of a test suite T is given by:

$$\frac{\sum\limits_{o \in O} \frac{|K_o|}{|M_o|-|I_o|}, |M_o| - |I_o| > 0}{|O|} \times 100\% \qquad (1)$$

where O represents the set of mutation operators and o a single mutation operator, $|M_o|$ the number of mutants injected by o, $|I_o|$ the number of incompetent mutants generated by o, and $|K_o|$ the number of mutants (of o) killed by T.

As some mutants might not be killed by T because the mutated code is not even executed by T, in our empirical analysis we also report a mutation score which ignores mutants that are not executed by T. This score would allow one to assess the maximum mutation score T could achieve. Formally, this score is computed as:

$$\frac{\sum\limits_{o \in O} \frac{|K_o|}{|E_o|-|I_o|}, |E_o| - |I_o| > 0}{|O|} \times 100\% \qquad (2)$$

where $|E_o|$ represents the number of mutants injected by m and exercised by T.

Regarding time, we compute and report three different runtimes: (1) total time to perform mutation analysis on test suite T which includes the time to create the mutants and run all tests on all mutants (Runtime column in Table 4), (2) time to inject a mutant in a non-mutated code (Generate mutant in Figure 3), (3) time to create a mutated module after injecting the mutant (Create mutated module in Figure 3).

We also perform the Kruskal-Wallis non-parametric test [48], with a significance level of 0.01, and Cohen's d effect-size measure to evaluate the statistical significance of the results reported in Section IV. Note that, in Section V, we performed ad-hoc experiments on specific tests, and therefore there are not enough data points to perform a statistical analysis.

D. THREATS TO VALIDITY

Based on the guidelines in [49], we discuss the threats to validity.

Threats to External Validity: The QPs used in our empirical evaluation might not be representative of the whole QPs population. Moreover, the state of test cases selected for each QP might not be complete (i.e., we may have missed other test cases in Qiskit-Aqua that test the QPs' code). Note that the lack of real-world QPs is a well-known challenge [50, 40]. Another threat is that we compared the results for only one yet popular quantum framework (Qiskit). Caution is required when generalizing to other frameworks (e.g., Cirq).

Threats to Internal Validity: The main threat to internal validity lies in the complexity of the underlying tools leveraged to build QMutPy as well as the ones supporting our experimental infrastructure. To mitigate this threat, the authors have peer-reviewed the code before making the changes final.

Threats to Construct Validity: The parameters for drawing our conclusions may not be sufficient. In particular, by default, MutPy (hence, QMutPy) runs a test case t on a mutant m for 5 times the time t takes to run on the non-mutated version. Increasing this number may lead to different results (i.e., fewer timeouts).

IV. RESULTS

Section III defines the methodology and protocol for our mutation analysis and poses a set of research questions related to QMutPy's effectiveness and efficiency. The following subsections answer these questions in detail. Figure 2 summarizes our results, detailing and classifying all of our mutation operations for each QP and mutation operator.

A. RQ1: HOW EFFICIENT IS QMutPy AT CREATING QUANTUM MUTANTS?

Figure 3 shows the distribution of time QMutPy takes to generate a mutant using classical and quantum mutation operators. On the one hand, the time taken to remove or inject new nodes into the program's AST is higher on all quantum mutation operators (except QMD) than on classical mutation operators. The latter takes up to a maximum of 2.68s (SCD) whereas the former takes up to 5.53s (QGD), 11.36s (QMI), 61.13s (QGR), and 75.04s (QGI). On the other hand, the time taken to create a mutated version, i.e., to convert the mutated AST back to Python code, is relatively small (less than 0.1s) for all classical and quantum mutation operators. According to Figure 3, there is no runtime difference between creating a mutated version with a classical mutation operator or a quantum mutation operator.

QMutPy is statistically significantly slower (p-value < $2.20e^{-16}$ for an effect-size measure of 2.03), up to 16x times more, at generating quantum mutants than at generating classical mutants.

We hypothesize the following reasons to explain its performance while developing our quantum operators:

1) **Mutation operators based on functions calls (i.e., calls to quantum gates).** Our set of quantum mutation operators, conversely to the classical ones, are based on function calls. Mutating a function is more complex than mutating, for example, a constant or a logical operator (e.g., "+") since specific grammar exists (e.g., ast.BinOp) for these types of mutations. It is worth noting that classical mutation operators that also modify function calls (e.g., SCD) are also more time consuming than operators that work at, e.g., logical operator level, as the LOD.
2) **Search for quantum gates.** Quantum mutation operators QGR, QGD, QGI, and QMI first visit all nodes of the AST and for each function call checks whether it is a call to a quantum gate. As the number of function calls in a program is typically high, we estimate that the consecutive checking is time-consuming. Possible solutions to address this problem would be to create a new type of operation in the Python AST, analogous to logical operators, but dedicated explicitly to quantum gates.
3) **Modifying or adding nodes in the AST.** Although quantum mutation operators QGR, QMD, and QGD only modify one node of the program's AST, QGI and QMI not only modify one node but also add another to the end of the AST. We estimate this to increase the runtime of these operators.

B. RQ2: HOW MANY QUANTUM MUTANTS ARE GENERATED BY QMutPy?

To answer this research question, we analyze our data at two different levels: (i) mutation operator, i.e., how many mutants are generated by each quantum mutation operator (see Table 3), and (ii) program level, i.e., how many quantum mutants are generated per program (see Table 4). For these sub research questions, we focus on the columns "# Mutants" and "# Mutated LOC" on both tables.

1) RQ2.1: How many mutants are generated by each quantum mutation operator?

As shown in Table 3 (column "# Mutants"), on average, our set of quantum mutation operators generated 140 mutants. The operator that generated fewer mutants is QMD (12 mutants), whereas QGI (328 mutants) is the one generating more mutants. These results show that

- Quantum measurements are not that common in QPs (as only 12 measurements were mutated).
- Out of the 40 quantum gates with at least one syntactical equivalent gate, 28 appear in the evaluated QPs.
- The insertion and replacement of quantum gates with their syntactical equivalent ones represent 90% of all quantum mutants. This shows the importance of syntactically equivalent gates, tailored for QPs, in mutation testing.

It is worth noting that the average number of mutants generated by our quantum mutation operators is slightly below the number of mutants generated by classical mutation operators (140 vs. 186, which CRP highly dominates). As there are many more LOCs that could be targeted by classical mutation operators (e.g., usage of constants) and many more classical operators (18 vs. our set of 5 quantum ones), it is expected

