Mutation Testing of Quantum Programs: A Case Study with QISKit

Anonymous Author(s)*

ABSTRACT

As quantum computing is still in its infancy, there is an inherent lack of knowledge and technology to properly test a quantum program. 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 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 the IBM's QISKit library. QMutPy has proven to be an effective quantum mutation tool, providing insight on the current state of quantum test suites and on how to improve them.

KEYWORDS

Quantum computing, Quantum software engineering, Quantum software testing, Quantum mutation testing

ACM Reference Format:
Anonymous Author(s). 2022. Mutation Testing of Quantum Programs: A Case Study with QISKit. In *ICSE'22: IEEE/ACM International Conference on Software Engineering, 21–29 May, 2022, Pittsburgh, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/AAA.BBB

1 INTRODUCTION

Quantum computation uses the qubit—the quantum-mechanical analogue of the classic bit—as its fundamental unit instead of the classic computing bit. Whereas classic 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 classic bits. Thus, quantum computers can, in theory, quickly solve problems that would be extremely difficult for classic computers. Such computation is possible because of qubit properties known as superposition of both 0 and 1, entangle multiple quantum bits, and interference [38, 42].

The field of quantum computing is evolving at a pace faster than originally anticipated [36]. 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 as 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 originally 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 in many, if not all, computer science disciplines [43], e.g., *software testing*. *Testing* is one of the most used techniques in software development to ensure software quality [5, 12]. It refers to the execution of the software *in vitro* environments that replicate (as close as possible) real scenarios to ascertain its correct behavior. Despite the fact that, in the classical computing realm, *testing* has been extensively investigated and several approaches and tools have been proposed [4, 13, 17, 23, 25, 31], such approaches for Quantum Programs (QPs) are still in their infancy [18, 27, 40]. Worth noting that (i) QPs are much harder to develop than classic programs and therefore programmers, mostly familiar with the classic world, are more likely to make mistakes in the counter-intuitive quantum programming one, and (ii) QPs are necessarily probabilistic and impossible to examine without disrupting execution or without compromising their results. Thus, ensuring a correct implementation of a QP is even harder in the quantum computing realm [20].

Mutation testing [22, 34] has been shown to be an effective technique in improving testing practices, hence helping in guaranteeing program correctness. Big tech companies, such as Google and Facebook, have conducted several studies [6, 33, 35] advocating for mutation testing and its benefits. The general principle underlying mutation testing is that the *bugs* considered to create versions of the program 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 classic 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE 2022, 21-29 May, 2022, Pittsburgh, USA
© 2022 Association for Computing Machinery.
ACM ISBN AAA-B-CCCC-DDDD-E/FF/GG...$15.00
https://doi.org/10.1145/AAA.BBB

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

computing. Thus, and in order 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 [11], IBM's Quantum Information Software Kit (QISKit) [1][2]. QISKit was 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 on simulators on a local computer. In a nutshell, QISKit translates QPs written in Python into a lower level language called OpenQASM [10], which is its quantum instruction language. Many famous quantum algorithms such as Shor [37] and Grover [14] have already been implemented using QISKit's API[3]. In detail, the main contributions of this paper are:

- A set of 5 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 [1] full-stack library.
- An empirical evaluation of QMutPy's effectiveness and efficiency on 24 real QPs.
- A detailed discussion on how to extend 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 is capable in generating fault-revealing quantum mutants and it 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 quality of the test assertions. Such improvements greatly increase the mutation score of the test suites — hence, leading to high quality QPs.

2 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 [1].

2.1 Quantum Mutation Operators

Similar to classic programs, a QP is fundamentally a circuit in which quantum bits (*qubits*) are initialized and go through a series of operations that change their state. These operations are commonly known and referenced to as *quantum gates*. Two of the most popular and used quantum gates are the NOT gate and the Hadamard gate, usually referred as the `x` gate and the `h` gate, respectively. They are single-qubit operations, i.e., they change the state of one qubit [8]. The `x` gate is analogous to the classic `NOT` gate where 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).

