Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Metamorphic Testing of Quantum Query Algorithms

Sinhué García-Gil

sinugarc@ucm.es

Universidad Complutense de Madrid

Luis Llana

Universidad Complutense de Madrid

José Ignacio Requeno Jarabo

Universidad Complutense de Madrid

Research Article

Keywords: Quantum computing, metamorphic testing, input reduction, mutation testing, quantum software engineering

Posted Date: November 12th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11469178/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Metamorphic Testing of Quantum Query Algorithms

Sinhué García-Gil[1*], Luis Llana[1] and José Ignacio Requeno Jarabo[1]

[1*]Universidad Complutense de Madrid, Madrid, España.

*Corresponding author(s). E-mail(s): sinugarc@ucm.es;
Contributing authors: llana@ucm.es; jrequeno@ucm.es;

Abstract

Testing quantum programs remains challenging due to probabilistic measurement and the difficulty of establishing reliable test oracles. Although metamorphic testing, MT, has been explored in quantum software, its fault-detection effectiveness and input requirements remain insufficiently investigated. To address these challenges, we propose six metamorphic relations, MRs, for three quantum query algorithms: Deutsch-Jozsa, Bernstein-Vazirani, and Simon. We introduce a testing structure that translates these relations into executable tests and integrates with QCRMut to automate mutant generation, test execution, and evaluation. Some MRs also allow checks before measurement, minimising dependence on statistical analysis of measurement outcomes.

We systematically evaluate the validity and fault-detection effectiveness of the proposed MRs through three experiments. Exhaustive evaluation confirms their validity for the selected configurations, with mutation scores ranging from 41.80% to 83.59%. We then investigate input reduction across 20 random selections per configuration, finding that all selections reproduce their exhaustive mutation scores within 15 inputs. Finally, we examine larger configurations without exhaustive reference scores, observing mutation-score stabilisation within the evaluated range. These findings demonstrate the fault-detection capability of the proposed MRs and show that reduced input sets can preserve this effectiveness, supporting MT as a promising approach when exhaustive evaluation becomes computationally impractical.

Keywords: Quantum computing, metamorphic testing, input reduction, mutation testing, quantum software engineering.

1 Introduction

Quantum computing is a computational paradigm based on the principles of quantum mechanics, in which information is represented by quantum bits, or qubits, rather than the classical bits used in conventional computing. Through properties such as superposition and entanglement, quantum systems can provide computational advantages for certain problems. Quantum computations are represented through quantum circuits, consisting of sequences of quantum gates that manipulate qubits and evolve their quantum states, together with measurement that produce classical outcomes.

Testing is a critical process in software development aimed at increasing confidence in the functional correctness of a program or application and its compliance with specified requirements. Dynamic testing involves executing software with the aim of detecting faults and checking whether it behaves as expected under different conditions. Two well-established testing techniques from classical computing that are considered in this work are mutation testing and MT.

Mutation testing is a technique in which small changes are introduced into a program or circuit to represent potential faults. A test suite is then executed to determine whether it can detect these mutants. Considerable research has been conducted in this field over the years, as surveyed by Papadakis et al. [1].

MT uses MRs, which define properties of the program under test by establishing relationships between inputs and their corresponding outputs. It is particularly useful when a reliable test oracle is unavailable. Chen et al. [2] provided a comprehensive discussion of MT, its capabilities, and its challenges. MT has since been applied to many different areas of classical computing. For example, Almendros-Jiménez et al. [3] applied MT to OpenStreetMap, while Méndez et al. [4] used MT to test chess engines.

Despite recent advances in quantum software testing, two challenges remain particularly relevant to this work. First, the probabilistic nature of quantum measurement can require repeated executions and statistical tests to assess program behaviour, increasing the cost of testing. Second, although MT has shown promise in addressing the oracle problem, further evaluation of its fault-detection capability across different quantum algorithms using automatically generated mutants is needed.

In this work, we investigate whether part of the testing process can be moved before measurement, allowing certain properties considered by our MRs to be checked through deterministic operations within the quantum circuit, thereby minimising dependence on the statistical analysis of measurement outcomes. However, this does not imply that the measurement results themselves are deterministic. To this end, we define six MRs for three quantum algorithms with function-based inputs: Deutsch-Jozsa, Bernstein-Vazirani, and Simon. We integrate MT with mutation testing and use QCRMut to automatically generate structured mutants and systematically assess the fault-detection capability of the proposed MRs. Furthermore, we investigate whether this effectiveness can be preserved using a reduced number of inputs and whether mutation scores stabilise for larger configurations without an exhaustive reference. Our objective is therefore not only to address the oracle problem through MT, but also to evaluate the effectiveness of the proposed MRs in detecting faults and the potential for reducing the number of inputs required.