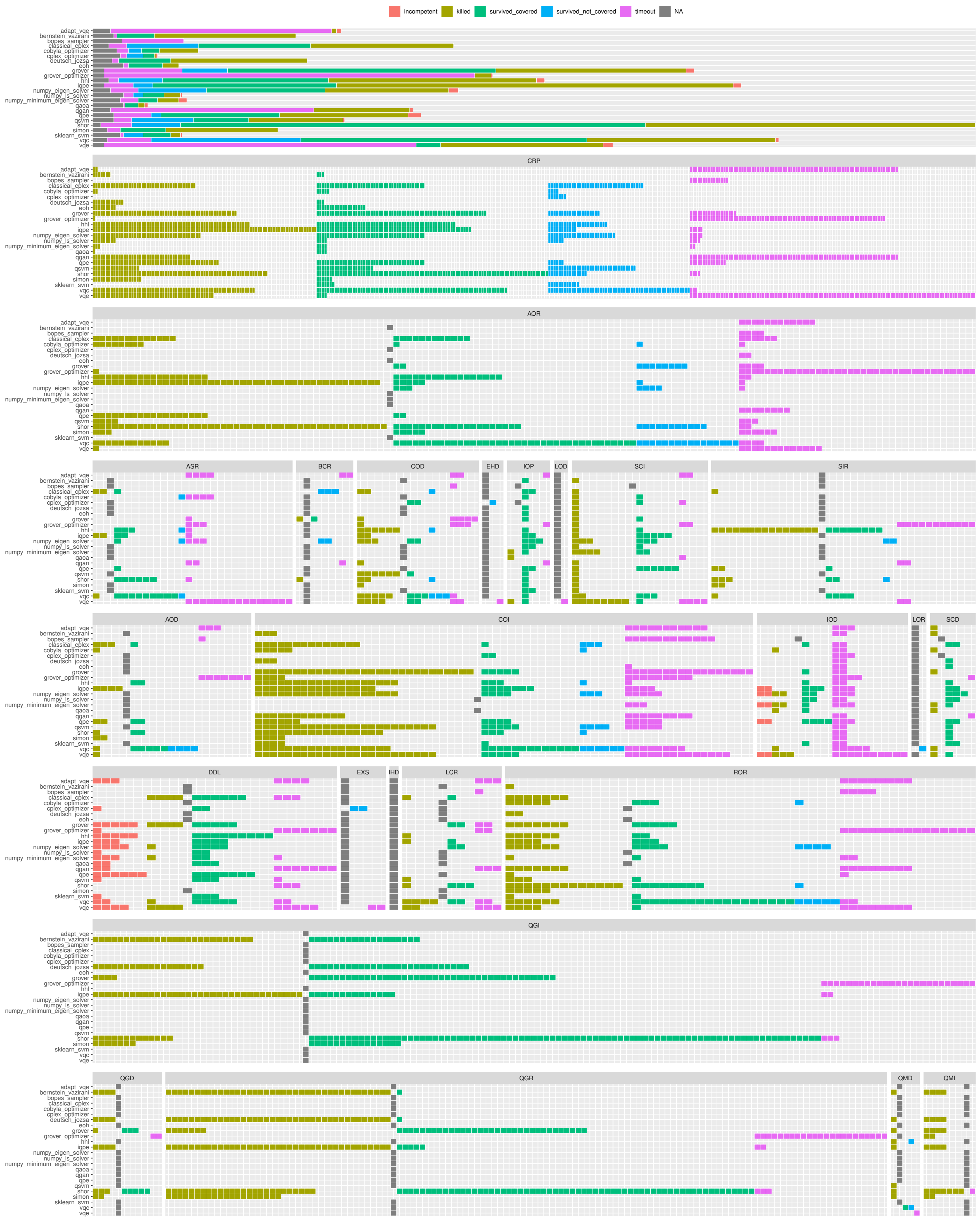


FIGURE 2. Detailed analysis and classification of all mutation operations performed in our study per algorithm and mutation operator.

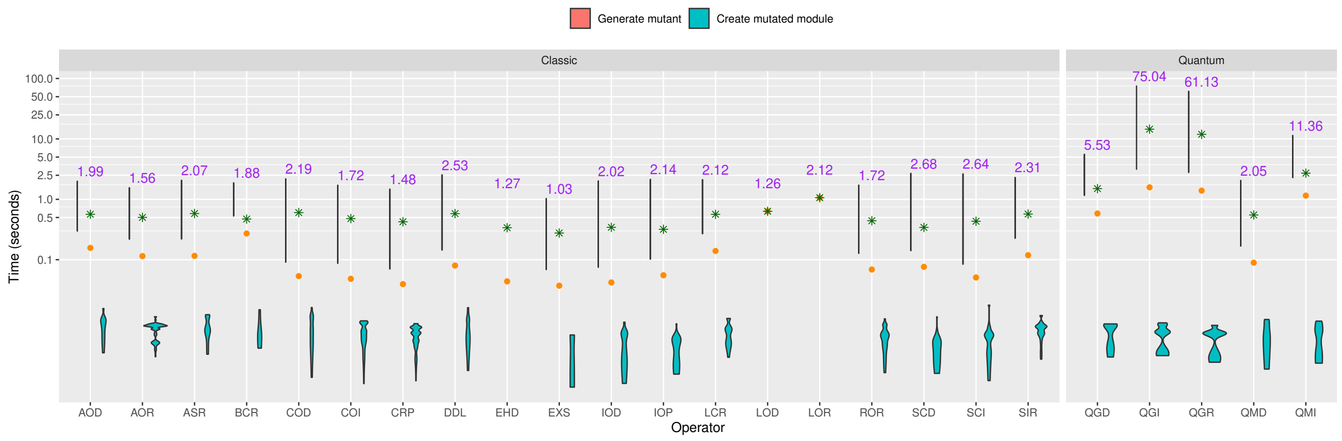


FIGURE 3. Distribution of the time required to inject a mutant and create a mutated target version. For each mutation operator, the purple text reports the maximum time of the 'Generate mutant' phase, i.e., time to inject or remove nodes from the AST, the green star reports the average time a mutation operator takes to create a mutated module (i.e., Python code), and the orange circle reports the median time a mutation operator takes to generate a mutant and create a mutated module.

TABLE 3. Results per mutation operator. (Refer to Table 4 for an explanation of each column.)

Operator	# Mutants	# Killed	# Survived	# Incompetent	# Timeout
			Classic mutants		
AOD	42	15	12 / 4	0	11
AOR	421	169	105 / 41	0	106
ASR	67	5	23 / 4	0	35
BCR	11	2	1 / 5	0	3
COD	63	34	10 / 6	0	13
COI	397	221	53 / 19	0	104
CRP	1860	634	551 / 256	0	419
DDL	147	15	55 / 0	44	33
EHD	2	0	0 / 1	0	1
EXS	4	0	0 / 2	0	2
IOD	100	10	17 / 0	10	63
IOP	31	3	25 / 0	0	3
LCR	38	11	11 / 0	0	16
LOD	1	0	0 / 0	0	1
LOR	1	0	0 / 1	0	0
ROR	185	79	47 / 11	0	48
SCD	31	8	21 / 0	0	2
SCI	69	34	25 / 0	0	10
SIR	57	24	15 / 3	0	15
Average	185.63	66.53	51.11 / 18.58	2.84	46.58
			Quantum mutants		
QGD	28	18	8 / 0	0	2
QGI	328	102	196 / 0	0	30
QGR	300	170	102 / 0	0	28
QMD	12	8	1 / 2	0	1
QMI	28	27	0 / 0	0	1
Average	139.20	65.00	61.40 / 0.40	0.00	12.40

that there are more classical mutants than quantum mutants. Nevertheless, the top-2 quantum mutation operators (i.e., QGI and QGR) generated more mutants than 15 out of the 18 classical mutation operators.

On average, for 11 out of 24 QPs, QMutPy mutates 4 LOCs and generates 14 different quantum mutants per mutated line. It generates a total of 696 quantum mutants, 140 per mutation operator. Overall, the number of quantum mutants generated by QMutPy is not statistically significantly lower (p-value = $5.98e^{-06}$ for an effect-size measure of 0.17) than the number of classical mutants.

2) RQ2.2: How many quantum mutants are generated on each program?