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

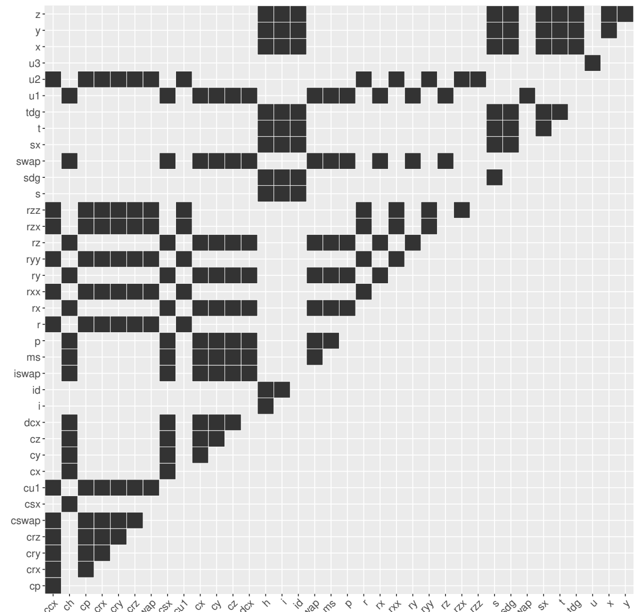


Figure 1: Equivalent gates in the QISKit full-stack library.

At the time of writing this paper, QISKit v0.29.0 provides support to more than 50 quantum gates[4]. This includes single-qubit gates (e.g., `h` gate), multiple-qubit gates (e.g., `cx` gate) and composed gates, or 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*.

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 required the same number and type of arguments.

The following subsections briefly describe the five quantum mutation operators proposed in this paper. Interested readers can refer to *the accompanying supplementary material* to find examples of each quantum mutation operator using the implementation of the Shor [37] algorithm available in the QISKit-Aqua's repository[6].

2.1.1 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,

[4] https://qiskit.org/documentation/apidoc/circuit_library.html
[5] Optional arguments are not taken into consideration.
[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

for the `x` quantum gate, 10 mutants are generated as there are 10 syntactically-equivalent gates (see Figure 1).

2.1.2 Quantum Gate Deletion (QGD). Adding and removing quantum gates from a QP can have a significant impact on its output. The QGD operation deletes an invocation to a quantum gate.

2.1.3 Quantum Gate Insertion (QGI). This quantum mutation operator performs the opposite action of the QGD operator. That is, 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 each 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.

2.1.4 Quantum Measurement Insertion (QMI). In quantum computing, measuring a qubit breaks the state of superposition and therefore the qubit's value becomes either 1 or 0 (as in classical computing), which can be considered a mutation by design. That is the underline idea of the QMI operator. It adds a call to the `measure` function[9] for each quantum gate call.

2.1.5 Quantum Measurement Deletion (QMD). Contrary to QMI, the QMD 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.

2.2 QMutPy Toolset

QPs written in Python and using QISKit library are a mix of classic operations (e.g., initialization of variables, loops), as well as 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 popular testing frameworks for Python: `unittest` and `pytest`.
- Supports various classic 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 3).
- 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 how we built QMutPy (available *in the accompanying supplementary material*) on top of an existing, already well adopted, mutation tool.

2.2.1 Python-based Mutation Testing Tools. Mutatest [26], mutmut [19], MutPy [16], and CosmicRay [7] are the most popular mutation testing tools for Python that are available through pip[10] (the package installer for Python). Albeit being open-source, fully automated, and support classic mutation operators, not all tools fulfil all our requirements.

Mutatest [26] 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 postmortem analysis (e.g., statistical analysis) could not be easily performed.

mutmut [19] does not allow one to instantiate the tool with a single mutation operator or a defined set of mutation operators. 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 [16] and Cosmic Ray [7] 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 in installing and running the tools, MutPy's learning curve is more gradual than Cosmic Ray. Thus, MutPy [16] tool is the one that fulfils all requirements we aim in a mutation tool.

2.2.2 MutPy Flow. 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 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 done 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, or timeout, the time it took to run the T on P, time it took to run T on each mutant.

2.2.3 QMutPy. MutPy is built in a way that it is straightforward to extend it with new mutation operators. Notwithstanding, addressing the technical challenges to implement the quantum operators, we added the possibility for the tool to mutate AST *Calls*[13].

3 EMPIRICAL STUDY

We have conducted an empirical study to evaluate QMutPy's effectiveness and efficiency at creating quantum mutants. In particular, we aim to answer the following research questions:

RQ1: How does QMutPy perform 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?

