Which Quantum Circuit Mutants Shall Be Used?
An Empirical Evaluation of Quantum Circuit Mutations

Eñaut Mendiluze Usandizaga
Simula Research Laboratory
Oslo, Norway
enaut@simula.no

Tao Yue
Simula Research Laboratory
Oslo, Norway
taoyue@gmail.com

Paolo Arcaini
National Institute of Informatics
Tokio, Japan
arcaini@nii.ac.jp

Shaukat Ali
Simula Research Laboratory
Oslo, Norway
shaukat@simula.no

Abstract

As a new research area, quantum software testing lacks systematic testing benchmarks to assess testing techniques' effectiveness. Recently, some open-source benchmarks and mutation analysis tools have emerged. However, there is insufficient evidence on how various quantum circuit characteristics (e.g., circuit depth, number of quantum gates), algorithms (e.g., Quantum Approximate Optimization Algorithm), and mutation characteristics (e.g., mutation operators) affect the most mutant detection in quantum circuits. Studying such relations is important to systematically design faulty benchmarks with varied attributes (e.g., the difficulty in detecting a seeded fault) to facilitate assessing the cost-effectiveness of quantum software testing techniques efficiently. To this end, we present a large-scale empirical evaluation with more than 700K faulty benchmarks (quantum circuits) generated by mutating 382 real-world quantum circuits. Based on the results, we provide valuable insights for researchers to define systematic quantum mutation analysis techniques. We also provide a tool to recommend mutants to users based on chosen characteristics (e.g., a quantum algorithm type) and the required difficulty of killing mutants. Finally, we also provide faulty benchmarks that can already be used to assess the cost-effectiveness of quantum software testing techniques.

***K*eywords** quantum software testing, mutation analysis, benchmarks, quantum circuit

1 Introduction

Quantum Computing (QC) is a fairly recent field that is advancing quickly [1], promising to revolutionize computing by offering solutions to some complex problems with the enormous computational power of quantum computers. Quantum software empowers the QC application development [2]. Naturally, there is a growing need to test quantum software to ensure quantum software's correctness. To this end, several quantum software testing techniques have emerged in the last few years [3, 4, 5, 6, 7, 8, 9, 10].

Quantum software testing techniques need benchmarks such that their cost-effectiveness can be assessed. To this end, some open-source benchmarks appeared recently [11, 12, 13, 14]. However, such benchmarks are small-scale and do not provide systematic classifications of bug features that could be used to systematically assess the effectiveness of quantum software testing techniques. At the same time, some quantum mutation analysis techniques with tools have been published recently [15, 5, 7, 8]. However, these tools generate too many mutants, which become infeasible to execute due to scarce QC resources. Even if the execution is not an issue, many mutants generated by these tools are redundant and are often too easy to kill; thereby are not useful for testing. Finally, these tools do not provide a systematic and intelligent way to generate a small subset of mutants with varied characteristics such that quantum mutation testing techniques can be assessed more systematically.

In general, there is no sufficient understanding of quantum mutations, e.g., *which mutants* are difficult to detect, *where to seed* a fault in a quantum circuit so that it is difficult to detect, and *which types* of mutations are related to each algorithm type. To build such understanding, to generate new knowledge about quantum mutants, and to generate faulty benchmarks of different characteristics to systematically and efficiently assess the cost-effectiveness of quantum software testing techniques, we present results of a large-scale empirical evaluation with more than 700K faulty benchmarks generated with an existing quantum mutation analysis tool [15]. Each generated benchmark is a faulty version of an original quantum circuit and is called *faulty benchmark*.

This empirical evaluation aims to study various mutation characteristics (e.g., mutation operator types), quantum algorithms and their classification (e.g., Variational Quantum Eigensolver), and circuit characteristics (e.g., circuit depth and the number of gates) on the "survivability" of faulty benchmarks, i.e., whether the fault seeded in a benchmark can survive the fault detection of a quantum software testing technique. Our motivation for choosing survivability is that we want to assess the difficulty of detecting a seeded fault. To this end, such survivability indicates how various mutation and circuit characteristics, and quantum algorithms and their classification, play a role in the effectiveness of detecting the fault in a faulty benchmark with a given quantum software testing technique.

Based on the results, key observations are: First, we found that faulty benchmarks generated with the add mutation operator (i.e., adding a new quantum gate) have higher survivability than removing or replacing a quantum gate from a circuit. Second, regarding the position where a mutation operator is applied to create a faulty benchmark, mutating at the beginning or the end of the circuit leads to higher survivability, concluding that mutating the middle part of the circuit will likely change circuit behavior. Third, survivability is strongly related to the algorithm used. Notably, the algorithms that are designed to produce one dominant output, i.e., output with the highest probability (e.g., optimization algorithms), are likely to lead to high survival rates. Finally, we also found no significant correlation between the circuit complexity characteristics (e.g., the number of qubits in a circuit) and survivability.

Our contributions are:

1) A comprehensive empirical study to generate new knowledge on understanding relationships of mutation characteristics, circuit characteristics, quantum algorithms, and their interactions with the survivability of faulty benchmarks. Such knowledge does not exist;
2) A command line-based recommendation tool that can assist users in generating a desired number of faulty benchmarks of varied survivability by considering characteristics of interested quantum algorithms;
3) A large-scale faulty benchmark consisting of more than 700K faulty benchmark circuits that can be readily used to assess the cost-effectiveness of quantum software testing techniques.

All detailed experiment results, code, and data are provided in the online repository [16].

Paper structure: Section 2 and Section 3 present the background and related work, respectively. We present the design of the empirical study in Section 4, and results and discussion in Section 5. We conclude the paper in Section 6.

2 Background

Mutation Analysis. Mutation analysis is a widely used technique in software engineering to evaluate the effectiveness and quality of testing techniques. A typical mutation analysis process systematically uses mutation operators to introduce changes to the correct program. As a result, each change produces a faulty version (*mutant*) of the program. Test cases are executed on mutants to determine which test case can detect the mutants. The testing results measure the adequacy of testing techniques used to generate test cases [17].

The mutants can have different behaviors depending on the operator applied and the program itself. Within this context, each mutant can be categorized into several groups. First, regarding whether the testing technique can detect it or not, a mutant is classified as *killed*, i.e., detected, or as a *survivor*, i.e., not detected. On the other hand, those mutants that are not even able to be executed, due to some syntactical errors, are called the *stillborn* mutants [18].

Each group is further divided into some subcategories. Inside the not-detected mutants, one of the main challenges of mutation analysis is related to *equivalent mutants*. Equivalent mutants are those that behave the same as the original program for all the inputs, thus, they cannot be detected by any test. Moreover, another critical category is constituted by *redundant mutants* that make minimal contributions to the testing process. Those are the ones that are detected whenever other mutants are detected. This category includes *duplicated* mutants, which are equivalent to each other but not to the original program, and *subsumed* mutants, jointly detected when other mutants are killed. Any error detected in a subsumed mutant is also identified by the first mutant [19, 20, 21, 22].

Classically, each of the detected mutants is also categorized depending on the killing strength. *Weakly killed mutants* that expose differences in the program state immediately after execution compared to the execution of the original program; *firmly killed mutants* that expose differences at a later point; and *strongly killed mutants* that show observable differences in the outputs [23].

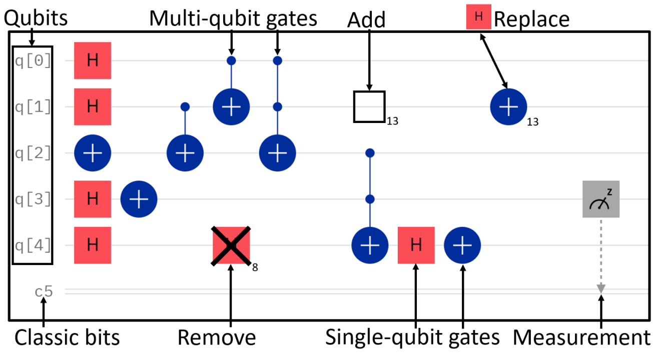


Figure 1: A quantum circuit example. The circuit has five qubits (e.g., $q[0]$) and a group of five classical bits (i.e., together denoted as c_5). The measurement collapses the state of $q[3]$. The three selected operators are: *Add* a gate at position 13 in the circuit, *Remove* the Hadamard gate at position 8, and *Replace* the NOT gate with a Hadamard gate at position 13.

Quantum Computing. The main difference between a quantum computer and a classical computer is the smallest data unit on which they perform computations. Such data unit in classical computing is a bit, whereas, in Quantum Computing (QC), it is a quantum bit (qubit). A classical bit can only have a value of 0 or 1 at a one-time point, whereas a qubit can be in a *superposition* state of both 0 and 1. Such phenomenon (i.e., superposition) is one of the special quantum characteristics leading to quantum speedup. Another key quantum characteristic is *entanglement*, where two or more qubits are strongly correlated, e.g., they will always be in the same state [24]. Another special characteristic of QC is that it obeys the *no-cloning theorem* [25], i.e., one cannot simply copy or measure a quantum state since it will result in collapsing a quantum state from a superposition state to a definite state (classical state) [26]. In QC, the collapse of a qubit is used to obtain the measurement result of a quantum operation [26].

Figure 1 shows an example of a quantum circuit visually drawn in IBM's quantum circuit composer [27]. Figure 1 shows the key elements of the circuit having five qubits ($q[0]$ to $q[4]$) and a group of five classical bits, i.e., c_5. A quantum circuit performs computations with quantum gates on qubits. A quantum gate takes qubit(s) as inputs and performs computation to alter the state of the quantum circuit. If a quantum gate operates on one qubit, we call it a *Single Qubit Gate*, e.g., a Hadamard gate (H) in the circuit. A *Multi-qubit Gate* operates on more than one qubit (e.g., conditional NOT gate (CNOT), with the control qubit showing with a small filled circle and the target qubit symboled with a + sign within a circle). A measurement is also shown, which, in this example circuit, will collapse the state of $q[3]$ into a classical definite state.