As we can see in Table 4 (column "# Mutants"), QMutPy generates at least one quantum mutant for 11 out of the 24 QPs. This means that the remaining programs neither use quantum gates nor measurements. It is worth noting that the quantum technique used impacts the number of generated quantum mutants, e.g., fewer (or no) mutants were generated for hybrid algorithms (e.g., `vqe`, `qaoa`) compared to purely quantum algorithms (e.g., `classical_cplex`, `cplex_optimizer`). Thus, more quantum mutation operators should be investigated and developed to support those QPs.

On average, QMutPy generated 64 quantum mutants (e.g., 1 mutant for `vqe` and `qsvm`, 207 mutants for `shor`). Given that our set of mutation operators targets function calls which might not occur as often as, e.g., classical arithmetic operations in a program, on average, QMutPy only mutated 4 LOCs with an average of 13 mutants per line (see column "# Mutated LOC"). In contrast, at least one classical mutant was generated for all programs. 147 mutants on average (+83) and 64 LOCs mutated (+60) with an average of 3 mutants per line (-10). Note that QPs are composed of more traditional programming blocks such as conditions, loops, and arithmetic operations than calls to the quantum API. Thus, and as there are many more LOCs that can be mutated using classical mutation operators than using quantum mutation operators, it is expected to have fewer quantum mutants in a QP.

C. RQ3: HOW DO TEST SUITES FOR QPs PERFORM AT KILLING QUANTUM MUTANTS?

The question aims to analyze the quality and resilience of test suites designed to verify QPs. As mentioned before, the idiosyncrasies underlying QPs (e.g., superposition, entangle-

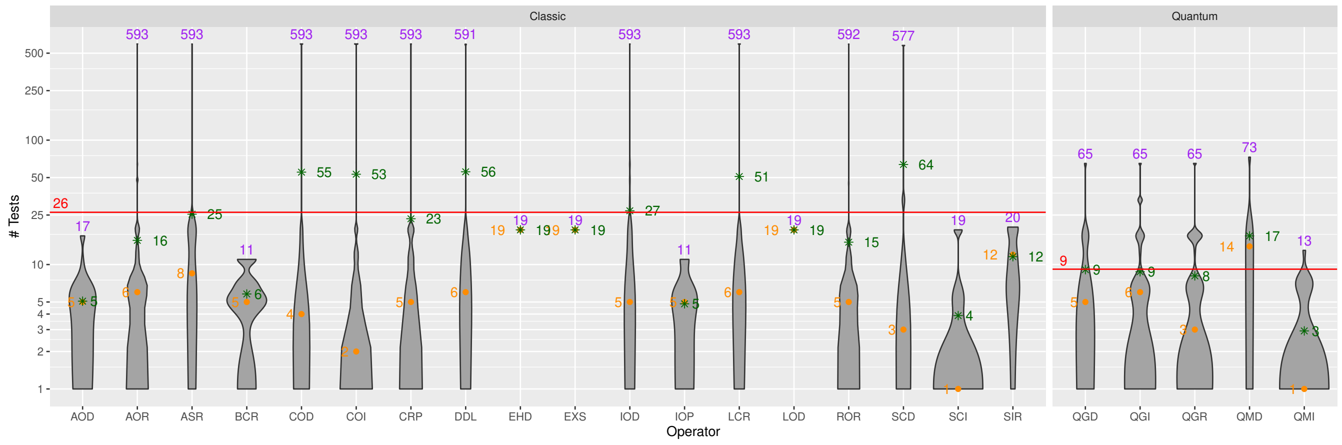
TABLE 4. Summary of our results per QP. Note that although 24 QPs were considered in our study, here we only list the ones for which QMutPy was able to generate at least one mutant (either classical or quantum). Column "Quantum Program" lists the subjects used in our experiments. Column "# Mutants" reports the number of mutants per subject. Column "# Mutated LOC" reports the number of LOCs with at least one mutant and the ratio of mutants per line of code. Column "# Killed" reports the number of mutants killed by the subject's test suite. Column "# Survived" reports the number of mutants that survived and were exercised by the test suite, and the number of mutants that survived and *were not exercised* by the test suite. Note that any buggy code or mutant that is not exercised by the test suite cannot be detected or killed. Column "# Incompetent" reports the number of mutants considered incompetent, e.g., mutants that make the source code uncompilable. Column "# Timeout" reports the number of mutants for which the subject's test suite ran out of time. Column "% Score" reports the mutation score considering all mutants killed and survived (but excluding incompetents), and reports the mutation score considering all mutants killed by the test suite and all mutants that survived and were exercised by the test suite. Column "Runtime" reports the time, in minutes, QMutPy took to run all mutants and each mutant on average.

Quantum Program	# Mutants	# Mutated LOC	# Killed	# Survived	# Incompetent	# Timeout	% Score	Runtime
Classic mutants								
adapt_vqe	142	64 (2.22)	3	0 / 0	3	136	7.31 / 7.31	1023.66 (6.33)
bernstein_vazirani	19	10 (1.90)	13	4 / 0	0	2	67.14 / 67.14	3.51 (0.06)
bopes_sampler	38	22 (1.73)	0	0 / 0	0	38	0.00 / 0.00	1119.35 (26.44)
classical_cplex	212	82 (2.59)	88	69 / 44	0	11	49.50 / 53.77	4.54 (0.01)
cobyla_optimizer	50	25 (2.00)	24	11 / 8	0	7	51.31 / 55.44	4.35 (0.04)
cplex_optimizer	23	14 (1.64)	1	7 / 10	1	4	4.17 / 4.17	1.96 (0.02)
deutsch_jozsa	27	11 (2.45)	18	5 / 0	0	4	47.50 / 47.50	4.21 (0.10)
eoh	34	14 (2.43)	10	21 / 0	0	3	22.02 / 22.02	36.61 (1.28)
grover	270	137 (1.97)	100	89 / 28	5	48	31.90 / 32.28	1031.75 (5.12)
grover_optimizer	187	73 (2.56)	8	0 / 0	1	178	6.65 / 6.65	329.12 (1.62)
hhl	266	121 (2.20)	127	102 / 26	5	6	39.14 / 41.04	1998.06 (9.22)
iqpe	287	93 (3.09)	162	94 / 12	5	14	43.31 / 43.73	81.05 (0.27)
numpy_eigen_solver	214	94 (2.28)	76	73 / 42	6	17	21.37 / 23.83	5.90 (0.01)
numpy_ls_solver	36	14 (2.57)	10	13 / 6	1	6	14.86 / 17.16	1.60 (0.01)
numpy_minimum_eigen_solver	41	19 (2.16)	13	12 / 0	5	11	35.42 / 35.42	2.28 (0.03)
qaoa	15	9 (1.67)	4	8 / 0	2	1	45.00 / 45.00	29.94 (0.95)
qgan	186	80 (2.33)	59	0 / 0	2	125	23.98 / 23.98	3779.19 (21.59)
qpe	189	68 (2.78)	79	73 / 6	8	23	29.59 / 29.80	82.51 (0.39)
qsvm	141	88 (1.60)	57	34 / 38	1	11	45.94 / 48.50	674.82 (5.19)
shor	331	123 (2.69)	153	136 / 30	0	12	40.78 / 44.99	1011.41 (4.23)
simon	58	21 (2.76)	37	13 / 0	0	8	63.40 / 63.40	23.94 (0.24)
sklearn_svm	38	20 (1.90)	6	17 / 12	1	2	28.75 / 28.75	1.25 (0.01)
vqc	411	181 (2.27)	116	175 / 91	2	27	27.25 / 30.52	8630.39 (26.07)
vqe	312	136 (2.29)	100	15 / 0	6	191	31.87 / 31.87	13419.82 (38.01)
Average	146.96	63.29 (2.25)	52.67	40.46 / 14.71	2.25	36.88	32.42 / 33.51	1387.55 (6.14)
Quantum mutants								
bernstein_vazirani	93	5 (18.60)	74	19 / 0	0	0	91.32 / 91.32	7.29 (0.01)
deutsch_jozsa	93	5 (18.60)	66	27 / 0	0	0	87.68 / 87.68	7.70 (0.01)
grover	93	5 (18.60)	17	76 / 0	0	0	50.32 / 50.32	212.24 (1.31)
grover_optimizer	52	2 (26.00)	2	0 / 0	0	50	25.00 / 25.00	118.56 (1.41)
hhl	2	2 (1.00)	1	0 / 1	0	0	50.00 / 100.00	97.70 (8.62)
iqpe	105	5 (21.00)	82	19 / 0	0	4	90.56 / 90.56	31.07 (0.14)
qsvm	1	1 (1.00)	1	0 / 0	0	0	100.00 / 100.00	47.85 (0.03)
shor	207	9 (23.00)	50	150 / 0	0	7	53.34 / 53.34	779.68 (2.70)
simon	47	3 (15.67)	32	15 / 0	0	0	86.36 / 86.36	13.45 (0.10)
vqc	2	2 (1.00)	0	1 / 1	0	0	0.00 / 0.00	170.21 (12.24)
vqe	1	1 (1.00)	0	0 / 0	0	1	0.00 / 0.00	144.21 (61.95)
Average	63.27	3.64 (13.22)	29.55	27.91 / 0.18	0.00	5.64	57.69 / 62.23	148.18 (8.05)