[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
[12] MutPy supports 20 classic 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] https://docs.python.org/3/library/ast.html#ast.Call

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

Table 1: 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.

As baseline, we have compared the results achieved by QMutPy's quantum mutation operators with MutPy's classic mutation operators[14]. Note that works [30, 32] on quantum mutation are very preliminary and no other classic or quantum mutation tool could have been used as baseline (see Sections 2.2.1 and 6).

We show our commitment to open science [29] by making QMutPy and our experimental infrastructure (data and scripts) available to the research community to assist in future research. The QMutPy tool, data, and scripts are available *in the accompanying supplementary material*. All artefact will immediately be publicly available on GitHub upon acceptance of this paper.

3.1 Experimental Subjects

To conduct our empirical study we require (1) real QPs written in the QISKit's framework [1] (as, currently, QMutPy only supports QISKit's quantum operations), (2) QPs written in Python[15], (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 main candidate sources of QPs that fulfil (1): the QISKit-Aqua's repository[16] itself, the "Programming Quantum framework repository Computers" book's repository[17] from O'Reilly, the "QISKit Textbook Source Code"'s repository[18] from the QISKit Community, and the official "QISKit tutorials"'s repository[19].

QISKit-Aqua's[20] repository provides the implementation of 24 QPs in Python, including the successful Shor [37], Grover [14], and HHL [15], 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 Jupiter 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 1 lists all QPs used in our empirical evaluation. For each program it provides the number of lines of code, the number of correspondent test cases, the time required to run the tests, and the code coverage at line level of the tests.

In total, the 24 QPs in the QISKit-Aqua's repository meet our criteria. On average, the considered QPs have 184 Lines of Code (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 differ greatly. 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 lines of code. This is in line with best practices [21] and also in line with a previous study conducted by Fingerhuth et al. [11] where ratio of code exercise by QPs' tests was slightly above the industry-expected standard.

3.2 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. To run all experiments in parallel we used the GNU Parallel tool [39].

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

[14] https://github.com/mutpy/mutpy#mutation-operators
[15] Although Jupiter Python notebooks include Python source code, they are not supported by QMutPy.
[16] https://github.com/Qiskit/qiskit-aqua/tree/stable/0.9/qiskit/aqua/algorithms
[17] https://github.com/oreilly-qc/oreilly-qc.github.io/tree/1b9f4c1/samples
[18] https://github.com/qiskit-community/qiskit-textbook/tree/3ffedf9
[19] https://github.com/Qiskit/qiskit-tutorials/tree/eb189a6
[20] 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.

465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522

3.3 Experimental Metrics

To be able to compare the effectiveness of each test suite at killing mutants we first compute its *mutation score* [22], 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:

$$\sum_{o \in O} \frac{\frac{|K_o|}{|M_o|-|I_o|},\ |M_o| - |I_o| > 0}{|O|} \times 100\% \tag{1}$$

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:

$$\sum_{o \in O} \frac{\frac{|K_o|}{|E_o|-|I_o|},\ |E_o| - |I_o| > 0}{|O|} \times 100\% \tag{2}$$

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 2), (2) time to inject a mutant in a non-mutated code (*Generate mutant* in Figure 2), (3) time to create a mutated module after injecting the mutant (*Create mutated module* in Figure 2).

3.4 Threats to Validity

Based on the guidelines in [41], 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). To minimize these threats, we selected QPs of various sizes, types, and levels of test coverage. Note that the lack of real-world QPs is a well-known challenge [2, 9]. 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).

4 RESULTS

Section 3 poses a set of research questions related to QMutPy's effectiveness and efficiency. The following subsections answer these questions in detail.

4.1 RQ1: How does QMutPy perform at creating quantum mutants?

Figure 2 shows the distribution of time QMutPy takes to generate a mutant using classic 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 classic 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 classic and quantum mutation operators.

> QMutPy takes up to 16x more time to generate quantum mutants than to generate classic mutants.

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

(1) *Mutation operators based on functions calls (i.e., calls to quantum gates).* Our set of quantum mutation operators, conversely to the classic ones, are based on function calls. Mutating a function is more complex than mutating, for example, a constant or a logical operator. It is worth noting that classic 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 specifically dedicated to quantum gates.