We introduce faults in quantum circuits by introducing small changes at the quantum gate level, by adding, removing, or replacing quantum gates in an original quantum circuit to generate faulty benchmarks. These are referred to as *mutation operator types* in [15]. Figure 1 shows examples of these operator types. For example, we replace a NOT gate (+) with a Hadamard gate (H). Such a way of creating faulty versions of quantum circuits for assessing testing techniques has been performed in some existing works [4, 3, 28, 15, 5, 7, 8]. Another important characteristic is *quantum gate type*, e.g., we can add any available gate in a quantum circuit, whereas we can delete or replace only quantum gates already present in a quantum circuit. This characteristic is called *gate type* in [15]. The third characteristic is the position in a quantum circuit where a fault is introduced, and further details can be consulted in [15].

3 Related Work

Classic Mutation Analysis. The surveys of Jia and Harman [29] and Papadakis et al. [30] show the growing interest of the research community in mutation analysis. Inside mutation analysis, in order to improve and adapt the mutation to different use cases, several different strategies have been developed. Some of them involve new mutant generation techniques or new mutant selection strategies [29, 30]. Different works have proposed operators for specific programming languages [31, 32, 33, 34, 35], categories of programming languages [36, 37, 38, 39], categories of applications [40, 41, 42, 43, 44, 45], and specific bug categories [45, 46, 47, 48, 49, 50]. Other works, instead, focused on devising new reduction strategies [51, 52, 53, 54, 55, 56].

Papadakis et al. [30] propose a checklist of best practices for using mutation testing in the context of controlled experiments for classical programs. Instead, we focus on quantum circuits and provide guidelines to select mutants for quantum circuits based on an extensive empirical evaluation.

Different works assessed the effectiveness of mutation operators for different artefacts, as we do in this work for mutation operators for quantum programs. Smith et al. [57] conducted an empirical evaluation to assess the effectiveness of MuJava's mutation operators [58] in software testing. The study categorized the behavior of mutants generated by selected mutation operators during successive attempts to detect them. The categorization used in the study includes the crossfire (subsumed mutants), dead on arrival (mutants killed by the initial test suite without any specific focus on mutation testing), Killed, and Stubborn (equivalent) mutants. This categorization provides a deeper understanding of the performance of individual operators and the behaviors exhibited by their resultant mutants. Our work studies the quantum mutation operators, and quantum mutants generated by them, across different quantum circuits. Our study presents a classification ranking for different mutants and their characteristics, providing valuable observations about their behavior and use.

Just et al. [59] conducted a study to assess whether mutants can be an alternative to real faults. The study uses real faults from subject programs and compares the effectiveness of developer-written and automatically generated test suites in detecting these faults. The study investigated the correlation between real faults and mutants generated by commonly used mutation operators. It found that a statistically significant correlation exists for 73% of the real faults. The study also proposed improvements to the mutation analysis technique by introducing new or stronger mutation operators. They observed that 17% of faults were not coupled to any mutants, which reveals a fundamental limitation of mutation analysis, and the other 10% of actual faults required implementing new mutation operators. Even though our study does not directly relate to real faults, it helps to do so by conducting a substantial empirical study that can be used as a reference point for associating them with real quantum faults.

Zhang et al. [60] explored how mutation analysis can be used for assessing the quality of use case models with use case specifications detailed in restricted natural language, with the ultimate goal of supporting requirements inspection. They proposed a taxonomy of defect types and defined nearly 200 mutation operators. A set of case studies demonstrated the feasibility of the proposed mutation analysis methodology. In contrast to this work, our study focuses on mutation analysis for quantum circuits.

Quantum Mutation Analysis. Two key mutation analysis tools are available, Muskit [15] and QMutPy [5, 7, 8]. Both tools can generate mutated quantum circuits with mutation operators, e.g., related to adding, removing, or replacing quantum gates. QMutPy further looks into bug patterns identified in [61] and mutating a quantum gate with another "syntactically-equivalent" with the same number and types of arguments. QMutPy and Muskit could generate many mutants, which might be infeasible to execute and even redundant. This paper instead studies the influence of various mutation characteristics, circuit characteristics, and algorithms and their classifications on the survivability of mutants via large-scale empirical evaluation to collect evidence to support researchers and practitioners in selecting meaningful mutants for mutation analysis. Moreover, based on the evidence collected, diverse mutant generation strategies can be developed and possibly integrated into QMutPy or Muskit. In addition, some works have employed manually created mutants to assess the effectiveness of their testing techniques [4, 3, 28, 9]. Note that these papers were published before QMutPy and Muskit were released, which automate the generation of mutants, including the ones used by these works.

Wang et al. [6] proposed a mutation-based approach (*MutTG*) for generating the minimum number of test cases that maximizes the number of killed mutants to save the cost required to execute many test cases. *MutTG* also defines a metric to measure the difficulty of killing a mutant based on the number of inputs that can kill the mutant out of the total number of inputs. The higher the number of inputs killing the mutant, the easier it is to kill the mutant. Instead, we study the relationships between different circuit and mutant characteristics and quantum algorithms on the survivability of mutants with a large-scale empirical evaluation.

Quantum Software Bug Repositories. Recently, a few bug repositories have been published. The Bugs4Q benchmark suite [11] collects bugs from the Qiskit GitHub repository, ensuring that each bug has both a buggy and a fixed version. They collected 36 bugs. In [13], a proposal is presented for reproducible bugs in quantum software with a set of quantum programs, their corresponding bugs, and infrastructure to support experimentation. In [14], a multi-lingual benchmark for property-based testing of quantum programs coded in Q# is proposed, consisting of a set of programs in Q# and corresponding properties. These works provide small-scale benchmarks and do not make a systematic analysis of mutant characteristics, thereby giving no evidence about how easy it is to detect which mutants and which circuit characteristics make it difficult to detect a mutant. Thus, this paper presents a large-scale empirical evaluation to study how various mutant and circuit characteristics, algorithms, and their interactions affect detecting mutants.

4 Experiment Design

We first define various characteristics in Section 4.1 followed by research questions in Section 4.2. We describe metrics, subject systems, mutant generation, and experimental setup together with execution in Sections 4.3–4.6, respectively.

4.1 Characteristics of Mutations, Circuits, and Algorithms – Independent Variables

4.1.1 Mutation Characteristics

Mutation Operator Type (*Operator*): We have three types of mutation operators, i.e., adding (*Add*), removing (*Remove*), or replacing (*Replace*) a quantum gate as described in Section 2. **Quantum Gate Mutations:** We study quantum gate characteristics from three perspectives. First, we study *mutated gates* (*Gate*) such as *Hadamard* and *CNOT* with a mutation operator (e.g., *Add*).

In total, we have 19 gates that are currently implemented in Muskit [15], which we used as a mutation framework. These gates are *ccx*, *cswap*, *cx*, *cz*, *h*, *id*, *p*, *rx*, *rxx*, *ry*, *rzz*, *s*, *swap*, *sx*, *t*, *x*, *y*, and *z*. Interested readers may contact the following reference for more details about each gate [62]. Second, we study *mutated gate types* (*Gate Type*) by classifying the implemented gates in Muskit into these seven categories representing the basic building blocks of Qiskit, i.e., Controlled gates (*Controlled*), Hadamard gates (*Hadamard*), Pauli gates (*Pauli*), Phase gates (*Phase*), Rotation gates (*Rotation*), Swap gates (*Swap*), and T gates (*T*). Third, we study *mutated gate size* (*Gate Size*), which classifies quantum gates into two categories: single-qubit (*Single*) and multi-qubit gates (*Multi*). This classification is common in quantum circuit design [62]. Note that these three independent variables (i.e., *Gate*, *Gate Type*, and *Gate Size*) are intertwined, which makes it very difficult to interpret their interaction effects. As a result, we do not study their interactions. **Position (*Position*:** We study the position in the circuit where a change is introduced as described in Section 2. Given that the total number of positions varies from one circuit to another, we use the *relative position* to the whole in terms of percentage, i.e., 10%, 20%, . . . , 100% to describe the position in the quantum circuit where a fault is seeded. For instance, 10% means the first 10% of the positions in a circuit.

4.1.2 Algorithms Characteristics

We study the effect of various algorithms (*Algorithm*) and their categorization from two aspects on the survivability of a faulty benchmark. We have 28 algorithms from MQT Bench, i.e., QP1–QP28 as shown at the bottom of Table 1. More details about these algorithms can be found in [63]. Moreover, we use a classification from [63] to classify the 28 algorithms into 12 categories and name this classification as *Algorithm Group*. Its 12 categories are *ae*, *dj*, *ghz*, *graphstate*, *grover*, *qaoa*, *qft*, *qgan*, *qpe*, *qwalk*, *vqe*, and *wstate*. An interested reader can consult [63] for more details on it. In addition, we classify all the algorithms into two categories according to their *Output Dominance*: (1) *output-dominant* algorithms that focus on finding a dominant output with a maximum probability, such as the case for optimization algorithms. For such algorithms, we check if they produce a dominant output that matches the expected one. In total, we have 19 output-dominant algorithms; (2) *diverse-output* algorithms with many outputs of different probabilities. As a result, to check the correctness of diverse-output algorithms, we need to compare all possible outputs and their probabilities with the expected ones. In total, we have nine such algorithms.

4.1.3 Circuit Characteristics

Circuit Complexity: We study the typical metrics used to measure the complexity of circuits, i.e., the *number of qubits* (*#qubits*), the *total number of quantum gates* (*#gates*), and the *number of measurements* (*#measurements*), counting the numbers of qubits, gates, and measurements in a circuit. In addition, we use circuit depth (*depth*), a commonly used metric, to measure the complexity of a quantum circuit, which is defined as the length of the longest path (measured as the number of gates) of the circuit from its beginning to the end. **Gate Complexity.** We study three characteristics, i.e., the number of single-qubit gates (*#singleGates*), the number of multi-qubit gates (*#multiGates*), and the number of entangled qubits (*#eQubits*) to assess the effect of gate complexity on survivability. We count the number of entangled qubits in a circuit by checking all its interaction states in the circuit and the qubits they relate to.

4.2 Research Questions

- **RQ1**: How do the various quantum mutation characteristics influence the survivability of faulty benchmarks? This RQ is further divided into three sub-research questions:
  - ***RQ1.1*** focuses on studying each characteristic individually.
  - ***RQ1.2*** focuses on pair-wise interactions between characteristics.
  - ***RQ1.3*** focuses on interactions among all the characteristics.