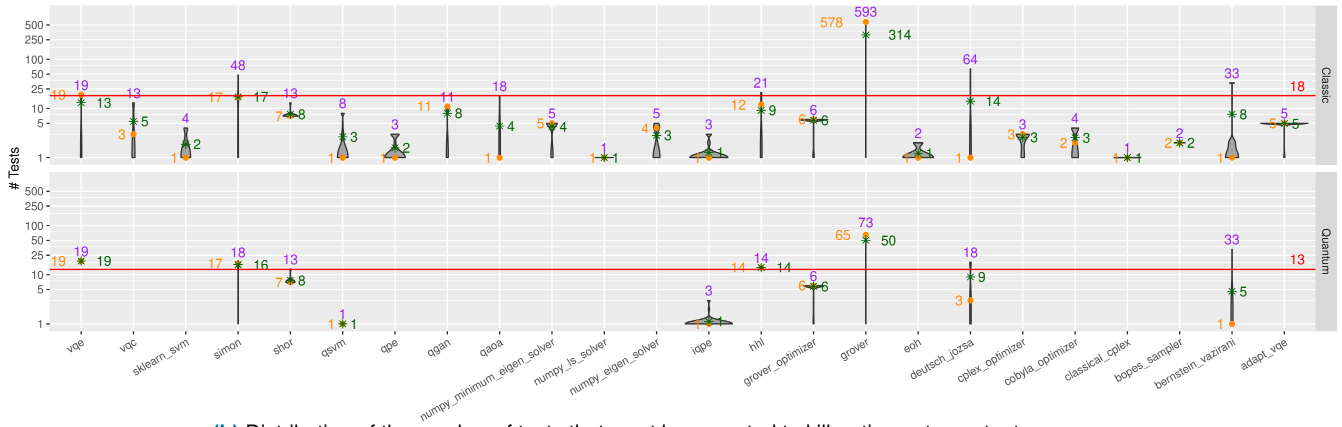
ment) make testing far from trivial. We argue that QMutPy's mutants can be used as benchmarks to assess the quality of tests designed to verify QPs. Table 4 reports the results of performing mutation testing on the 24 QPs described in Table 2, whereas Table 3 summarizes the results per mutation operator.

As we can see in Table 3, out of the 696 mutants generated by our quantum mutation operators, 325 (46.70%) were killed by the programs' test suites. QGI, the mutation operator that generated more mutants, killed 102 mutants out of 328, followed by QGR with 170 killed mutants out of 300 generated. The non-killed mutants either survived to the test suites (307, 44.11%), were not even exercised by the test suites (2 QMD mutants, 0.29%), or resulted in a timeout (62, 8.91%). In comparison, out of the 3527 generated by classical mutation operators, 1264 (35.84%) were killed, 971 (27.53%) survived, 353 (10.01%) were not exercised by the test suites, and 885 (25.10%) timeout. Note that +10.86% more quantum mutants are killed than classical ones and that only 0.29% of all quantum mutants are not exercised by the test suites, as opposed to 10.01% (+9.72%) of all classical mutants. These results show that the programs' test suites might have been designed to mainly verify the quantum aspect of each program.

To verify whether quantum mutants are not killed by chance and that instead the tests were tailored to verify quantum behavior, we conducted a small experiment on two QPs, i.e., `shor` and `grover`. We first removed all assertions from `shor`'s and `grover`'s test suites, then re-ran our mutation analysis on each QP, and finally re-computed mutation scores. The mutation scores achieved in this experiment dropped from 53.34% to 24.22% (`shor`) and from 50.32% to 20.00% (`grover`). This further shows that the intention of testing specific quantum behavior is the main reason tests kill quantum mutants.



(a) Distribution of the number of tests that must be executed to kill or timeout a mutant per mutation operator.



(b) Distribution of the number of tests that must be executed to kill or timeout a mutant per program.

FIGURE 4. Distribution of the number of tests that must be executed to kill or timeout each mutant. The purple text reports the maximum number of tests needed to kill a mutant, the green star reports the median of the number of tests needed to kill a mutant, and the orange circle reports the average number of tests needed to kill a mutant. The red line represents the overall average number of tests needed to kill a classical mutant or a quantum mutant.

At program level, on average, the mutation score achieved by all programs' test suites was 57.69% if all mutants are considered (Equation (1)) and 62.23% if only mutants covered by the test suite are considered (Equation (2)). Recall that noncovered mutants would never be killed by any test as the mutated code is never executed. The mutation score achieved by each test suite ranged from 0% (`vqc` and `vqe`, more on this in Section V-A) to 100% (`hhl` and `qsvm`). The mutation score achieved by all programs' test suites on classical mutants was 33.51% on average (considering all programs) and 41.61% if we only consider the same set of 11 programs for which quantum mutation operators were able to generate at least one mutant. The programs' test suites achieved a higher mutation score on quantum mutants than on classical mutants, +20.62% (62.23% vs. 41.61%). Hence, reinforcing the idea that the test suites have been designed to mainly verify the quantum characteristics of each QP.

Regarding the time required to run mutation testing, on average, test suites took 148.18 minutes to run on quantum mutants. Note that although different programs have more / less mutants or test cases, the runtime of each QP's test suite on quantum mutants differs largely. For instance, `shor`'s test suite, the QP with more quantum mutants, took 779.68 minutes; `qsvm`, the QP with fewer mutants and tests, took 47.85 minutes; and `grover`, the QP with more tests, took 212.24 minutes. In comparison to classical mutants, programs' test suites took longer to run on quantum mutants than on classical. For example, `qsvm`'s test suite took 47.85 minutes to run on the only generated quantum mutant and 4.79 minutes on average ($\frac{674.82 \text{ minutes}}{141 \text{ classical mutants}}$) on each classical mutant. The reasons behind these time differences are explained in Section IV-A.