(3) *Search for an equivalent gate.* In QMutPy's current implementation, once a call to a quantum gate is found, quantum mutation operators QGI and QGR (the two most time-consuming operators) attempt to find a correspondent equivalent gate in the set of available operators. This issue could be mitigated by pre-processing the set of equivalent gates.

(4) *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.

> The generation of quantum mutants is more complex to perform than classic mutants and therefore, as expected, more time consuming. Given the low number of quantum mutants we were able to generate (see RQ2), we argue QMutPy's runtime at generating quantum mutants slightly affects the overall time spent on mutation testing.

523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580

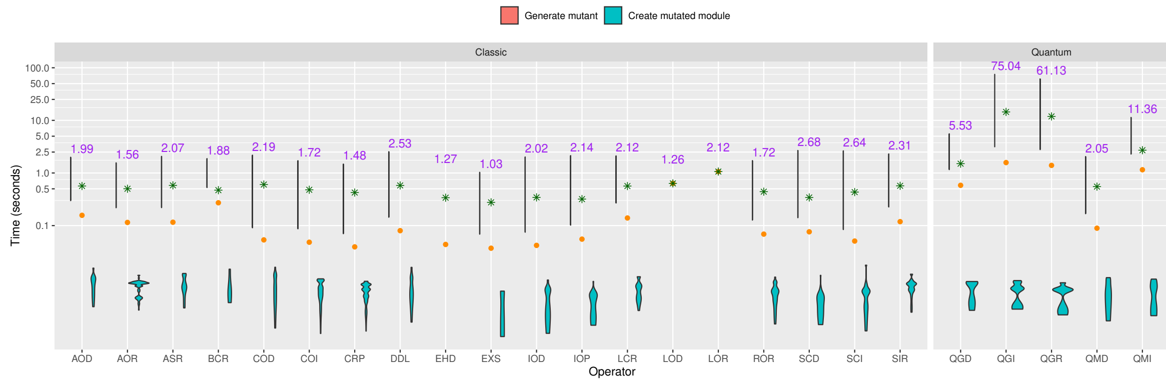


Figure 2: Distribution of the time required to inject a mutant and create a mutated target version. For each mutation operator, the purple text reports the maximum time of the 'Generate mutant' phase, i.e., time to inject or remove nodes from the AST, the green star reports the average time a mutation 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.

4.2 RQ2: How many quantum mutants are generated by QMutPy?

To answer this research question, we analyze our data at two different levels: (i) program level, i.e., how many quantum mutants are generated per program (see Table 2), and (ii) mutation operator, i.e., how many mutants are generated by each quantum mutation operator (see Table 3). For this research questions, we focus on the columns "# Mutants" and "# Mutated LOC" on both tables.

4.2.1 RQ2.1: How many quantum mutants are generated on each program? As we can see in Table 2 (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. Thus, further more quantum mutation operations 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 focus on specific function calls which might not occur as often as, e.g., classic arithmetic operations in a program, on average, QMutPy only mutated 4 lines of code with an average of 13 mutants per line (see column "# Mutated LOC"). In contrast, at least one classic mutant was generated for all programs. 147 mutants on average (+83) and 64 lines of code 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 lines of code that can be mutated using classic mutation operators than using quantum mutation operators, it is expected to have fewer quantum mutants in a QP.

4.2.2 RQ2.2: How many mutants are generated by each quantum mutation operator? As we can see in Table 3 (column "# Mutants"), on average, 140 mutants were generated by our set of quantum mutation operators. The quantum mutation 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.

Worth noting that the average number of mutants generated by our quantum mutation operators is slightly below the number of mutants generated by classic mutation operators (140 vs. 186, which is highly dominated by CRP). As there are many more lines of code that could be targeted by classic mutation operators (e.g., usage of constants) and many more classic operators (18 vs. our set of 5 quantum ones), it is expected that there are more classic 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 classic mutation operators (i.e., AOD, AOR, ASR, BCR, COD, COI, CRP, DDL, EHD, EXS, IHD, IOD, IOP, LCR, LOD, LOR, ROR, SCD, SCI, and SIR), 628 vs. 517 mutants.