  In RQ1, we study the main mutation characteristics (Section 4.1.1) and their interactions.
- **RQ2**: How does a quantum algorithm or quantum algorithm type affect the survivability of faulty benchmarks? RQ2 studies individual algorithms and the effect of their two types of outputs (Section 4.1.2) on the survivability of faulty benchmarks:
  - ***RQ2.1*** focuses on studying the difference of the output dominance in the algorithms.
  - ***RQ2.2*** focuses on algorithms group characteristics.
  - ***RQ2.3*** focuses on individual algorithm characteristics.
- **RQ3**: How do the characteristics of a quantum circuit affect the survivability of its faulty benchmarks? In particular, we study the influence of circuit and gate complexity on survivability (Section 4.1.3).
- **RQ4**: How do the interactions of the algorithm characteristics with the mutation characteristics influence the survivability of faulty benchmarks? This RQ is further divided into three sub-research questions:
  - ***RQ4.1*** focuses on individual algorithms.

Which quantum circuit mutants shall be used?

- ***RQ4.2*** focuses on algorithms groups.
- ***RQ4.3*** focuses on algorithm classification based on output dominance.

Note that we also studied interactions with more than one mutation characteristic. However, given many possible combinations, they are only considered when automatically generating recommendations (Section 5.5). Nonetheless, all interaction data is available in our repository [16].

4.3 Metrics and Statistical Tests

To quantify the effect of the mutation and circuit characteristics, algorithms and their classifications, and interactions (i.e., captured as independent variables) on the survivability of faulty benchmarks (the dependent variable), we define metric *Survival Rate* (SR). The survival rate refers to the percentage of survived mutants (i.e., undetected) obtained for a particular independent variable. This is in contrast to a typically used metric, mutation score, which is based on killing the mutants and is used to evaluate the test cases. Since our approach does not aim to evaluate the test cases, we decided to use a different metric that focuses on categorizing the mutants. The motivation was that we wanted to study the characteristics of mutants that are difficult to kill; therefore, we chose a metric that focuses on mutants that are not killed. The metric is calculated by dividing the number of survived mutants divided by the total number of mutants, corresponding to each independent variable as:

$$SR_{IV} = \frac{totalSurvivors_{IV}}{totalMutants_{IV}}$$

IV represents a set of independent variables corresponding to each mutant, algorithm, circuit characteristics, and interactions, e.g., *Gate*. $totalSurvivors_{IV}$ represents the total number of survived mutants for a particular independent variable, e.g., for *Gate* independent variable, an example is *h* gate. $totalMutants_{IV}$ represents the total number of mutants for a particular independent variable (e.g., *h* for *Gate*). To determine the survival of a mutant, we determine whether each mutant was detected with the following test oracles [15, 3, 28]:

1. *Wrong Output Oracle (WOO)*: A program's output does not match with the expected output of the program, i.e., a new output is observed. As described in Section 4.1.2, we have two broad categories of algorithms for *Output Dominance*. For *Output-dominant* algorithms, which are expected to produce one dominant output, if the dominant output produced by an algorithm does not match the expected dominant output, we consider it as a mutant detected. For the rest of the algorithms (i.e., *Diverse-output*), if any of the observed outputs do not match the expected outputs, we consider it as a mutant detected;
2. *Output Probability Oracle (OPO)*: The observed outputs match with the expected ones; however, the observed probabilities are significantly different than the expected ones. To compare the expected probabilities with observed probabilities, OPO employs the Chi-square test. We chose a significance level of 0.01, i.e., if the p-value is less than 0.01, we conclude that a mutant is detected. Note that we only consider this oracle for *Diverse-Output* algorithms.

If a mutant is not detected with these two test oracles, we consider the mutant as survived.

RQ3 studies the relations between circuit characteristics (e.g., #qubits) and SR. RQ3 also studies the correlation between an independent variable and survival rate with the Pearson correlation test [64]. Pearson coefficient ranges between -1 and 1, where a value below (or above) 0 indicates a negative (or positive) correlation.

4.4 Subject Systems

In our empirical study, we used a set of real quantum circuits provided by MQT Bench [63][1]. The MQT Bench offers two types of circuits: Non-scalable circuits with fixed numbers of qubits and scalable ones, i.e., the same circuits being implemented with different numbers of qubits. For scalable benchmarks, we configured the range of the number of qubits from 2 to 30 as allowed in MBT Bench. The maximum number of 30 was chosen as this is the maximum number of qubits we could simulate on a classical computer with the IBM simulator for most of the algorithms. Note that setting the maximum qubits to 30 for some circuits resulted in a very complex circuit with many quantum gates that became infeasible to execute on quantum simulators that execute on classical computers. Consequently, for such algorithms, we reduced the number of qubits according to this practical constraint. We obtained in total 382 circuits (21 non-scalable and 361 scalable ones), which are implemented in Open Quantum Assembly Language (QASM [65]) V.2 as the original benchmarks. We further automatically translated them to Qiskit to be compatible with Muskit, which currently can only generate mutants for Qiskit. Table 1 shows the characteristics of the original benchmarks for each quantum algorithm.

[1] We conducted this empirical study with an earlier version of MBT Bench. The current version was released after we had collected all the data.

Table 1: Characteristics of the original benchmarks (i.e., quantum algorithm). ID represents the unique identifier of a quantum algorithm, whereas the bottom of the table shows which ID maps to which algorithm. An interested reader can check [63] for more algorithm details. For each algorithm, #Q, #G, #M, D, #SG, #MQG, and #EQ denote the minimum and maximum number of qubits, gates, measurements, depth, single qubit gates, multi-qubit gates, and entangled qubits.

ID	#Q	#G	#M	D	#SG	#MQG	#EQ	ID	#Q	#G	#M	D	#SG	#MQG	#EQ
QP1	2-20	8-287	2-20	6-114	5-77	3-210	0	QP15	2-25	5-338	2-25	5-51	2-25	3-313	0
QP2	2-30	5-89	1-29	4-32	3-59	2-30	0-22	QP16	2-25	7-363	2-25	7-53	3-26	4-337	2-25
QP3	2-30	3-31	2-30	3-31	1	2-30	2-30	QP17	2-13	6-105	2-13	4-26	4-26	2-79	0
QP4	3-30	7-61	3-30	5-10	3-30	4-31	0	QP18	2-25	5-358	1-24	4-66	3-49	2-309	0
QP5	14	225	14	43	42	183	0	QP19	2-25	5-362	1-24	4-74	3-49	2-313	0
QP6	12	169	12	37	36	133	0	QP20	3-4	26-191	3-4	20-170	13-94	13-97	0-2
QP7	4	25	4	13	12	13	0	QP21	3-13	26-266	3-13	20-230	13-43	13-223	2-2
QP8	2-6	3-451	2-6	2-402	2-98	1-353	0-5	QP22	2-17	12-477	2-17	8-70	8-68	4-409	0
QP9	2-7	3-311	2-7	2-271	2-46	1-265	0-5	QP23	2-12	12-82	2-12	8-20	8-48	4-34	0
QP10	3-11	23-211	6-22	15-47	12-44	11-167	0	QP24	2-16	12-425	2-16	8-66	8-64	4-361	0
QP11	3-17	22-477	3-17	14-70	12-68	10-409	0	QP25	4-16	40-172	4-16	18-30	24-96	16-76	0
QP12	5-15	43-384	5-15	36-343	22-180	21-204	0	QP26	2-17	12-477	2-17	8-70	8-68	4-409	0
QP13	5-15	43-402	5-15	36-347	22-198	21-204	0	QP27	3-19	14-94	3-19	8-24	9-57	5-37	0
QP14	3-15	17-77	6-30	10-12	9-45	8-32	0	QP28	2-30	6-118	2-30	5-61	3-59	3-59	0

*QP1: Amplitude Estimation (ae); QP2: Deutsch-Jozsa (dj); QP3: Greenberger-Horne-Zeilinger State (ghz); QP4: Graph State (graphstate); QP5: Ground State (groundstatelarge); QP6: Ground State (groundstatemedium); QP7: Ground State (groundstatesmall); QP8: Grover Search without Ancilla (grover-noancilla); QP9: Grover Search with Ancilla (grover-v-chain); QP10: Portfolio Optimization with QAOA (portfolioqaoa); QP11: Porfolio Optimization with VQE (portfoliovqe); QP12: Pricing Call Option (pricingcall); QP13: Pricing Put Option (pricingput); QP14: Quantum Approximate Optimization Algorithm (qaoa); QP15: Quantum Fourier Transform (qft); QP16: Quantun Fourier Transform Entangled (qftentangled); QP17: Quantum Generative Adversarial Networks (qgan); QP18: Quantum Phase Estimation Exact (qpeexact); QP19: Quantum Phase Estimation Inexact (qpeinexact); QP20: Quantum Walk without Ancilla (qwalk-noancilla); QP21: Quantum Walk with Ancilla (qwalk-v-chain); QP22: Real Amplitudes ansatz with Random Parameters (realamprandom); QP23: Routing Algorithm (routing); QP24: Efficient SU2 ansatz with Random Parameters (su2random); QP25: Travelling Salesman (tsp); QP26: Two Local ansatz with random parameters (twolocalrandom); QP27: Variational Quantum Eigensolver (vqe); QP28: W-State (wstate).