Test suites for QPs achieved a mutation score statistically significantly higher than the mutation score achieved on classical mutants (62.23% vs. 33.51%), p-value = $2.00e^{-05}$ for an effect-size measure of 0.92.

D. RQ4: HOW MANY TEST CASES ARE REQUIRED TO KILL OR TIMEOUT A QUANTUM MUTANT?

The questions aims to understand the effectiveness of current quantum test suites. Figure 4 shows the distribution of the

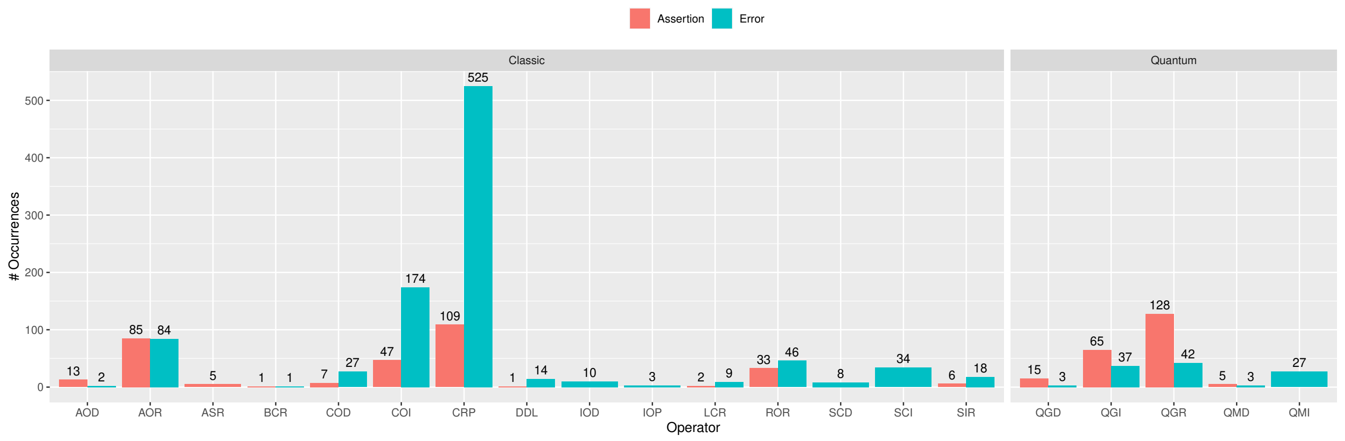


FIGURE 5. Number of mutants killed by an *assertions* or an *error* per mutation operator. In our experiments we found three types of *errors* thrown by the test suites. (1) Qiskit-related: `AquaError`, `QiskitOptimizationError`, `QiskitError`, and `CircuitError`. (2) Python: `NotImplementedError`, `IndexError`, `ValueError`, `AttributeError`, `IsADirectoryError`, `ZeroDivisionError`, `OverflowError`, `UnboundLocalError`, `RuntimeError`, `NameError`, and `KeyError`. (3) Third-party: `CplexSolverError`, `DQCPError`, `AxisError`, and `LinAlgError`.

number of tests required to kill or timeout each mutant per mutation operator and per QP.

At the mutation operator level, the average number of tests needed to kill or timeout each quantum mutant is 9 (e.g., 1 test for QMI and 73 tests for QMD). The average number of tests needed to kill or timeout each classical mutant is 26, with 10 out of 18 classical mutation operators executing more than 500 tests.

At program level, the average number of tests needed to kill or timeout a quantum mutant is 13 (e.g., 1 test for `bernstein_vazirani`, `iqpe`, and `qsvm`, and 73 for `grover`). Regarding classical mutants, the average number of tests needed to kill or timeout each classical mutant was 18 (considering all programs) or 64 if only the 10 programs for which at least one quantum mutant was generated and killed or timeout are considered.

Although on average quantum mutants require -65.38% tests to be killed or timeout than classical mutants (9 vs. 26), there is no statistically significant difference (p-value = 0.52 for an effect-size measure of -0.10) between the number of tests required to either kill or timeout a classical mutant and a quantum mutant.

E. RQ5: HOW ARE QUANTUM MUTANTS KILLED?

With this question, we aim to analyze what kills quantum mutants. Figure 6 depicts the overall number of mutants killed by an assertion or an error. Figure 5 shows us the same but by mutation operation.

Out of the 1589 killed mutants, we observed that two-thirds of mutants were killed by errors (1067) and the other one-third by test assertions (522). Figure 5 reports the number of mutants killed by errors and test assertions per mutation operator. Overall, the majority of classical mutants are killed by errors. As already mentioned, we argue that

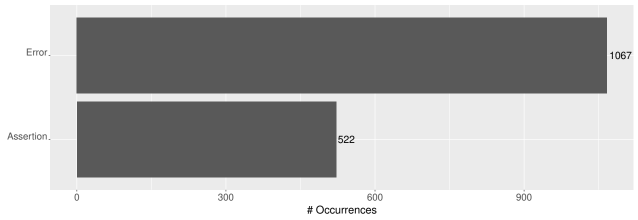


FIGURE 6. Overall number of mutants killed by an *assertions* or an *error*, e.g., an exception. In our experiments we found three types of *errors* thrown by the test suites. (1) Qiskit -related: `AquaError`, `QiskitOptimizationError`, `QiskitError`, and `CircuitError`. (2) Python: `NotImplementedError`, `IndexError`, `ValueError`, `AttributeError`, `IsADirectoryError`, `ZeroDivisionError`, `OverflowError`, `UnboundLocalError`, `RuntimeError`, `NameError`, and `KeyError`. (3) Third-party: `CplexSolverError`, `DQCPError`, `AxisError` and `LinAlgError`.

Qiskit's test suites are mainly designed to check for the correct behavior of QPs. Therefore, they are less resilient to classical mutations and likely to be killed by errors instead of test assertions. This observation does not hold for quantum mutants.

QGD, QGR, QGI, and QMD mutants are killed more often by test assertions than by errors. We also observed that QMI mutants, as expected, are killed by errors only. The reason is that Qiskit does not have a fail-safe mechanism for inserting measurements. When a measurement operation is inserted in a random position, the circuit may become unprocessable and an error is thrown. Developing better approaches to reduce the number of design errors of QMI mutants remains, however, as future work.

On the one hand, classical mutants are mainly killed by errors. Quantum mutants, on the other hand, are statistically more likely (p-value = 0.01 for an effect-size measure of 0.80) and mainly killed by test assertions (with the exception of QMI mutants).

V. IMPROVING QUANTUM TEST SUITES

The results in Section IV suggest that there is room for improvement in Qiskit's test suites. For example, we observed that 150 out of the 207 quantum mutants generated for `shor` survived.

We draw on two hypotheses to guide our discussion on how to improve QPs' test suites to kill more quantum mutants:

h_1 The low mutation score achieved by each test suite is due to their low coverage.

h_2 The low mutation score achieved by each test suite is due to their low number of test assertions.

Note that the described mutations and improvements to the test suites are available at https://github.com/jose/qmutpy-experiments.