> For 11 out of 24 QPs, QMutPy mutates 4 lines of code and generates 14 different mutants per mutated line. In total, it generates a total of 696 mutants, 140 per mutation operator.

4.3 RQ3: How do test suites for QPs perform at killing quantum mutants?

The goal of this question is to analyze the quality and resilience of test suites designed to verify QPs. As mentioned before, the idiosyncrasies underlying QPs (e.g., superposition, entanglement) makes 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 2 reports the results of performing mutation testing on the 24 QPs described in Table 1, 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, had a ratio of 102 killed mutants, 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

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

Table 2: Summary of our results per QP. Note that although 24 QPs were considered in our study, in here we only list the ones for which QMutPy was able to generate at least one mutant (either classic 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 lines of code 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 that were 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 on all mutants.

classic 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. These results show that the programs' test suites might have been designed to mainly verify the quantum aspect of each program as

- +10.86% more quantum mutants are killed than classic ones.
- Only 0.29% of all quantum mutants are not exercised the test suites, as opposed to 10.01% (+9.72%) of the classic mutants.

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. The mutation score achieved by each test suite ranged from 0% (`vqc` and `vqe`, more on this in Section 5.1) to 100% (`hhl` and `qsvm`). The mutation score achieved by all programs' test suites on classic 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 generated at least one mutant. That is, the programs' test suites achieved a higher mutation score on quantum mutants than on classic mutants, +20.62% (62.23% vs. 41.61%). Hence, reinforcing the idea that the test suites may 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

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

Table 3: Results per mutation operator. (Refer to Table 2 for an explanation of each column.)

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 classic mutants, programs' test suites took longer to run on quantum mutants than on classic. For example, `qsvm`'s test suite took 47.85 minutes to run on the only generated quantum mutant and 4.79 minutes on average ($\frac{674.82 \text{ minutes}}{141 \text{ classic mutants}}$) on each classic mutant. The reasons behind these time differences are explained in Section 4.1.

> Test suites for QPs achieved a low mutation score on quantum mutants (62.23%), although +28.72% higher than the mutation score achieved on classic mutants.

4.4 RQ4: How many test cases are required to kill or timeout a quantum mutant?

The goal of this question is to understand the effectiveness of current quantum test suites. Figure 3 shows the distribution of the 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 — 73 tests for QMD). The average number of tests needed to kill or timeout each classic mutant is 26, with 10 out of 18 classic 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 classic mutants, the average number of tests needed to kill or timeout each classic 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.

As fewer tests are required to kill quantum mutants than to kill classic mutants, these results are in line with the assumption that these test suites primarily check quantum-related behavior.

> On average, quantum mutants require -65% tests to be killed or timeout than classic mutants (9 vs. 26).

4.5 RQ5: How are quantum mutants killed?

With this question we aim to analyze what kills quantum mutants.

We have observed that, out of the 1589 killed mutants, two-thirds of mutants are killed by *errors* (1067) and the other one-third by *test assertions* (522). Figure 4 reports the number of mutants killed by *errors* and *test assertions* per mutation operator. Overall, the majority of classic mutants are killed by *errors*. As already mentioned, we argue that QISKit test suites are mainly designed to check for the correct behavior of QPs. Therefore, they are less resilient to classic 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 randomly inserted, the circuit may become unprocessable and an error is thrown.

> Quantum mutants are mainly killed by *test assertions* (with the exception of QMI mutants). Classic mutants, on the other hand, are mainly killed by *errors*.

5 IMPROVING QUANTUM TEST SUITES

The results in Section 4 suggest that, despite the average quantum mutation score of the QISKit's test suites is high, there is room for improvement. For example, we observed that 150 out of the 207 quantum mutants generated for `shor` survived.