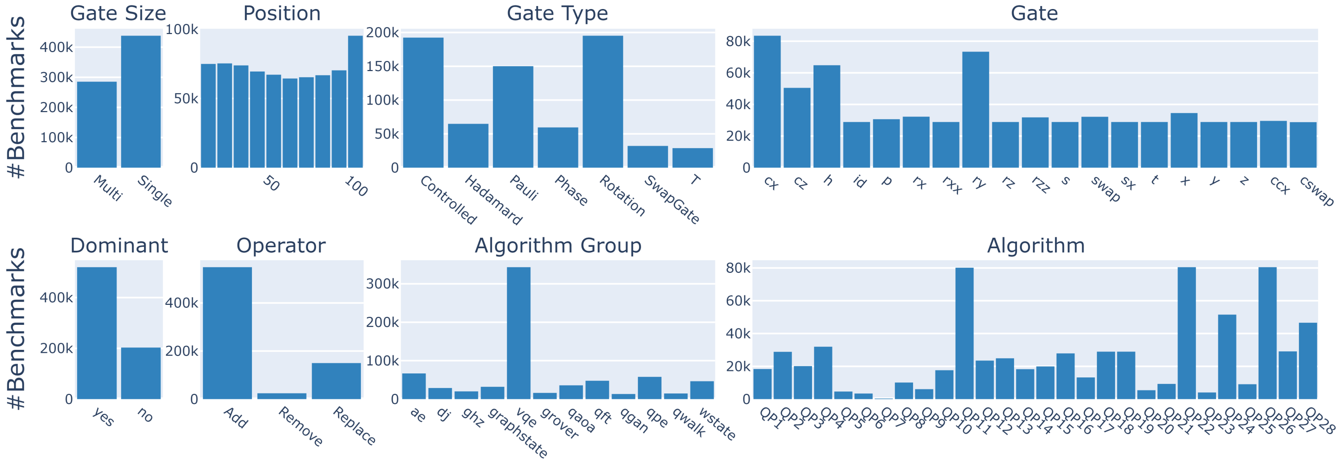


Figure 2: Descriptive statistics of the dataset

4.5 Mutant Generation

We use the Muskit tool [15] to generate faulty benchmarks by applying *Add*, *Remove*, and *Replace* mutation operators. To be comprehensive, for this empirical study, we applied all mutation operators combined with a total of 19 available gate types, on all possible positions, in each original quantum circuit for each quantum algorithm. The add operation is applied in all the possible gaps using all supported gates, the replace operation replaces an existing gate with a new supported gate, and the remove operation will just remove an existing gate. In the end, we obtained 723079 faulty benchmarks, most of which were created with the *Add* operator (75%), followed by *Replace* (20%) and *Remove* (3%). Note that, as mentioned above, the *Replace* and *Remove* operators can only be applied on existing gates of the original quantum circuits while the add operation is applied in the gaps of the circuit, which are always more than the gates; therefore, compared with *Add*, we obtained fewer numbers of faulty benchmarks created with them. Note that we treat all operations as equally relevant, and since we calculate the survival rate for each independent variable, we anticipate that the over-representation of any single operator can be treated equally without biasing the results. Figure 2 presents the descriptive statistics of the generated benchmarks.

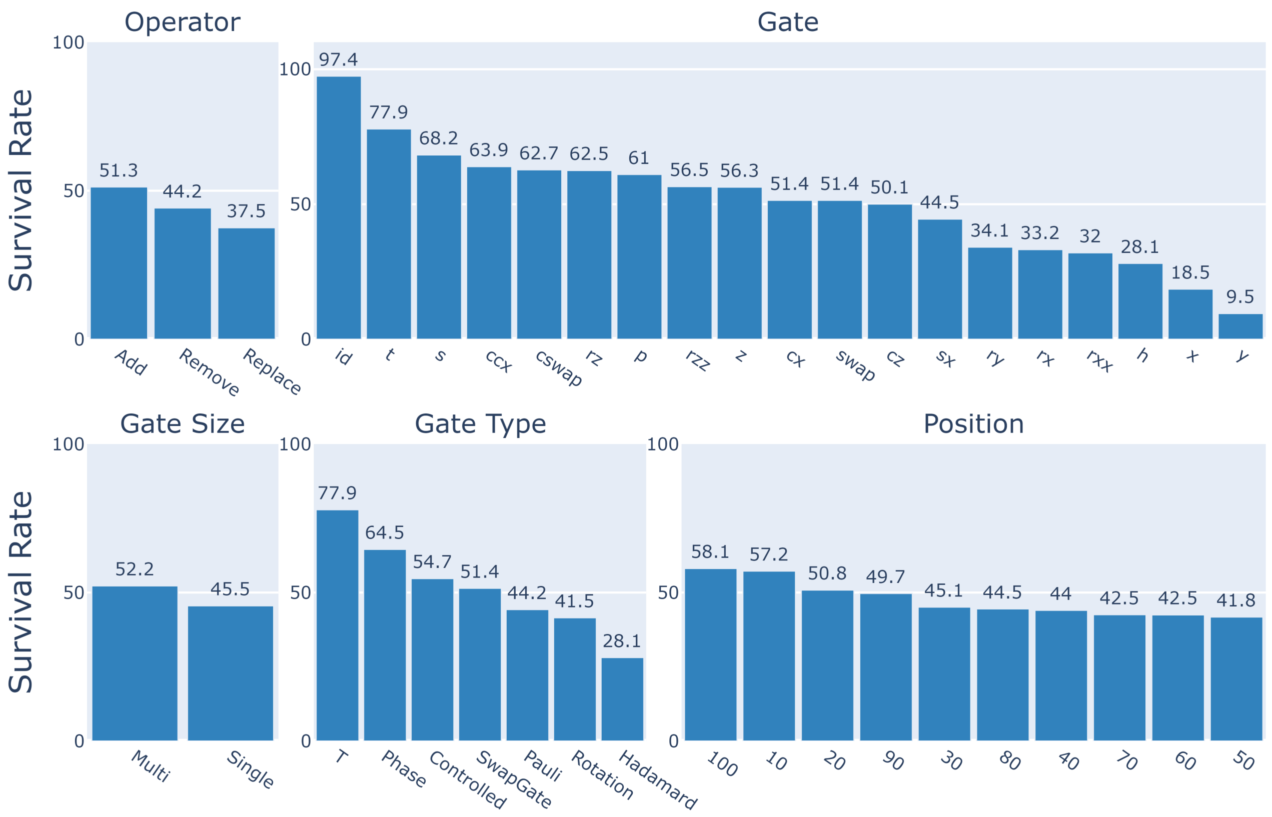


Figure 3: Average SR of all faulty benchmarks in terms of each mutation characteristic – RQ1.1

4.6 Experimental Setup and Execution

All the original and faulty benchmarks were run in the same conditions using the same computational resources. The experiments were executed on a national high-performance cluster of servers, including 2x AMD Epyc 7601 processors, 2TB RAM, an AMD Vega20 GPU, and a high-speed 4TB NVMe drive. We performed a total of 100,000 shots for each circuit to deal with the inherent uncertainty in quantum computing. All the programs were executed using the Qiskit 0.43.1 version's Aer simulator to execute quantum circuits. To ensure consistency and reproducibility, we employed a fixed random seed–a key parameter of the Aer simulator for executing all quantum circuits, to deal with the inherent uncertainty of the quantum circuit execution.

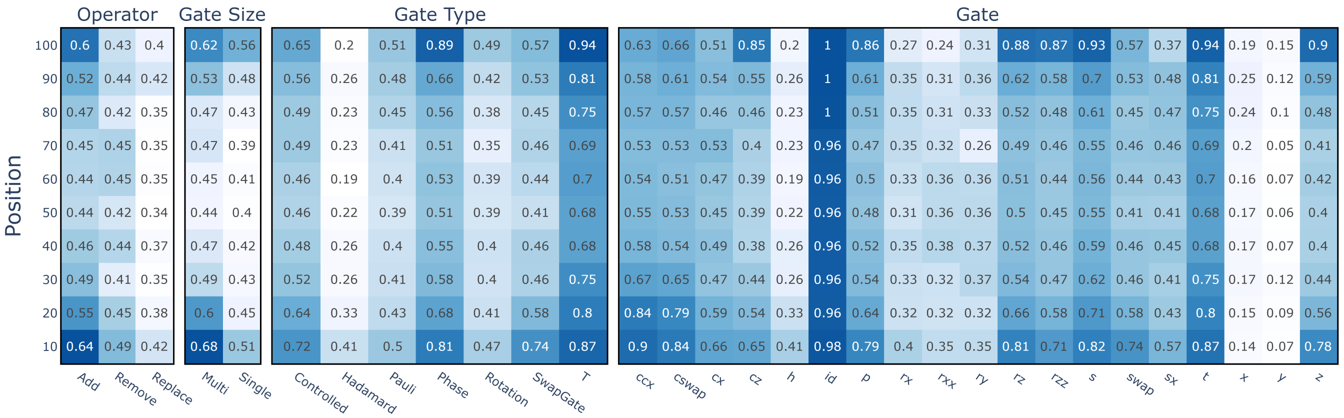
5 Results and Analysis

We present empirical evaluation results. For 382 circuits corresponding to 28 algorithms, we generated a total of 723079 faulty benchmarks. After executing them, we obtained the overall SR of 48%, against the 51% that were detected with test oracle WOO, and the rest with test oracle OPO.

5.1 Results for RQ1 – Analyzing SR by Mutation Characteristics

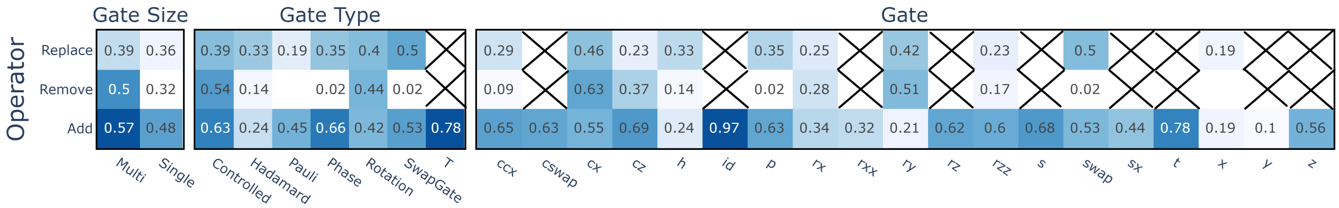
5.1.1 Results for RQ1.1 (individual characteristics)

Figure 3 presents the SR's descriptive statistics in each characteristic across all the faulty benchmarks. For *Operator*, we observe that mutation operator *Add* achieved higher SR (see Figure 3) than *Remove* and *Replace*. This observation provides evidence that the *Add* operator is more likely to generate faulty benchmarks with high chances of surviving, which is often favored for assessing testing methods. For *Position*, we can notice that manipulating faults at the beginning and end of a quantum circuit (100%, 90%, 10%, and 20%) achieved the top four SR (around 50% and above) among the ten categories. As for *Gate Size*, whether being added, deleted, and replaced gate via a mutation operator is a single-qubit gate or multi-qubit gate does not lead to a big difference in SR, i.e., 45.5% and 52.2%, respectively. Regarding characteristic *Gate*, we note that gate *Id* achieved the highest SR. This is expected as *Id*–a single-qubit gate does not change the qubit's state. It is often used for error correction, fault