A. IMPROVING COVERAGE

Figure 7 shows the relation between coverage and mutation score overall, of each test suite and for each mutation operation. We computed the Spearman-rank correlation coefficient between coverage and mutation score of each test suite, and observed that mutation score and coverage are correlated (+0.28, i.e., mutation score increases with coverage, p-value $1.02e^{-06}$). Thus, with this first hypothesis, we aim to investigate whether increasing the coverage of QPs, e.g., covering mutated LOCs that are not exercised by the program's test suite, leads to a higher mutation score.

Table 4 shows that there are two QPs (`hhl` and `vqc`) that have one mutant, generated by the QMD operator, that survived the test suites and are not covered by any test. The mutants are generated by the QMD operator and are in uncovered methods: `construct_circuit` (see Listing 6) and `get_optimal_vector` (see Listing 8), respectively. We extended `hhl`'s and `vqc`'s test suite[23][24], as shown in Listings 7 and 9 respectively, to cover these methods and added a more specific test assertion to each test. The test assertions verify that the number of combinations of qubits measurements is correct, which it would not be if no measurement was performed. We verified that our hypothesis holds by rerunning the mutation analysis using the augmented test suites. In both QPs, the mutants that survived our initial mutation analysis are killed by the augmented test suites. That is, `hhl`'s mutation score increased from 50% to 100% (coverage increased from 86.55% to 89.16%), and `vqc`'s mutation score from 0% to 50% (coverage increased from 93.26% to 94.43%).

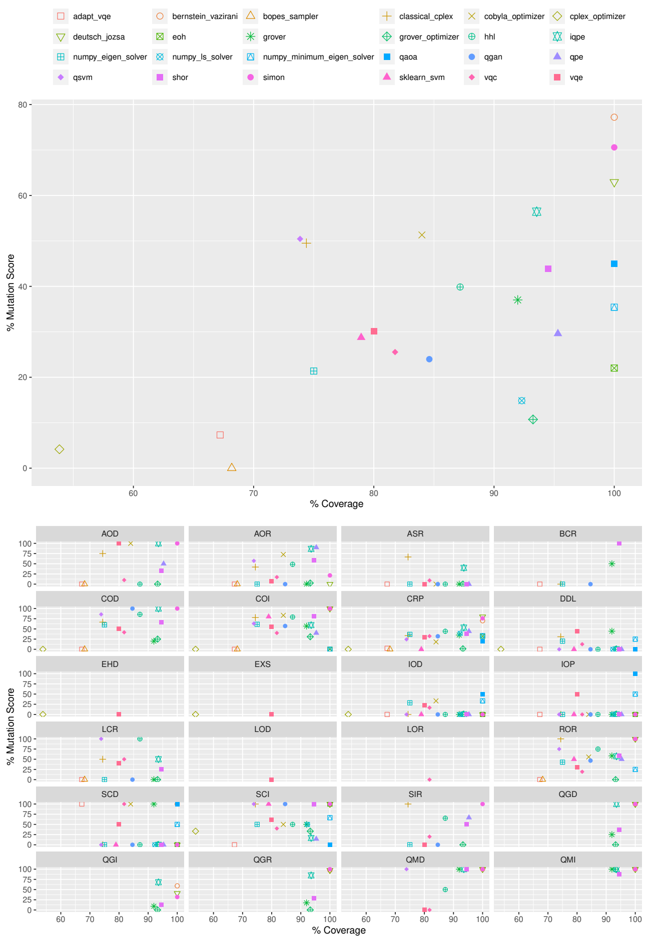


FIGURE 7. % Mutation score vs. % Coverage.

LISTING 6. Mutant not exercised by hhl's original test suite and therefore not killed.

```
def construct_circuit(self, measurement: bool =
False) -> QuantumCircuit:
    ...
    if measurement:
        c = ClassicalRegister(1)
        qc.add_register(c)
-       qc.measure(s, c)
+       pass
        self._success_bit = c
```

LISTING 7. Augmented hhl's test suite.

```
@data([0, 1], [1, 0], [1, 0.1], [1, 1], [1, 10])
def test_hhl_diagonal(self, vector):
    ...
    self.log.debug('fidelity HHL to algebraic: %s',
fidelity)
    self.log.debug('probability of result: %s',
hhl_result.probability_result)
+   qc = algo.construct_circuit(True)
+   result = execute(qc, backend =
BasicAer.get_backend('qasm_simulator'), shots =
1000).result()
+   counts = result.get_counts()
+   self.assertTrue(len(counts) == 2)
```

LISTING 8. Mutant not exercised by vqc's original test suite and therefore not killed.

```
def get_optimal_vector(self):
    ...
```

[23]https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/test/aqua/test_hhl.py
[24]https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/test/aqua/test_vqc.py

```
    else:
       c = ClassicalRegister(qc.width(), name='c')
       q = find_regs_by_name(qc, 'q')
       qc.add_register(c)
       qc.barrier(q)
-      qc.measure(q, c)
+      pass
       ret = self._quantum_instance.execute(qc)
       self._ret['min_vector'] = ret.get_counts(qc)
```

LISTING 9. Augmented vqc's test suite.

```
 def test_minibatching_gradient_free(self):
     ...
     self.log.debug(result['testing_accuracy'])
self.assertAlmostEqual(result['testing_accuracy'], 0.3333333333333333)
+    vector = vqc.get_optimal_vector()
+    self.assertTrue(len(vector) == 4)
```

B. IMPROVING TEST ASSERTIONS

As mentioned before, QPs are probabilistic in nature. Suppose a quantum circuit with 2 qubits. When read, these qubits could either be 00, 01, 10, or 11. Suppose that the correct behavior is to observe 00 with 25% probability and 11 with 75%. If instead, we observe a survived mutant that measured 00, 01, 10, and 11 with some probability, then we would have a false negative since the mutant should have been killed.

We argue that asserting the number of measurements in the test suites is necessary to avoid these false negatives—hence, improving the mutation score. To verify this intuition, we augmented `shor`'s test suite[25] (the QP with the most generated quantum mutants, see Table 4) with additional test assertions, as shown in Listing 10. The added assertions check the correctness of the number of obtained measurement values.

LISTING 10. Augmented `test_shor` with four additional assertions.

```
 def test_shor_factoring(self, n_v, backend, factors):
     ...
     result_dict = shor.run(QuantumInstance(BasicAer.get_backend(backend), shots=1000))
     self.assertListEqual(result_dict['factors'][0], factors)
     self.assertTrue(result_dict["total_counts"] >= result_dict["successful_counts"])
+    self.assertTrue(result_dict["total_counts"] >= 55)
+    self.assertTrue(result_dict["total_counts"] <= 75)
+    self.assertTrue(result_dict["successful_counts"] >= 10)
+    self.assertTrue(result_dict["successful_counts"] <= 25)
```

Similar to h_1, we re-run the mutation analysis using the augmented test suites to verify that h_2 holds. Mutation score achieved by `shor`'s original test suite was 53.34% (50 mutants killed and 150 survived out of 207). The augmented test suite achieved a mutation score of 72.81% (109 mutants killed and 91 survived). In detail, the augmented test suite killed 6 out of 8 QGD mutants (+3 than the original test suite), 32 out of 99 QGI mutants (+19), 63 out of 91 QGR mutants (+37), and the same QMD and QMI mutants (1 out of 1 and 7 out of 8, respectively) as the original test suite.

[25]https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/test/aqua/test_shor.py

VI. RELATED WORK

To the best of our knowledge, there are four works in the literature that have performed quantum mutation on QPs [40, 41, 42, 43]. However, these are preliminary attempts to conduct quantum mutation testing, empirically evaluating these prior works on the same set of quantum programs and tests and comparing those tools' performance with QMutPy is impossible due to several limitations.