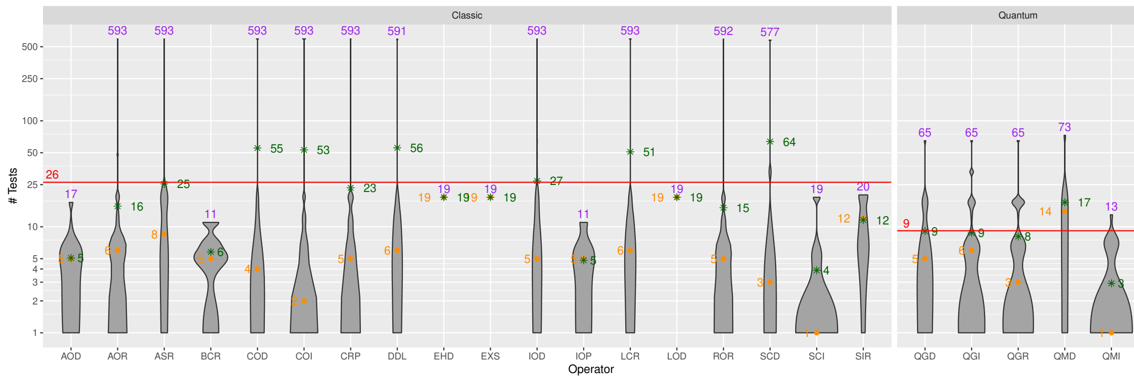
We draw on two hypotheses to guide our discussion on how to improve QPs' test suites to kill more quantum mutants:

- 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.

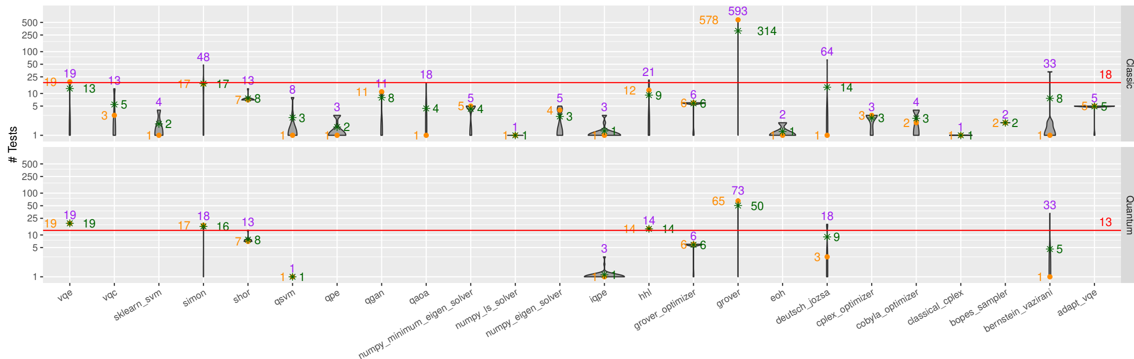
Note that the described mutations and improvements to the test suites are available *in the accompanying supplementary material*, and will be publicly available on GitHub upon acceptance.

5.1 Improving coverage

Figure 5 shows the relation between coverage and mutation score of each test suite. We can see that, on one hand, QPs' test suite with higher coverage tend to achieve higher mutation scores. `bernstein_vazirani`, `simon`, and `deutsch_jozsa` are three of the



(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 3: 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 classic mutant or a quantum mutant.