Position	Operator			Gate Size		Gate Type							Gate																		
	Add	Remove	Replace	Multi	Single	Controlled	Hadamard	Pauli	Phase	Rotation	SwapGate	T	ccx	cswap	cx	cz	h	id	p	rx	rxx	ry	rz	rzz	s	swap	sx	t	x	y	z
100	0.6	0.43	0.4	0.62	0.56	0.65	0.2	0.51	0.89	0.49	0.57	0.94	0.63	0.66	0.51	0.85	0.2	1	0.86	0.27	0.24	0.31	0.88	0.87	0.93	0.57	0.37	0.94	0.19	0.15	0.9
90	0.52	0.44	0.42	0.53	0.48	0.56	0.26	0.48	0.66	0.42	0.53	0.81	0.58	0.61	0.54	0.55	0.26	1	0.61	0.35	0.31	0.36	0.62	0.58	0.7	0.53	0.48	0.81	0.25	0.12	0.59
80	0.47	0.42	0.35	0.47	0.43	0.49	0.23	0.45	0.56	0.38	0.45	0.75	0.57	0.57	0.46	0.46	0.23	1	0.51	0.35	0.31	0.33	0.52	0.48	0.61	0.45	0.47	0.75	0.24	0.1	0.48
70	0.45	0.45	0.35	0.47	0.39	0.49	0.23	0.41	0.51	0.35	0.46	0.69	0.53	0.53	0.53	0.4	0.23	0.96	0.47	0.35	0.32	0.26	0.49	0.46	0.55	0.46	0.46	0.69	0.2	0.05	0.41
60	0.44	0.45	0.35	0.45	0.41	0.46	0.19	0.4	0.53	0.39	0.44	0.7	0.54	0.51	0.47	0.39	0.19	0.96	0.5	0.33	0.36	0.36	0.51	0.44	0.56	0.44	0.43	0.7	0.16	0.07	0.42
50	0.44	0.42	0.34	0.44	0.4	0.46	0.22	0.39	0.51	0.39	0.41	0.68	0.55	0.53	0.45	0.39	0.22	0.96	0.48	0.31	0.36	0.36	0.5	0.45	0.55	0.41	0.41	0.68	0.17	0.06	0.4
40	0.46	0.44	0.37	0.47	0.42	0.48	0.26	0.4	0.55	0.4	0.46	0.68	0.58	0.54	0.49	0.38	0.26	0.96	0.52	0.35	0.38	0.37	0.52	0.46	0.59	0.46	0.45	0.68	0.17	0.07	0.4
30	0.49	0.41	0.35	0.49	0.43	0.52	0.26	0.41	0.58	0.4	0.46	0.75	0.67	0.65	0.47	0.44	0.26	0.96	0.54	0.33	0.32	0.37	0.54	0.47	0.62	0.46	0.41	0.75	0.17	0.12	0.44
20	0.55	0.45	0.38	0.6	0.45	0.64	0.33	0.43	0.68	0.41	0.58	0.8	0.84	0.79	0.59	0.54	0.33	0.96	0.64	0.32	0.32	0.32	0.66	0.58	0.71	0.58	0.43	0.8	0.15	0.09	0.56
10	0.64	0.49	0.42	0.68	0.51	0.72	0.41	0.5	0.81	0.47	0.74	0.87	0.9	0.84	0.66	0.65	0.41	0.98	0.79	0.4	0.35	0.35	0.81	0.71	0.82	0.74	0.57	0.87	0.14	0.07	0.78

Figure 4: Interaction effects between *Position* and all other mutation characteristics – RQ1.2. Each cell shows the SR corresponding to a specific interaction, with a darker (or lighter) blue indicating a higher (or lower) SR.



Operator	Gate Size		Gate Type							Gate																		
	Multi	Single	Controlled	Hadamard	Pauli	Phase	Rotation	SwapGate	T	ccx	cswap	cx	cz	h	id	p	rx	rxx	ry	rz	rzz	s	swap	sx	t	x	y	z
Replace	0.39	0.36	0.39	0.33	0.19	0.35	0.4	0.5	X	0.29	X	0.46	0.23	0.33	X	0.35	0.25	X	0.42	X	0.23	X	0.5	X	X	0.19	X	X
Remove	0.5	0.32	0.54	0.14		0.02	0.44	0.02	X	0.09	X	0.63	0.37	0.14	X	0.02	0.28	X	0.51	X	0.17	X	0.02	X	X		X	X
Add	0.57	0.48	0.63	0.24	0.45	0.66	0.42	0.53	0.78	0.65	0.63	0.55	0.69	0.24	0.97	0.63	0.34	0.32	0.21	0.62	0.6	0.68	0.53	0.44	0.78	0.19	0.1	0.56

Figure 5: Interaction effects between *Operator* and all other mutation characteristics – RQ1.2. Each cell shows SR corresponding to a specific interaction. A darker (or lighter) blue indicates a higher (or lower) SR; a white empty cell denotes an absolute zero SR; a cell with zero in it denotes a very-near-zero positive number; a cell with X tells that no benchmarks can be generated with the given combination.

tolerance, or as a placeholder for maintaining the same circuit depth. However, surprisingly the *id* gate did not achieve a 100%. Our investigation found that the remaining mutants that were killed easily were for *Graphstate* (QP4). This algorithm produces all possible outputs with certain probabilities. Given our chosen number of shots (i.e., 100,000), we could not ensure covering all outputs for its implementation with more than 16 qubits, and therefore, we obtained false positives.

On the other hand, gates *x*, *y*, and *h* achieved the lowest SR since these gates introduce big changes in circuit logic, i.e., *h* introduces superposition, whereas *x* flips the state of a qubit (i.e., $|1\rangle$ to $|0\rangle$ and vice versa) and *y* in addition to state flip, also rotates the phase about the Y axis by π radians. As a result, survival rates were the lowest.

When looking at *Gate Type*, we notice that the *T* gates, *Phase* gates, and *Controlled* gates achieved the top three SR. On the other hand, the *Hadamard* gate achieved the lowest SR. This is because the Hadamard creates superposition, and manipulating a fault with Hadamard changes the logic of a quantum circuit, thereby killing the fault more easily compared with the other gates.

5.1.2 Results for RQ1.2 (pair-wise interactions)

When looking at the impact of the pair-wise interactions of the characteristics of the mutation operators on SR, from Figure 4, the effect of the interactions between *Position* and each circuit characteristic (*Gate*, *Gate Size*, *Gate Type*, and *Operator*) are minor. For instance, the interaction effects between *Position* and gates *x* and *y* can not be noticed since at positions 10%, 20%, 90% and 100%, we cannot observe much gap between the SR at these positions with those at the other positions.

Figure 5 presents the interaction effects of *Operator* with all other characteristics. Note that in some cases (denoted with × in cells), the combinations (e.g., removing a T gate, replacing an id gate) are impossible for given quantum circuits since the original circuits do not contain these gates. Therefore, there were no corresponding faulty benchmarks generated. From the figure, we can observe that, in most cases, the *Add* operator is prominent in leading to high SR, as we have also discussed in RQ1.1. However, there are some exceptions. For instance, adding, removing, and replacing the rotation gates do not differ significantly. Further investigation is needed to make a solid conclusion about this observation, as we lack sufficient data. This is because, as shown in Figure 5, certain gates (e.g., *rxx*, *sx* and *t*) are not removed or replaced due that they are not used in the original circuits selected from MQTBench.

Table 2: Top 5 interactions of mutation characteristics *Operator*, *Gate* (or *Gate Type*, *Gate Size*) and *Position* that achieved the highest SR, e.g., $Add\_id\_80.0\{1.0\}$: adding an *id* gate at position 80% achieved 100% SR– RQ1.3.

Combination	Top 1	Top 2	Top 3	Top 4	Top 5
Operator_Gate_Position	Add_id_80.0{1.0}	Add_id_90.0{1.0}	Add_id_100.0{1.0}	Add_id_70.0{0.98}	Add_id_10.0{0.97}
Operator_Gate Type_Position	Add_T_100.0{0.92}	Add_T_90.0{0.89}	Add_Phase_100.0{0.86}	Add_T_80.0{0.83}	Add_T_10.0{0.83}
Operator_Gate Size_Position	Remove_Multi_100.0{0.71}	Remove_Multi_90.0{0.71}	Add_Multi_10.0{0.69}	Remove_Single_100.0{0.67}	Add_Multi_100.0{0.64}

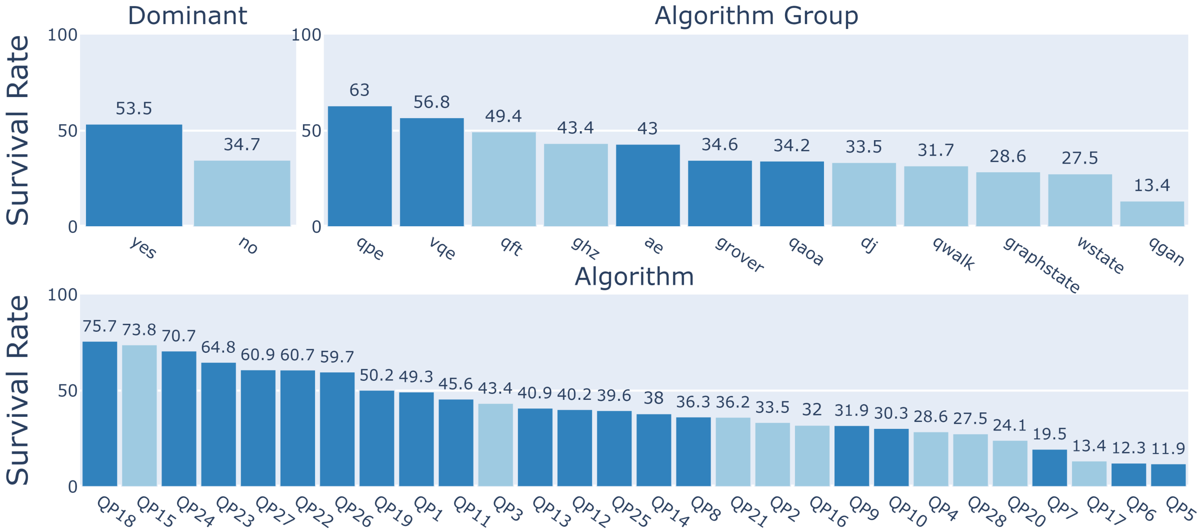


Figure 6: Average SR of all faulty benchmarks regarding algorithms – RQ2. Dark and light blues differentiate output-dominant and diverse-output algorithms.