Ali et al. [40] performed mutation analysis on automatically generated tests for QPs to assess their effectiveness at finding seeded faults. Their study introduces four mutation operators: QGD, QGI and QGR (with no concept of syntactically equivalent gates), and a classical operator named 'replace mathematical operator'. Such studies could further benefit from a fully automated tool such as QMutPy which supports a more extensive set of mutation operators, including 20 classical operators.

Mendiluze et al. [41] proposed Muskit, a Python mutation tool for Qiskit QPs. Muskit supports the mutation of 19 Qiskit's gates, the mutation operator QGD as defined in Section II-A2, and the mutation operators QGI and QGR but with no concept of syntactically equivalent gates. QMutPy, on the other hand, supports two additional mutation operators, i.e., QMD and QMI, which can mutate `measurement` calls, and is able to mutate 40 gates (+21 than Muskit). To use Muskit, one must provide the specification of the QP so that Muskit is able to assess whether a mutant has been killed by a test. This requires expertise in quantum computing and/or on Qiskit. As the manually-written tests used in our study are equipped with test assertions, QMutPy does not require any program specification to assess whether a test kills a mutant. Mendiluze et al. [41] also conducted an experimental evaluation of Muskit on four QPs, one shared with our study, the Bernstein-Vazirani cryptography algorithm. They reported that Muskit generated 343 mutants for that algorithm (255 generated by the QGI operator, 9 QGD, and 79 QGR) and achieved a mutation score of 77.35%. In our study, QMutPy only generated 88 mutants (44 generated by the QGI operator, 4 QGD, and 40 QGR) but achieved a mutation score of 91.32%. These differences can be explained by the following:

1) Mendiluze et al. [41]'s implementation of the Bernstein-Vazirani algorithm is 14 lines long and contains 9 gates[26], and the implementation available on Qiskit-Aqua's repository (and used in our study) is 80 lines long and contains 4 gates[27] only.

[26]https://github.com/Simula-COMPLEX/muskit/blob/c148ad7/Experimental%20Data/LittleBV/LittleBV_program.py

[27]https://github.com/Qiskit/qiskit-aqua/blob/stable/0.9/qiskit/aqua/algorithms/education/bernstein_vazirani.py

2) We performed mutation analysis with the 33 manually-written tests as opposed to the 64 automatically generated tests used by Mendiluze et al. [41]. As the manually-written tests achieved a higher mutation score, further research on the automatic generation of tests for QPs should be conducted (e.g., [40]).
3) As put forward by Mendiluze et al. [41], the large number of mutants generated by the mutation operators QGI and QGR that survived might be equivalent/irrelevant mutants. QMutPy only injects or replaces syntactically equivalent gates, thus keeping the number of equivalent mutants, if any, low.

Wang et al. [42] proposed an approach named QDiff, to perform differential testing [51] on quantum software (e.g., Qiskit). QDiff generates semantically equivalent versions by applying equivalent gate transformations and mutations (QGI/QGD/QGR of random gates, gate swap, and qubit change). To identify whether a sequence of gates is semantically equivalent to another, both are executed and their measurements compared. This is time-consuming for programs with a large number of gates. QMutPy, on the other hand, generates syntactically equivalent versions which do not require the execution of any other program version to assess equivalency.

Finally, MTQC [43] is a Java quantum mutation tool that supports Qiskit and Q# QPs. MTQC supports the mutation of 17 Qiskit's gates (vs. 40 in QMutPy) and a subset of operations performed by our QGR operator (52 vs. 225, see dark squares in Figure 1). At the time of writing this paper, no study has been conducted with MTQC. We could not include MTQC in our study as (1) it does not support `unittest`, a requirement to run Qiskit-Aqua's manually-written tests, and (2) it requires one to manually use its GUI to perform the mutation analysis, one program at a time, which is time-consuming and prone to mistakes.

VII. CONCLUSION

In this paper, we propose a mutation-based technique to test QPs, coined QMutPy, that is capable of mutating QPs for Qiskit, the IBM quantum framework. This is a first attempt to perform mutation testing on QPs with a tool that is easy to use and works at scale. Furthermore, QMutPy offers classical and more quantum mutation operators than the approaches / tools proposed in the literature.

To demonstrate the effectiveness of QMutPy, we carried out an empirical study with 24 real QPs (selected from Qiskit). We observed several issues that may lead to future failures—non-optimal code coverage; low mutation scores; minimal number of test cases. Furthermore, we observed that quantum mutants required fewer test cases to be killed than classical mutants. This is likely due to the objective of the designed test suites—checking for the QP's behavior. As a consequence of our observations, we draw on two potential ways to improve test suites: coverage and assertion improvements. We show how both improvements can increase the mutation score significantly on the QPs considered in our study[28].

As for future work, we plan to extend QMutPy with other mutation operators, offer it to other quantum frameworks (e.g., Cirq and Q#), and run our mutation analysis on real quantum computers.

REFERENCES

[1] Andrew Steane. "Quantum computing". In: Reports on Progress in Physics 61.2 (1998), p. 117.

[2] Noson S Yanofsky and Mirco A Mannucci. Quantum computing for computer scientists. Cambridge University Press, 2008.

[3] John Preskill. "Quantum Computing in the NISQ era and beyond". In: Quantum 2 (Aug. 2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. URL: https://doi.org/10.22331/q-2018-08-06-79.

[4] Jianjun Zhao. Quantum Software Engineering: Landscapes and Horizons. 2020. arXiv: 2007.07047 [cs.SE].

[5] Paul Ammann and Jeff Offutt. Introduction to Software Testing. 1st ed. USA: Cambridge University Press, 2016. ISBN: 0521880386.

[6] Gordon Fraser and José Miguel Rojas. "Software Testing". In: Handbook of Software Engineering. Ed. by Sungdeok Cha, Richard N. Taylor, and Kyochul Kang. Cham: Springer International Publishing, 2019, pp. 123–192. ISBN: 978-3-030-00262-6. URL: https://doi.org/10.1007/978-3-030-00262-6_4.

[7] Natalia Juristo, Ana M Moreno, and Wolfgang Strigel. "Guest editors' introduction: Software testing practices in industry". In: IEEE software 23.4 (2006), pp. 19–21.

[8] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Jundefinednis Benefelds. "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application". In: Proceedings of the 39th ICSE-SEIP. 2017. ISBN: 9781538627174.

[9] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. "Effective Regression Test Case Selection: A Systematic Literature Review". In: ACM Comput. Surv. 50.2 (May 2017). ISSN: 0360-0300.

[10] Alessio Gambi, Marc Mueller, and Gordon Fraser. "Automatically Testing Self-Driving Cars with Search-Based Procedural Content Generation". In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, 318–328. ISBN: 9781450362245. DOI: 10.1145/3293882.3330566. URL: https://doi.org/10.1145/3293882.3330566.

[11] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-Directed Random Test Generation". In: Proceedings of the 29th International Conference on Software Engineering. ICSE '07. USA: IEEE Computer Society, 2007, 75–84. ISBN: 0769528287. DOI: 10.1109/ICSE.2007.37. URL: https://doi.org/10.1109/ICSE.2007.37.