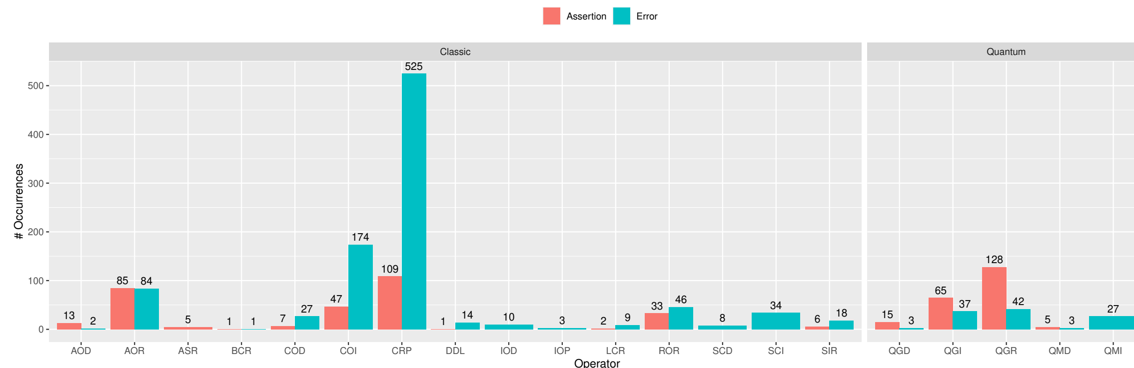


Figure 4: Number of mutants killed by an *assertions* or an *error* per mutation operator. In our experiments we found three types of *errors* thrown by the test suites. (1) Qiskit-related: `AquaError`, `QiskitOptimizationError`, `QiskitError`, and `CircuitError`. (2) Python: `NotImplementedError`, `IndexError`, `ValueError`, `AttributeError`, `IsADirectoryError`, `ZeroDivisionError`, `OverflowError`, `UnboundLocalError`, `RuntimeError`, `NameError`, and `KeyError`. (3) Third-party: `CplexSolverError`, `DQCPError`, `AxisError` and `LinAlgError`.

QPs with the highest coverage and mutation score. On the other hand, `cplex_optimizer`, `adapt_vqe`, and `bopes_sampler` are the lowest in terms of code coverage and mutation score. Thus, with this first hypothesis we aim to investigate whether increasing the coverage of QPs, e.g., covering mutated lines of code that are not exercised by the program's test suite, leads to a higher mutation score.

Table 2 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.

We extended `hhl`'s and `vqc`'s test suite[21] [22] to cover these methods and added a more specific test assertion to each test. By re-running the mutation analysis using the augmented test suites, we

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

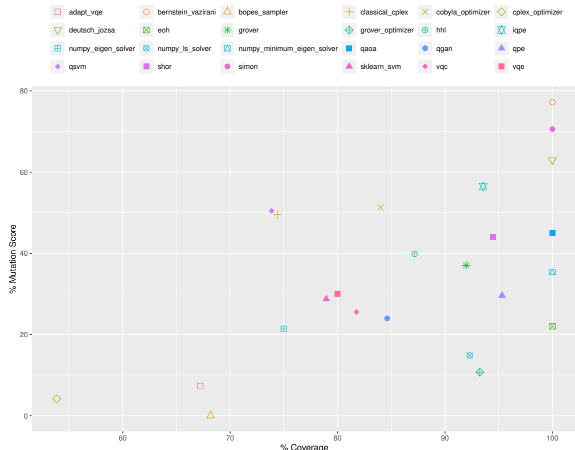


Figure 5: % Mutation score vs. % Coverage.

verified that our hypothesis hold. 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%).

5.2 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, and 11. Suppose that the correct behavior is to observe 00 with 25% probability and 11 with 75%. If, instead, we observe survived mutants for all the qubit values 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[23] (the QP with the most generated quantum mutants, see Table 2) with additional test assertions. The added assertions check the correctness of the number of obtained measurement values.

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 the 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 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.

6 RELATED WORK

Liu et al. [28] has shown that quantum mutation is useful to check the correct behavior of QPs. They proposed a search-based technique leveraging six mutation operations: insert and remove an operation, swap two operations, replace a gate in an operation, replace qubits in an operation, and replace an operation. The technique aims at reducing QPs' runtime while keeping their correctness.

Regarding quantum mutation tools, to the best of our knowledge MTQC [32] and Muskit [30] are the only two — preliminary — works that were published before. MTQC is a Java-based quantum mutation testing tool that uses a Graphic User Interface (GUI) to perform mutations on either QISKit or Q# QPs. MTQC performs primitive, custom mutations, albeit we can say that the operation it performs is similar to our QGR operation. However, the concept of equivalent gates is not defined and the gate swaps performed are a subset of our set of equivalent gates. They implement 52 isolated operations, each replacing a gate by another. Compared to MTQC, QMutPy is a fully automatic approach and offers a larger set of mutation operators, including classic ones.

Muskit [30] is a Python mutation tool that is provided as a command line interface, a GUI, and a web application. Muskit supports 19 QISKit gates and can perform three quantum mutation operations: add, remove, and replace gate. These are similar to our QGI, QGD, and QGR mutation operators, respectively. QMutPy, on the other hand, supports 40 gates (+21) and two additional mutation operators. Although Muskit has also been tailored for QISKit programs, it cannot be used out-of-the-box on, e.g., the 24 QPs evaluated in our empirical study. Muskit either uses a manually-written test suite or automatically generates a new suite [2]. Note that both test suites are sequences of test inputs and not complex sequence of code statements (e.g., calls to constructors to instantiate objects, method calls on these objects) as the ones used in our study which are required to test the 24 QPs. Furthermore, Muskit's test analyzer requires a program specification to determine whether a mutant has been killed by a test case. No specification is available for the 24 QPs considered in our study and writing one would require expertise on QISKit and on quantum computing.