5.1.3 Results for RQ1.3 (interactions among all characteristics)

We report the top five cases that achieved the highest SR in Table 2, to illustrate the categories' interaction effects. However, all the results are available in the online repository [16]. When looking at the second row of the table, one can observe that adding *id* gate at positions 90%, 100%, 80%, 70% and 60% achieved the top five SR (ranging from 100% to 95%). Regarding the effect of the interactions among *Operator*, *Gate Type* and *Position*, we can observe, from the third row of the table, that adding a *Phase* or *T* gate at position 100% achieved the top two SR, and adding a *Controlled*, *Phase* or *T* gate at position 10% were ranked at the 3rd, 4th and 5th, respectively. Regarding the interaction effects of the combination of *Operator*, *Gate Size* and *Position*, from the results of our study, we recommend adding a multi-gate at position 10% to generate faulty benchmarks that are most challenging, i.e., the highest SR.

Concluding Remarks for RQ1: Applying operator *Add* led to slightly higher SR than *Remove* and *Replace*; introducing faults at the beginning or end of a quantum circuit have a higher chance of generating faulty benchmarks that can survive testing; Gates *x*, *y* and *h* achieved the lowest SR while the *T* and *Phase* gates achieved the highest SR. Minor interaction effects between *Position*/*Operator* and the mutation characteristics can be observed.

5.2 Results for RQ2 – Analyzing SR by Algorithms and their Categorization

5.2.1 Results for RQ2.1 (Output Dominance)

When comparing algorithms regarding the type of output (i.e., *output-dominant* algorithms and *diverse-output* algorithms), we observe that the *output-dominant* algorithms have relatively higher SR (i.e., 53,5%) than the others (i.e., 34.7%), as shown in Figure 6. This tells that faulty benchmarks of the *output-dominant* algorithms are easier to survive. Recall from Section 4.3 that, to check whether a faulty benchmark of an *output-dominant* algorithm is survived, we assess the WOO oracle, i.e., checking whether the observed dominant output is the same as the expected. Therefore, a faulty benchmark of *output-dominant* algorithms does not change the expected dominant output but only results in variation in the expected probability of the dominant output. Thus, the assessment of WOO remains the same, meaning that the benchmark survived. Naturally, one could argue why we also do not assess the observed probability. We argue that an output-dominant algorithm cares the most about producing a dominant output that matches the expected one, less about the probability. Moreover, research is actively ongoing to define and assess test oracles for *output-dominant* quantum algorithms. When such oracles become more available, we can easily integrate them into our studies in the future.

5.2.2 Results for RQ2.2 (Algorithm Group)

When looking into the 12 algorithm groups (see Figure 6), the faulty benchmarks generated for *qpe* and *vqe* achieved the highest SR: 63% and 56.8%, respectively, indicating that they are more tolerant to seeded faults. On the lowest side, *qgan*, *wstate*, and *graphstate* obtained the lowest SR: 13.4%, 27.5% and 28.6%, respectively, indicating that the faulty benchmarks generated from these groups are difficult to survive. As already observed in RQ2.1, in general, *output-dominant* algorithms tend to have higher SR. However, *qft* and *ghz* algorithms are exceptions, as they achieved higher SR than three other *output-dominant* algorithms. For *ghz*, one plausible explanation is that the *ghz* algorithms entangle all qubits in a circuit, and once they are entangled, any mutation operator applied to one qubit will affect the state of all others; when they are measured, they will be all in the same basis state (i.e., all 0 or all 1). This logic seems to reduce the chance of producing incorrect outcomes even when faults are seeded, leading to high SR. As for the *qft* algorithms, they are typically implemented with *CP* and *H* gates. However, *CP* is not one of the gates that the three mutation operators can manipulate, as Muskit currently does not support it. Therefore, no cases exist for removing or replacing *CP* gates in the generated benchmarks for *qft*. Furthermore, as we already observed in RQ1.1, applying *Remove* and *Replace* led to lower SR, as compared to *Add*. Therefore, relying on adding gates probably led to the high SR. However, we need additional experiments to understand *qft* and *ghz* better.

5.2.3 Results for RQ2.3 (individual algorithm, i.e., Algorithm)

For each algorithm, we observe a similar pattern as in RQ2.2, where *output-dominant* algorithms tend to have higher SR. The exceptions are the two *diverse-output* algorithms: QP15 – *qft* (belonging to the *qft* algorithm group) and QP3 – *ghz* (the only algorithm in the *ghz* algorithm group), for which generated faulty benchmarks exhibited higher SR than some of the output-dominant algorithms. Note that we have explained the possible reason in RQ2.2.

When looking at the lowest SR, QP5 – *groundstatemedium* and QP6 –*groundstatelarge* and QP17 – *qgan* obtained the lowest SR: 11.9%, 12.3%, and 13.4% respectively. Interestingly, QP5 and QP6, even belonging to the *output-dominant* algorithm category, still have the lowest SRs. One possible reason is that these two algorithms have a larger number of qubits as compared with QP7 – *groundstatesmall* (Table 1), which all share the same overall structure. This explains that QP5 and QP6 have more possible outputs than QP7; consequently, the dominant output of both is less prominent and, therefore, more sensitive to faults. As for *qgan*, among the *diverse-output* algorithms, it has the lowest SR. This might be because *qgan* has two key components: generator and discriminator. Any changes to the generator possibly influence what the discriminator learns to discriminate real data from newly generated (and changed) data and vice versa. Hence, *qgan* is more sensitive to changes and therefore obtains low SR.

Concluding Remarks for RQ2: Typically, the *dominant-output* algorithms have higher survival rates except for ground state algorithms. For *diverse-output*, in general, we observed low survival rates except for *ghz* and *qft*.

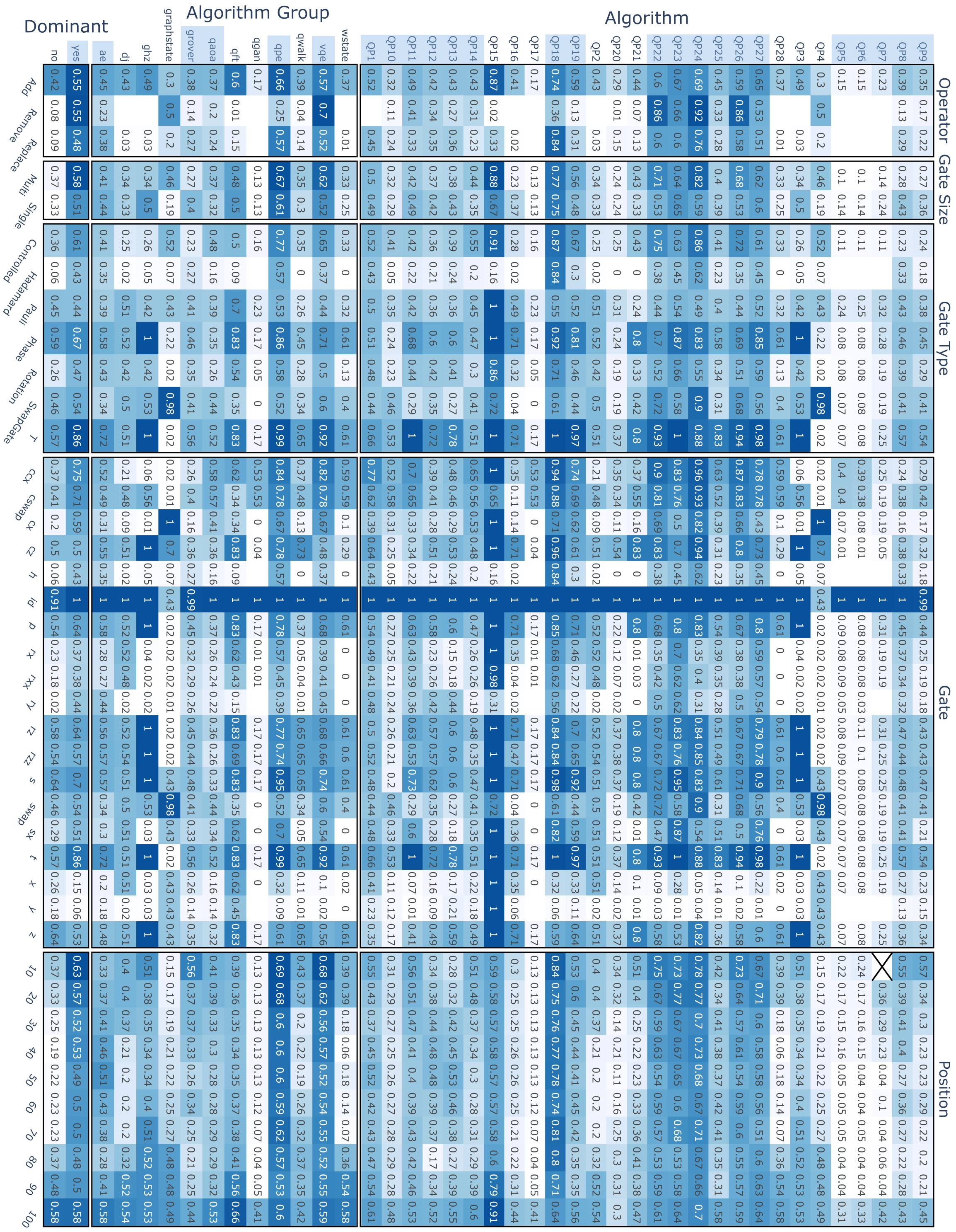
5.3 Results for RQ3 – Analyzing SR by Circuit Characteristics

We studied the relationship between circuit characteristics on the survivability of faulty benchmarks, i.e., the correlation between an independent variable (e.g., number of qubits, i.e., *#qubits*) and its corresponding survival rate with the Pearson correlation test. Results show that all correlation coefficients are between -0.1 and 0.1 for all circuit characteristics, indicating no correlation. This is expected as these characteristics, though they are important metrics for measuring the complexity of circuits, do not fully describe their computational logic, such as how qubits are entangled.

Concluding Remarks for RQ3: No significant correlations can be observed between the circuit characteristics and the survivability of faulty benchmarks.

5.4 Results for RQ4 – Interactions between Algorithm and Mutation Characteristics

All results for RQ4 are presented in Figure 7.