[12] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. "Achieving Scalable Model-Based Testing through Test Case Diversity". In: 22.1 (Mar. 2013). ISSN: 1049-331X. DOI: 10.1145/2430536.2430540. URL: https://doi.org/10.1145/2430536.2430540.

[13] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. "Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs". In: Proc. ACM Program. Lang. 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428218. URL: https://doi.org/10.1145/3428218.

[14] Jiyuan Wang, Ming Gao, Yu Jiang, Jianguang Lou, Yue Gao, Dongmei Zhang, and Jiaguang Sun. QuanFuzz: Fuzz Testing of Quantum Program. 2018. arXiv: 1810.10310 [cs.SE].

[15] Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. "Property-Based Testing of Quantum Programs in Q#". In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. ICSEW'20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, 430–435. ISBN: 9781450379632. DOI: 10.1145/3387940.3391459. URL: https://doi.org/10.1145/3387940.3391459.

[28] We are currently discussing with the IBM Qiskit developers how to integrate our findings into their codebase.

[16] Andriy V. Miranskyy and Lei Zhang. "On Testing Quantum Programs". In: CoRR abs/1812.09261 (2018). arXiv: 1812.09261. URL: http://arxiv.org/abs/1812.09261.

[17] Michael A. Nielsen and Isaac L. Chuang. Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press, 2010.

[18] Yipeng Huang and Margaret Martonosi. "QDB: from quantum algorithms towards correct quantum programs". In: arXiv preprint arXiv:1811.05447 (2018).

[19] Yue Jia and Mark Harman. "An analysis and survey of the development of mutation testing". In: IEEE transactions on software engineering 37.5 (2010), pp. 649–678.

[20] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. "Does Mutation Testing Improve Testing Practices?" In: Proc. of the 43rd IEEE/ACM ICSE. 2021.

[21] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What It Would Take to Use Mutation Testing in Industry–A Study at Facebook. 2021. arXiv: 2010.13464 [cs.SE].

[22] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. "Practical Mutation Testing at Scale: A view from Google". In: IEEE TSE (2021).

[23] Goran Petrović and Marko Ivanković. "State of Mutation Testing at Google". In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 163–171. ISBN: 9781450356596. DOI: 10.1145/3183519.3183521. URL: https://doi.org/10.1145/3183519.3183521.

[24] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. "Are Mutants a Valid Substitute for Real Faults in Software Testing?" In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, 654–665. ISBN: 9781450330565. DOI: 10.1145/2635868.2635929. URL: https://doi.org/10.1145/2635868.2635929.

[25] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. "Open source software in quantum computing". In: PLOS ONE (2018).

[26] Gadi Aleksandrowicz et al. Qiskit: An Open-source Framework for Quantum Computing. Version 0.7.2. Jan. 2019. DOI: 10.5281/zenodo.2562111. URL: https://doi.org/10.5281/zenodo.2562111.

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

[28] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: SIAM Review 41.2 (1999), pp. 303–332. URL: https://doi.org/10.1137/S0036144598347011.

[29] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, 212–219. ISBN: 0897917855. URL: https://doi.org/10.1145/237814.237866.

[30] Daniel Fortunato, José Campos, and Rui Abreu. "Mutation Testing of Quantum Programs Written in QISKit". In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). 2022, pp. 358–359. DOI: 10.1109/ICSE-Companion55297.2022.9793776. URL: https://doi.org/10.1109/ICSE-Companion55297.2022.9793776.

[31] Jean-Luc Brylinski and Ranee Brylinski. "Universal quantum gates". In: Mathematics of quantum computation. Chapman and Hall/CRC, 2002, pp. 117–134.

[32] P. Liu, S. Hu, M. Pistoia, C. R. Chen, and J. M. Gambetta. "Stochastic Optimization of Quantum Programs". In: Computer 52.6 (2019), pp. 58–67.

[33] Pengzhan Zhao, Jianjun Zhao, and Lei Ma. "Identifying Bug Patterns in Quantum Programs". In: Proc. of the 2nd Q-SE. 2021.

[34] Python Software Foundation. Python – pass statement. https://docs.python.org/3/tutorial/controlflow.html#pass-statements. Accessed: 2021-08-24. 2021.

[35] Konrad Hałas. MutPy: A Mutation Testing Tool for Python 3.x Source Code. https://github.com/mutpy/mutpy. Accessed: 2021-01-18. Mar. 2011.

[36] Anders Hovmöller. Mutmut: a Python mutation testing system. https://github.com/boxed/mutmut. Accessed: 2021-01-18. Nov. 2016.

[37] Austin Bingham. Cosmic Ray: mutation testing for Python. https://github.com/sixty-north/cosmic-ray.

[38] Evan Kepner. mutatest: Python mutation testing. https://github.com/EvanKepner/mutatest.

[39] Daniel Fortunato, José Campos, and Rui Abreu. "QMutPy: A Mutation Testing Tool for Quantum Algorithms and Applications in Qiskit". In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, 797–800. ISBN: 9781450393799. DOI: 10.1145/3533767.3543296. URL: https://doi.org/10.1145/3533767.3543296.

[40] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. "Assessing the Effectiveness of Input and Output Coverage Criteria for Testing Quantum Programs". In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). 2021, pp. 13–23. DOI: 10.1109/ICST49551.2021.00014.

[41] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. "Muskit: A Mutation Analysis Tool for Quantum Software Testing". In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2021, pp. 1266–1270. DOI: 10.1109/ASE51524.2021.9678563.

[42] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. "QDiff: Differential Testing of Quantum Software Stacks". In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2021, pp. 692–704. DOI: 10.1109/ASE51524.2021.9678792.

[43] Javier Pellejero. MTQC: Mutation Testing for Quantum Computing. https://javpelle.github.io/MTQC. Accessed: 2021-01-18. June 2020.

[44] Daniel Méndez Fernández, Martin Monperrus, Robert Feldt, and Thomas Zimmermann. "The open science initiative of the Empirical Software Engineering journal". In: Empirical Software Engineering 24.3 (2019), pp. 1057–1060. ISSN: 1573-7616. DOI: 10.1007/s10664-019-09712-x. URL: https://doi.org/10.1007/s10664-019-09712-x.

[45] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. "Quantum Algorithm for Linear Systems of Equations". In: Phys. Rev. Lett. 103 (15 2009), p. 150502. URL: https://link.aps.org/doi/10.1103/PhysRevLett.103.150502.

[46] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. "Code coverage at Google". In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019, pp. 955–963.

[47] O. Tange. "GNU Parallel - The Command-Line Power Tool". In: ;login: The USENIX Magazine 36.1 (Feb. 2011), pp. 42–47. DOI: 10.5281/zenodo.16303. URL: http://www.gnu.org/s/parallel.

[48] William H. Kruskal and W. Allen Wallis. "Use of Ranks in One-Criterion Variance Analysis". In: Journal of the American Statistical Association 47.260 (1952), pp. 583–621. DOI: 10.1080/01621459.1952.10483441. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1952.10483441. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1952.10483441.

[49] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. Experimentation in software engineering. Springer Science & Business Media, 2012.

[50] J. Campos and A. Souto. "QBugs: A Collection of Reproducible Bugs in Quantum Algorithms and a Supporting Infrastructure to Enable Controlled Quantum Software Testing and Debugging Experiments". In: 2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE). Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 28–32. URL: https://doi.ieeecomputersociety.org/10.1109/Q-SE52541.2021.00013.

[51] William M McKeeman. "Differential Testing for Software". In: Digital Technical Journal 10.1 (1998), pp. 100–107.