The remainder of the paper is structured as follows. Section 2 reviews the current state of the art, outlines the starting point of our research, and presents the algorithms considered in this work. Section 3 introduces our MRs, followed by Section 4, which describes their integration into our testing framework. Section 5 presents the experiments and results used to evaluate the effectiveness of the proposed MRs. Finally, Section 6 concludes the paper.

2 Related work

In this section, we review the main testing techniques introduced for quantum computing. We first discuss approaches based on assertions, coverage criteria, and program analysis. We then examine mutation testing, focusing on the tools developed for automatic mutant generation, followed by existing applications of MT to quantum software. Finally, we discuss the limitations of these approaches and identify the main challenges motivating our work. Hoare logic [5] and assertions were among the first approaches explored by researchers to develop testing techniques for quantum programs. Huang et al. [6], for example, proposed the use of assertions as a debugging mechanism for quantum programs. This idea was later extended to runtime assertions. Zhou et al. [7] proposed dynamic assertions based on the introduction of ancilla qubits and later expanded this work with further experiments [8]. Li et al. proposed Proq [9], a projection-based runtime assertion scheme in which projection operators are used to express predicates, providing flexibility in the placement of assertions.

These works influenced the development of QSharpCheck by Honarvar et al. [10], a property-based testing framework tailored to $Q\#$. Another line of research has investigated input-output coverage criteria. Ali et al. [11, 12] introduced the Quito approach, which defines three coverage criteria according to their focus: input, output, and input-output coverage. Their work formally defines concepts such as quantum programs, specifications, inputs, and valid output values. Combinatorial testing was explored by Wang et al. [13] through QuCAT, which generates test suites using classical generation algorithms and evaluates their effectiveness against random test-suite generation.

Other approaches have addressed debugging and program analysis. Metwalli et al. [14] introduced a debugging approach based on slicing quantum programs so that individual slices can be tested separately. Further approaches include the use of automata by Chen et al. [15] and static analysis of quantum programs [16, 17]. Broader overviews of quantum software testing can be found in surveys by Shang et al. [18], Serrano et al. [19], García et al. [20], and Leite et al. [21].

Mutation testing is a fault-based testing technique in which small modifications, or mutations, are introduced into a program to create mutated versions. These mutants are then executed against a test suite to determine whether the introduced faults can be detected, thereby providing a means of evaluating the effectiveness of the test cases. In quantum computing, mutation testing has been adapted to account for the particular characteristics of quantum programs. Some of the approaches discussed above [10, 12] use mutation testing as an evaluation mechanism to address their hypotheses and research questions.

Wang et al. [22] introduced quantum mutation operators to assess quantum computing platforms rather than individual quantum programs, identifying bugs and behavioural differences between platforms. The authors later investigated equivalent mutants using mutation testing [23]. They define a discount factor based on the number of previous tests that have failed to kill a mutant, which is used to estimate the likelihood of the mutant being equivalent.

Automatic mutant generation for quantum programs has been addressed by three tools: Muskit, proposed by Mendiluze et al. [24]; QMutPy, proposed by Fortunato et al. [25]; and QCRMut, proposed by García et al. [26]. These tools target Qiskit, an SDK for quantum computing developed by IBM. Muskit operates at source-code level, analysing the program and exhaustively generating the applicable mutations for each line and insertion location. The generated mutants are then executed and their outputs are assessed using an oracle, following criteria developed in previous work [11, 12, 22].

QMutPy extends the MutPy library with quantum mutation operators and a notion of gate equivalence, operating on Python's abstract syntax tree. It executes and analyses the generated mutants using assertions rather than requiring an explicit output oracle. Its evaluation also considers modifications to these assertions to study their effect on mutation scores. QCRMut, in contrast, introduces a random mutant-generation approach that operates directly on Qiskit's high-level circuit representation. Unlike Muskit and QMutPy, which were developed for legacy Qiskit 0.x versions, QCRMut supports Qiskit 2.x. Among its features, QCRMut allows selected circuit instructions to remain immutable during mutation, thereby preserving parts of the original circuit structure.