QP1: ae; QP2: dj; QP3: ghz; QP4: graphstate; QP5: groundstatelarge; QP6: groundstatemedium; QP7: groundstatesmall; QP8: grover-noancilla; QP9: grover-v-chain; QP10: portfolioqaoa; QP11: portfoliovqe; QP12: pricingcall; QP13: pricingput; QP14: qaoa; QP15: qft; QP16: qftentangled; QP17: qgan; QP18: qpeexact; QP19: qpeinexact; QP20: qwalk-noancilla; QP21: qwalk-v-chain; QP22: realamprandom; QP23: routing; QP24: su2random; QP25: tsp; QP26: twolocalrandom; QP27: vqe; QP28: wstate

Figure 7: Interaction effects between *Algorithm* (or *Algorithm Group*, *Dominant*) and all mutation characteristics – RQ4. A darker (or lighter) blue indicates a higher (or lower) SR; a white empty cell denotes an absolute zero SR; a cell with zero in it denotes a very-near-zero positive number; a cell with X tells that no benchmarks can be generated with the given combination.

Which quantum circuit mutants shall be used?

5.4.1 Results for RQ4.1 – Interactions with Algorithm

When looking at the interactions of each algorithm with each mutation characteristic, one can notice that for most algorithms, applying the *Add* operator led to the highest SR, as we already observed in RQ1. However, for certain algorithms (e.g., QP26–*twolocalrandom*, QP24–*su2random*, QP22–*realamprandom*, which all belong to *vqe*), the *Remove* operator led to the highest SR, implying that removing a gate from these algorithms has the least impact on their outcomes.

Regarding the interaction of *Algorithm* and *Gate Size*, we observe that, generally, manipulating multi-qubit gates achieves higher SR than manipulating single-qubit gates with a few exceptions. For instance, for QP4–*graphstate*, manipulating multi-qubit gates led to 46% SR, which is notably higher than what manipulating single-qubit achieved: 19%. When looking at the interaction of *Algorithm* and *Gate Type*, manipulating a *Hadamard* gate led to the least SR for most algorithms. However, there are a few exceptions. For instance, for QP18 – *qpeexact*, manipulating a *Hadamard* gate achieved 84% SR, i.e., higher than manipulating a *Pauli*, *Rotation*, or *Swap* gate. One plausible explanation is that the algorithm only has one dominant output, and therefore introducing a fault by manipulating a *Hadamard* gate has a chance of changing the output probabilities but not necessarily altering the dominant output. Manipulating a *T* gate led to the highest SR with a few exceptions, such as QP4 – *graphstate*, for which manipulating a *Swap* gate achieved the highest SR (i.e., 98%). This is because this algorithm produces all possible outputs, so swapping the qubits does not have much effect, as the same state will be reached in another shot.

Regarding specific gates, as discussed in RQ1, *id* led to the 100% SR, with two exceptions on QP9 – *grover-v-chain* and QP4 – *graphstate*. For QP9, after checking the data, we noted that only the two-qubit circuit was affected by *id* because switching the dominant output from one to the other is easy due to similar probabilities. As for QP4 – *graphstate*, as discussed in RQ1.1, it produces all possible outputs, and with the given number of shots, we could not cover all possible outputs.

When comparing Pauli gates: *X*, *Y* and *Z*, *Z* achieved the highest SR for most cases. This might be because *Z* does not change the probabilities of outputs. For QP5 – *groundstatelarge* and QP6 – *groundstatemedium*, the differences in SR across the different gates excluding *id* are very small. We note that for QP15 – *qft*, 14 out of 19 gates achieved 100% SR, implying that the faulty benchmarks for *qft* are very difficult to kill, as already discussed in RQ2.2.

When checking *Position*, we note that introducing faults to the beginning and end of the circuits of most algorithms led to high SR with some exceptions (e.g., QP17, QP18, QP4). For instance, for QP4 – *graphstate*, the SR at position 10% is only 0.15, implying that introducing a fault at the beginning of the circuit of the *graphstate* algorithm is possible to change its behavior.

5.4.2 Results for RQ4.2 (Interaction with Algorithm Group)

Figure 7 shows that the *vqe* and *graphstate* algorithms achieved the highest SR when the *Remove* operator is applied. For *vqe*, removing a gate only changes the probability of a dominant output. In *vqe*, we only care about the correct dominant output, and as long as the correct output remains dominant, it is considered survived.

Regarding *Gate Size*, we only observe a difference in terms of SR between manipulating single-qubit gates and multi-qubit gates for *grover* (40% vs 27%) and *graphstate* (19% vs 46%). As we saw in Section 5.1, multi-qubit, overall, had a higher survival rate than single-qubit, with which the behavior of *graphstate* conforms. However, for *grover*, it is reversed. A possible reason is that *grover* has many multi-qubit gates (Table 1); therefore, removing or replacing multi-qubit gates has a high chance of changing the logic of the circuits, which, hence, led to lower SR than manipulating single-qubit gates.

Manipulating *Hadamard* (or *T*) achieved the lowest (highest) SR for most algorithm types, as also discussed in RQ1. We further observe that SR of *vqe* and *qpe* are higher than those of other algorithms across the different gate types (also discussed in RQ2.2).

For each individual gate, we observe differences in the SR among the same gate types. For instance, for the *ghz* algorithms, manipulating *cz* has a much higher chance of making the faulty *ghz* benchmarks survive when compared with *cx*, i.e., 100% vs 1% SR. This is reasonable because a *cz* gate induces a phase flip, which might not affect the final measurement results, but a *cx* gate introduces both bit and phase flips. When comparing the three Pauli gates: *x*, *y* and *z*, we observe that manipulating a *z* gate led to much higher SR across the algorithm groups. Similarly, this might be because *z* gates do not change the probabilities of outputs.

Regarding *Position*, the *vqe* and *qpe* algorithms are less sensitive to the positions where the faults are seeded. This is because initializing the *vqe* and *qpe* algorithms is not only about applying the Hadamard gate to all qubits. Specifically, a parameterized trial state should be created for a *vqe* algorithm as part of the initialization, and an eigenstate should be prepared for *qpe*. Therefore, they both do not show the pattern we observed about *Position*: seeding a fault at the beginning or end of a circuit is easy to survive (RQ1.2).

5.4.3 Results for RQ4.3 – Interaction with Output Dominance

As expected, results for the *output-dominant* algorithms and *diverse-output* algorithms are quite different. First, for *output-dominant* algorithms (e.g., *qpe*), our results do not reveal much difference in terms of the mutation operators (55% vs 55% vs

48% for *Add*, *Remove* and *Replace*, respectively). Also, the SR of these algorithms' faulty benchmarks is generally higher than the *diverse-output* algorithms. This is reasonable because *output-dominant* algorithms are generally more robust or fault-tolerant of faults, as their working mechanisms amplify the probability of the correct answer (i.e., dominant output). Even with small errors, the correct answer can still be the most likely outcome. Due to this reason, we think manipulating *Hadamard* gates in the *output-dominant* algorithms led to comparable SR as manipulating *Pauli* and *Rotation* gates, and they are not sensitive to where a fault is seeded. This is, however, not the case for *diverse-output* algorithms.

Concluding Remarks for RQ4: The interactions of *Algorithm*, *Algorithm Group*, and *Output Dominance* with the mutation characteristics conform to what we observed in RQ1 and RQ2 with a few exceptions, mostly caused by individual algorithms' unique characteristics. We also observed the importance of distinguishing *output-dominant* and *diverse-output* algorithms in terms of where to seed a fault and which mutation operator to apply.

5.5 Discussion and Recommendations

Availability of Dataset and Recommendation Tool. The database from our empirical evaluation is available for anyone interested in using them.[2]. On top of the faulty benchmarks, we also built a software tool[3], which can recommend faulty benchmarks to users based on selection criteria. For instance, a user can specify an *Algorithm*, *Algorithm Group*, or algorithms by *Output Dominance* together with desired survival rate and a maximum number of faulty benchmarks, and the software tool can provide the faulty benchmarks. This recommendation tool was built by studying all the possible interactions between *Algorithm*, *Algorithm Group*, or algorithms by *Output Dominance* with all possible mutation characteristics (i.e., single, pair-wise, and three-ways).

Assessing Quantum Software Testing Techniques. One typical use of faulty benchmarks is to assess the cost-effectiveness of quantum software testing techniques from various perspectives. For instance, users can assess whether their testing techniques can detect faults seeded in faulty benchmarks of varying survival rates. Moreover, one could also assess whether their testing techniques can identify faults seeded with specific mutation characteristics (e.g., the position of a fault).

Generating New Faulty Benchmarks. We generate new knowledge about relationships between the SR of faulty benchmarks and characteristics of mutations, circuits, algorithms, and their interactions. This knowledge provides evidence about faulty benchmarks with which characteristics likely survive, based on which one can generate new faulty benchmarks for new quantum algorithms, algorithms not yet studied in our empirical study.

Building New Mutation Analysis Techniques. Based on the new knowledge generated from our empirical evaluation, one can develop more advanced quantum mutation analysis techniques. Some examples include developing an optimization approach (e.g., based on search algorithms) to minimize the number of faulty benchmarks to select based on similar survival rates and characteristics.

5.6 Threats to validity

A typical threat to the validity is about the generalization of our results. Note that we pick the real quantum algorithms from the most comprehensive circuit benchmark repository (i.e., MQT Bench). We generated more than 700K mutants from them, thus, making it the largest study on studying quantum mutation characteristics. Nonetheless, one could further enlarge the empirical evaluation with more circuits, which we will pursue in the future.

Concerning the maximum number of qubits, we used a maximum of 30 qubit programs, and the results may be different for circuits with a higher number of qubits. However, it is impossible to execute large qubit circuits on classical computers. Moreover, one could also ask if our results are replicable on real quantum computers since we executed the circuits on the ideal simulator on classical computers. We argue that performing the study on the ideal simulator is important in our context since running experiments on real quantum computers would impact the results due to noise and could potentially lead to invalid conclusions. Nonetheless, we need a dedicated empirical evaluation on noisy quantum computers.

We set the number of shots to 100K, which may be insufficient for large circuits producing all possible outputs. However, most of our algorithms produce dominant outputs; therefore, 100K shots are sufficient. For the OPO test oracle, we chose a significance level of 0.01. Using a lower threshold (e.g., 0.001) may lead to detecting more mutants, thereby decreasing SR. However, 0.01 is commonly used and is a reasonable choice [66, 67, 10]. In the WOO oracle for the *output-dominant* algorithms, we only checked whether the observed dominant output matches the expected one. This could impact survivability. However, there aren't many

[2] Faulty benchmarks can be downloaded from our GitHub repository during the review. If the paper is accepted, we will publish it on a dedicated web page.