7 CONCLUSIONS

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. To demonstrate the effectiveness of QMutPy, we have 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 less test cases to be killed than classic 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 showed how both improvements can increase the mutation score significantly on the QPs considered in our study[24].

As for future work, we plan to extend QMutPy with other mutation operators, and offer QMutPy to other quantum frameworks (e.g., Cirq and Q#). Moreover, combining QMutPy with techniques to automatically generate test suites for QPs [3, 18, 27, 40] is an interesting venue for future work. Finally, we plan to run our mutation analysis on real quantum computers (and not just simulators) and check for potential differences.

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

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

REFERENCES

[1] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. 2019. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2562111

[2] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. 2021. 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)*. IEEE, 13–23.

[3] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. 2021. 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)*. 13–23. https://doi.org/10.1109/ICST49551.2021.00014

[4] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Jundefinednis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track* (Buenos Aires, Argentina) *(ICSE-SEIP '17)*. IEEE Press, 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27

[5] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing* (1 ed.). Cambridge University Press, USA.

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

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

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

[9] J. Campos and A. Souto. 2021. 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 28–32. https://doi.ieeecomputersociety.org/10.1109/Q-SE52541.2021.00013

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

[11] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. 2018. Open source software in quantum computing. *PLOS ONE* 13, 12 (12 2018), 1–28. https://doi.org/10.1371/journal.pone.0208561

[12] Gordon Fraser and José Miguel Rojas. 2019. *Software Testing*. Springer International Publishing, Cham, 123–192. https://doi.org/10.1007/978-3-030-00262-6_4

[13] Alessio Gambi, Marc Mueller, and Gordon Fraser. 2019. 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 318–328. https://doi.org/10.1145/3293882.3330566

[14] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866

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

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

[17] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. 2013. Achieving Scalable Model-Based Testing through Test Case Diversity. 22, 1, Article 6 (March 2013), 42 pages. https://doi.org/10.1145/2430536.2430540

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

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

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

[21] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. 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*. 955–963.

[22] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[23] Natalia Juristo, Ana M Moreno, and Wolfgang Strigel. 2006. Guest editors' introduction: Software testing practices in industry. *IEEE software* 23, 4 (2006), 19–21.

[24] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. 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* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[25] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective Regression Test Case Selection: A Systematic Literature Review. *ACM Comput. Surv.* 50, 2, Article 29 (May 2017), 32 pages.

[26] Evan Kepner. [n.d.]. mutatest: Python mutation testing. https://github.com/EvanKepner/mutatest.

[27] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 150 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428218

[28] P. Liu, S. Hu, M. Pistoia, C. R. Chen, and J. M. Gambetta. 2019. Stochastic Optimization of Quantum Programs. *Computer* 52, 6 (2019), 58–67.

[29] Daniel Méndez Fernández, Martin Monperrus, Robert Feldt, and Thomas Zimmermann. 2019. The open science initiative of the Empirical Software Engineering journal. *Empirical Software Engineering* 24, 3 (01 Jun 2019), 1057–1060. https://doi.org/10.1007/s10664-019-09712-x

[30] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. 2021. *Muskit: A Mutation Analysis Tool for Quantum Software Testing*. Technical Report.

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

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

[33] Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 163–171. https://doi.org/10.1145/3183519.3183521

[34] Goran Petrovi'c, Marko Ivankovi'c, G. Fraser, and René Just. 2021. Does Mutation Testing Improve Testing Practices? *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), 910–921.

[35] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. 2021. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3107634

[36] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018), 79. https://doi.org/10.22331/q-2018-08-06-79

[37] Peter W. Shor. 1999. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Rev.* 41, 2 (1999), 303–332. https://doi.org/10.1137/S0036144598347011

[38] Andrew Steane. 1998. Quantum computing. *Reports on Progress in Physics* 61, 2 (1998), 117.

[39] O. Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb. 2011), 42–47. https://doi.org/10.5281/zenodo.16303

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

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

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

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