MT [2] is a testing approach commonly used when a reliable test oracle is unavailable, i.e., when determining the expected outcome of a test case is difficult. It relies on MRs, which express necessary properties of the intended behaviour of the software by defining relationships between source and follow-up test cases and their corresponding outputs. This makes MT particularly relevant to quantum software, where the oracle problem is further complicated by the probabilistic nature of quantum measurement. Assessing measured outputs may require repeated executions and statistical analysis to determine whether the observed behaviour is consistent with the expected behaviour.

MorphQ, presented by Paltenghi et al. [27], introduced MT for quantum software systems. MorphQ uses grammar-based approaches to generate syntactically correct quantum programs and addresses the oracle problem by checking whether the defined MRs hold. Other studies have explored MT for specific quantum programs. Abreu et al. [28] present a quantum adder together with two MRs based on a subset of Peano's axioms. Costa et al. [29] apply MT to Shor's algorithm, using MRs to test its behaviour without requiring the complete computation, which could not be completed on the quantum hardware available at the time.

These studies demonstrate the applicability of MT to quantum software, but its exploration across quantum algorithms with different properties remains limited. Existing studies focus on particular algorithms or specific types of MRs, providing limited evidence of how MT can be applied to quantum programs exhibiting different characteristics. Moreover, the fault-detection capability of the proposed MRs has not been systematically assessed using automatically generated mutants. Extending the

study of MT to algorithms with different properties is therefore important to better understand its applicability and limitations within quantum software testing.

Our work contributes to this direction by defining six MRs for three quantum query algorithms: Deutsch-Jozsa, Bernstein-Vazirani, and Simon. As in classical MT, although the testing approach itself can be applied generally, there is no general framework for automatically deriving MRs, as their definition depends on the specific properties and expected behaviour of the algorithm under test. We derive and assess the effectiveness of these MRs through mutation testing, using QCRMut to automatically generate mutants and systematically evaluate their fault-detection capability. QCRMut's ability to preserve selected circuit instructions during mutation is particularly relevant to our approach, as it allows the parts of the circuits representing the algorithm inputs to remain unchanged, preserving the controlled relationships required by the proposed MRs. In addition, some of these MRs allow properties to be verified before measurement, minimising dependence on the statistical analysis of measurement outcomes.

To conclude this section, we provide a brief presentation of the selected algorithms: Deutsch-Jozsa, Bernstein-Vazirani, and Simon. Each algorithm is presented following a consistent structure: first, we introduce the problem addressed by the algorithm; then, we explain how the quantum algorithm solves it and how its result is obtained.

- *Deutsch-Jozsa* algorithm, DJ, was proposed by David Deutsch and Richard Jozsa in 1992 [30]. This algorithm aims to solve the following problem.
  Problem: Let $f : \{0,1\}^n \to \{0,1\}$ be a function with $n \in \mathbb{N}$, $n > 1$ that has one of the following properties:
  - We say f is **constant** if all inputs map to either 0 or 1.
  - We say f is **balanced** if half of the inputs map to 0 and the remainder to 1.

  DJ algorithm determines whether a given function, f, is balanced or constant. Let us examine the possible outcomes depending on the function's property:
  - **Constant**: The outcome can only be $|\mathbf{0}\rangle$.
  - **Balanced**: The outcome can be anything except $|\mathbf{0}\rangle$.

  It is important to note that if the function is neither balanced nor constant, the algorithm's output will not be interpretable. Therefore, this is a precondition for f.
- *Bernstein-Vazirani* algorithm, BV, was presented by Ethan Bernstein and Umesh Vazirani in 1997 [31].
  Problem: Let $\mathbf{s} = s_0 s_1 \dots s_{n-1} \in \{0,1\}^n$ be an unknown bit string, with $n \in \mathbb{N}$ and $n > 1$, and let $f : \{0,1\}^n \to \{0,1\}$ be a function defined as $f(\mathbf{x}) = \mathbf{s} \cdot \mathbf{x} \pmod 2 = s_0 x_0 \oplus s_1 x_1 \oplus \dots \oplus s_{n-1} x_{n-1}$.
  The algorithm determines $\mathbf{s}$, which is the only possible measurement outcome under ideal execution.
- *Simon's* algorithm [32] combines quantum and classical processing; however, we focus only on the quantum routine. The algorithm was presented by Daniel Simon in 1997 [33].