[3] Note that the software tool is also available in the GitHub repository.

research results on how to check the correctness of *output-dominant* algorithms, thereby requiring more research on test oracles for such algorithms.

6 Conclusions and Future Work

We presented an empirical study with more than 0.7 million quantum circuit mutants to study how various mutation characteristics, circuit characteristics, algorithm types, and their interactions relate to mutants being undetected (called survivability). Such a study helps to systematically design and generate faulty benchmarks for evaluating quantum software testing techniques' effectiveness from various aspects, e.g., the capability of detecting various types and complexity of faults. Based on the results, we provide actionable recommendations for researchers and practitioners to generate faulty benchmarks. Moreover, we will extend our empirical evaluations with even larger circuits, possibly executing them on real quantum computers, and continuously update the recommendations and the tool.

Acknowledgements

Paolo Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST, Funding Reference number: 10.13039/501100009024 ERATO; and by Engineerable AI Techniques for Practical Applications of High-Quality Machine Learning-based Systems Project (Grant Number JPMJMI20B8), JST-Mirai.

References

[1] M. Coccia, S. Roshani, and M. Mosleh, "Evolution of quantum computing: Theoretical and innovation management implications for emerging quantum industry," *IEEE Transactions on Engineering Management*, pp. 1–11, 2022.

[2] S. Ali, T. Yue, and R. Abreu, "When software engineering meets quantum computing," *Commun. ACM*, vol. 65, no. 4, pp. 84–88, mar 2022. [Online]. Available: https://doi.org/10.1145/3512340

[3] S. Ali, P. Arcaini, X. Wang, and T. 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.

[4] S. Honarvar, M. R. Mousavi, and R. Nagarajan, "Property-based testing of quantum programs in Q#," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 430–435. [Online]. Available: https://doi.org/10.1145/3387940.3391459

[5] D. Fortunato, J. Campos, and R. 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*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 797–800. [Online]. Available: https://doi.org/10.1145/3533767.3543296

[6] X. Wang, T. Yu, P. Arcaini, T. Yue, and S. Ali, "Mutation-based test generation for quantum programs with multi-objective search," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1345–1353. [Online]. Available: https://doi.org/10.1145/3512290.3528869

[7] D. Fortunato, J. Campos, and R. Abreu, "Mutation testing of quantum programs written in QISKit," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 358–359. [Online]. Available: https://doi.org/10.1145/3510454.3528649

[8] ——, "Mutation testing of quantum programs: A case study with Qiskit," *IEEE Transactions on Quantum Engineering*, vol. 3, pp. 1–17, 2022.

[9] X. Wang, P. Arcaini, T. Yue, and S. Ali, "Generating failing test suites for quantum programs with search," in *Search-Based Software Engineering*. Cham: Springer International Publishing, 2021, pp. 9–25.

[10] ——, "Application of combinatorial testing to quantum programs," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 179–188.

[11] P. Zhao, J. Zhao, Z. Miao, and S. Lan, "Bugs4Q: A benchmark of real bugs for quantum programs," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1373–1376.

[12] P. Zhao, Z. Miao, S. Lan, and J. Zhao, "Bugs4Q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs," *Journal of Systems and Software*, vol. 205, p. 111805, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223002005

[13] 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, jun 2021, pp. 28–32. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/Q-SE52541.2021.00013

[14] G. Pontolillo and M. R. Mousavi, "A multi-lingual benchmark for property-based testing of quantum programs," in *Proceedings of the 3rd International Workshop on Quantum Software Engineering*, ser. Q-SE '22. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3528230.3528395

[15] E. Mendiluze, S. Ali, P. Arcaini, and T. Yue, "Muskit: A mutation analysis tool for quantum software testing," pp. 1266–1270, 2022. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678563

[16] E. Mendiluze Usandizaga, T. Yue, P. Arcaini, and S. Ali, "Supplementary materia for the paper "Which Quantum Circuit Mutants Shall Be Used? An Empirical Evaluation of Quantum Circuit Mutations"." [Online]. Available: https://github.com/EnautMendi/Which-Quantum-Circuit-Mutants-Shall-Be-Used

[17] M. Woodward, "Mutation testing—its origin and evolution," *Information and Software Technology*, vol. 35, no. 3, pp. 163–169, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0950584993900536

[18] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Quantitative evaluation of mutation operators for WS-BPEL compositions," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 142–150.

[19] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 354–365. [Online]. Available: https://doi.org/10.1145/2931037.2931040

[20] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 936–946.

[21] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584909000688

[22] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *2010 Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 300–309.

[23] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[24] N. S. Yanofsky and M. A. Mannucci, *Quantum computing for computer scientists*. Cambridge University Press, 2008.

[25] W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned," *Nature*, vol. 299, no. 5886, pp. 802–803, Oct 1982. [Online]. Available: https://doi.org/10.1038/299802a0

[26] H. M. Wiseman and G. J. Milburn, *Quantum measurement and control*. Cambridge university press, 2009.

[27] "IBM Quantum composer." [Online]. Available: https://quantum-computing.ibm.com/

[28] X. Wang, P. Arcaini, T. Yue, and S. Ali, "Quito: A coverage-guided test generator for quantum programs," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, pp. 1237–1241. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678798

[29] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[30] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 275–378. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0065245818300305

[31] P. Anbalagan and T. Xie, "Automated generation of pointcut mutants for testing pointcuts in AspectJ programs," in *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, 2008, pp. 239–248.

[32] J. Boubeta-Puig, I. Medina-Bulo, and A. García-Domínguez, "Analogies and differences between mutation operators for WS-BPEL 2.0 and other languages," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 398–407.

[33] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Quantitative evaluation of mutation operators for WS-BPEL compositions," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 142–150.

[34] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Guided mutation testing for javascript web applications," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 429–444, 2015.

[35] P. Delgado-Pérez, S. Segura, and I. Medina-Bulo, "Assessment of C++ object-oriented mutation operators: A selective mutation approach," *Software Testing, Verification and Reliability*, vol. 27, no. 4-5, p. e1630, 2017.

[36] J. Hu, N. Li, and J. Offutt, "An analysis of OO mutation operators," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 334–341.

[37] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs," in *2008 1st International Conference on Software Testing, Verification, and Validation*, 2008, pp. 52–61.

[38] L. Bottaci, "Type sensitive application of mutation operators for dynamically typed programs," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 126–131.

[39] M. Gligoric, S. Badame, and R. Johnson, "SMutant: A tool for type-sensitive mutation testing in a dynamic language," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 424–427. [Online]. Available: https://doi.org/10.1145/2025113.2025181

[40] A. Alberto, A. Cavalcanti, M. Gaudel, and A. Simão, "Formal mutation testing for Circus," *Information and Software Technology*, vol. 81, pp. 131–153, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491630057X

[41] U. Praphamontripong and J. Offutt, "Applying mutation testing to web applications," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 132–141.

[42] U. Praphamontripong, J. Offutt, L. Deng, and J. Gu, "An experimental evaluation of web mutation operators," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2016, pp. 102–111.

[43] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of Android apps," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[44] M. P. Usaola, G. Rojas, I. Rodríguez, and S. Hernández, "An architecture for the development of mutation operators," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 143–148.

[45] R. A. Oliveira, E. Alégroth, Z. Gao, and A. Memon, "Definition and evaluation of mutation operators for GUI-level mutation analysis," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[46] F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke, "Memory mutation testing," *Information and Software Technology*, vol. 81, pp. 97–111, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916300362

[47] M. E. Delamaro, J. Offutt, and P. Ammann, "Designing deletion mutation operators," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 11–20.

[48] R. Gopinath and E. Walkingshaw, "How good are your types? using mutation analysis to evaluate the effectiveness of type annotations," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 122–127.

[49] P. Arcaini, A. Gargantini, and E. Riccobene, "MutRex: A mutation-based generator of fault detecting strings for regular expressions," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 87–96.

[50] ——, "Fault-based test generation for regular expressions by mutation," *Software Testing, Verification and Reliability*, vol. 29, no. 1-2, p. e1664, 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1664

[51] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 435–444. [Online]. Available: https://doi.org/10.1145/1806799.1806863

[52] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "How hard does mutation analysis have to be, anyway?" in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 216–227.

[53] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 351–360. [Online]. Available: https://doi.org/10.1145/1368088.1368136

[54] M. E. Delamaro, L. Deng, V. H. S. Durelli, N. Li, and J. Offutt, "Experimental evaluation of SDL and one-op mutation for C," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 203–212.

[55] V. H. S. Durelli, N. M. De Souza, and M. E. Delamaro, "Are deletion mutants easier to identify manually?" in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 149–158.

[56] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 919–930. [Online]. Available: https://doi.org/10.1145/2568225.2568265

[57] B. H. Smith and L. Williams, "An empirical evaluation of the MuJava mutation operators," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 193–202.

[58] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: A mutation system for Java," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 827–830. [Online]. Available: https://doi.org/10.1145/1134285.1134425

[59] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. 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*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 654–665. [Online]. Available: https://doi.org/10.1145/2635868.2635929

[60] H. Zhang, T. Yue, S. Ali, and C. Liu, "Towards mutation analysis for use cases," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 363–373. [Online]. Available: https://doi.org/10.1145/2976767.2976784

[61] P. Zhao, J. Zhao, and L. Ma, "Identifying bug patterns in quantum programs," in *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2021, pp. 16–21. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/Q-SE52541.2021.00011

[62] J. L. Weaver and F. J. Harkins, *Qiskit Pocket Guide*. O'Reilly Media, Inc., 2022.

[63] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT Bench: Benchmarking software and design automation tools for quantum computing," *Quantum*, 2023, MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.

[64] T. J. Cleophas and A. H. Zwinderman, *Bayesian Pearson Correlation Analysis*. Cham: Springer International Publishing, 2018, pp. 111–118. [Online]. Available: https://doi.org/10.1007/978-3-319-92747-3_11

[65] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," 2017.

[66] J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "Evolutionary mutation testing," *Information and Software Technology*, vol. 53, no. 10, pp. 1108–1123, 2011, special Section on Mutation Testing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491100084X

[67] G. Mogos, "Quantum random number generator vs. random number generator," in *2016 International Conference on Communications (COMM)*, 2016, pp. 423–426.