Problem: Let $f : \{0,1\}^n \to \{0,1\}^n$, with $n \in \mathbb{N}$ and $n > 1$, be a function with an unknown non-zero period $\mathbf{c} = c_0 c_1 \ldots c_{n-1} \in \{0,1\}^n$, such that $f(\mathbf{x}) = f(\mathbf{x} \oplus \mathbf{c})$ for all $\mathbf{x} \in \{0,1\}^n$.
The objective is to determine the period $\mathbf{c}$. Simon's quantum routine produces a string $\mathbf{z}$ satisfying $\langle \mathbf{z}, \mathbf{c} \rangle = 0 \pmod 2$. By repeating the quantum routine, sufficient information can be obtained to determine $\mathbf{c}$ through classical processing, which involves solving a system of linear equations.

3 Metamorphic Testing Rules and Framework

Having reviewed existing approaches to quantum software testing and introduced the algorithms considered in this work, we formulate the research questions that guide the development and evaluation of our proposed approach:

RQ1: How effective are the proposed MRs at detecting faults in the selected quantum programs? We evaluate the fault-detection capability of the six proposed MRs across the three selected algorithms. Mutation score is used as the primary measure of effectiveness, considering the complete input space for the experimental configurations included in this analysis.
RQ2: Can a reduced set of inputs preserve the fault-detection effectiveness achieved using the complete input space? For configurations where exhaustive evaluation is performed, the resulting mutation scores serve as reference values. Smaller sets of inputs are generated using different random seeds and compared against these exhaustive results.

RQ2.1: How many inputs are required to achieve the mutation score obtained using the complete input space? We examine the evolution of the mutation score as inputs are added to determine the minimum number required to reach the exhaustive reference score.
RQ2.2: How consistent is the required number of inputs across different input selections? Multiple random input selections are considered to assess the variability in the number of inputs required to reach the reference mutation score.

RQ3: Does the mutation score stabilise with a limited number of inputs for larger quantum programs? As the input space grows, the computational cost of exhaustive evaluation increases, eventually making it impractical. We therefore investigate whether mutation scores stabilise as the number of selected inputs increases, and whether this behaviour is consistent across different random input selections without an exhaustive reference score.

Together, these research questions guide the evaluation of the proposed MRs, from their fault-detection effectiveness, RQ1, to the possibility of preserving this effectiveness with fewer input, RQ2, and the stabilisation of mutation scores for larger configurations without an exhaustive reference, RQ3. With these objectives established, we introduce the MRs considered in this work.

3.1 Metamorphic Rules

Before presenting the MRs for each algorithm, we define the MR structure used in this work, including the input and output spaces. We denote $\mathcal{I} = \{0,1\}^n$, where $n \in \mathbb{N}$.

Definition 1 (DJ algorithm input space) The input space of the DJ algorithm, denoted by $\mathcal{I}_{\mathrm{DJ}}$ for $n \in \mathbb{N}$, is defined as:

$$\mathcal{I}_{\mathrm{DJ}} = \{i : \mathcal{I} \to \{0,1\} \mid i \text{ satisfies the DJ precondition}\}$$

Definition 2 (BV algorithm input space) The input space of the BV algorithm, denoted by $\mathcal{I}_{\mathrm{BV}}$ for $n \in \mathbb{N}$, is defined as:

$$\mathcal{I}_{\mathrm{BV}} = \{i : \mathcal{I} \to \{0,1\} \mid i(\mathbf{x}) = \mathbf{s} \cdot \mathbf{x} \bmod 2, \mathbf{s} \in \mathcal{I}\}$$

Definition 3 (Simon algorithm input space) The input space of the Simon algorithm, denoted by $\mathcal{I}_{\mathrm{S}}$ for $n \in \mathbb{N}$, is defined as:

$$\mathcal{I}_{\mathrm{S}} = \{i : \mathcal{I} \to \mathcal{I} \mid i \text{ has period } \mathbf{c}, \text{ where } \mathbf{c} \in \mathcal{I}\}$$

Definition 4 (Output space) The output space of our quantum circuits, denoted by $\mathcal{O}$, is defined as the set of all possible observed results, independent of their probabilities: $\mathcal{O} = \mathcal{P}(\{0,1\}^n)$, where $n \in \mathbb{N}$ is the number of measured qubits.

Definition 5 ($\mathrm{MR_A}$) Given an implementation P of an algorithm A, an output space $\mathcal{O}$, and an input space $\mathcal{I}_A$, a metamorphic rule $\mathrm{MR}_A \subseteq \left(\mathcal{I}_A^m \times \mathcal{O}^m\right)$ is a relation over m inputs $i_0, \ldots, i_{m-1}$ and their corresponding outputs $P(i_0), \ldots, P(i_{m-1})$. We write $\mathrm{MR}_A(i_0, \ldots, i_{m-1}, P(i_0), \ldots, P(i_{m-1}))$ to denote $(i_0, \ldots, i_{m-1}, P(i_0), \ldots, P(i_{m-1})) \in \mathrm{MR}_A$.

Given inputs $i_0, \ldots, i_{m-1} \in \mathcal{I}_A$ and their corresponding outputs $o_0 = P(i_0), \ldots, o_{m-1} = P(i_{m-1})$, we define a specific metamorphic rule MR_A^k using the following pattern, where Φ is a first-order logic formula with no free variables:

$$\mathrm{MR}_A^k(i_0, ..., i_{m-1}, o_0, ..., o_{m-1}) \iff \Phi(i_0, ..., i_{m-1}, o_0, ..., o_{m-1})$$

We now present the proposed MRs for each algorithm.

Deutsch-Jozsa. The soundness of these MRs follows directly from the DJ problem precondition. A summary of all MRs is provided in Table 1. For brevity, we refer to these rules as DJ MR1 and DJ MR2, and use analogous notation for the remaining rules.

Definition 6 ($\mathrm{MR}_{\mathrm{DJ}}^1$) Let $g : \{0,1\}^n \to \{0,1\}^n$ be a bijection, $i_0 \in \mathcal{I}_{\mathrm{DJ}}$, and $i_1 = i_0 \circ g$;

$$\mathrm{MR}_{\mathrm{DJ}}^1(i_0, i_1, o_0, o_1) \iff \left(o_0 = \{|\mathbf{0}\rangle\} \iff o_1 = \{|\mathbf{0}\rangle\}\right)$$

Definition 7 ($\mathrm{MR}_{\mathrm{DJ}}^2$) Let $i_0 \in \mathcal{I}_{\mathrm{DJ}}$, and $i_1(x) = 1 - i_0(x), \ \forall x \in \mathcal{I}$;

$$\mathrm{MR}_{\mathrm{DJ}}^2(i_0, i_1, o_0, o_1) \iff \left(o_0 = \{|\mathbf{0}\rangle\} \iff o_1 = \{|\mathbf{0}\rangle\}\right)$$

Bernstein-Vazirani. In contrast to the Deutsch-Jozsa MRs, which follow from the problem definition, the soundness of the BV MRs is derived from the algebraic properties of the BV oracle; in particular, MR3 relies on successive application of the oracles. We are going to denote by $\mathbf{s_j} \in \mathcal{I}$, the string that determines the input $i_j \in \mathcal{I}_{\mathrm{BV}}$.

Definition 8 ($\mathrm{MR}^1_{\mathrm{BV}}$) Let $i_0, i_1 \in \mathcal{I}_{\mathrm{BV}}$, $\mathbf{s_2} = \mathbf{s_0} \oplus \mathbf{s_1}$ denoted as $i_2 = i_0 \oplus i_1$;

$$\mathrm{MR}^1_{\mathrm{BV}}(i_0, i_1, i_2, o_0, o_1, o_2) \iff o_2 = o_0 \oplus o_1$$

Definition 9 ($\mathrm{MR}^2_{\mathrm{BV}}$) Let $i_0 \in \mathcal{I}_{\mathrm{BV}}$, and let $\mathbf{s_1}$ be the bitwise complement of $\mathbf{s_0}$;

$$\mathrm{MR}^2_{\mathrm{BV}}(i_0, i_1, o_0, o_1) \iff \{|\mathbf{1}\rangle\} = o_0 \oplus o_1$$

Definition 10 ($\mathrm{MR}^3_{\mathrm{BV}}$) Let $i_0, i_1 \in \mathcal{I}_{\mathrm{BV}}$, $i_2 = i_0 \cdot i_1$, where $\cdot$ represents the successive application of the oracles;

$$\mathrm{MR}^3_{\mathrm{BV}}(i_0, i_1, i_2, o_0, o_1, o_2) \iff o_2 = o_0 \oplus o_1$$

MR1 and MR2 are closely related, as MR2 is a special case of MR1 and could therefore be omitted. However, as discussed in the literature on MT [2], simpler relations can still be useful because they are generally easier to verify. For example, from a general relation $f(k \times x) = k \times f(x)$, one may derive the simpler case $k = -1$. Similarly, MR2 requires only two executions of P, whereas MR1 requires three, reducing the computational cost of testing. Furthermore, MR2 considers $|\mathcal{I}|$ possible inputs, whereas MR1 considers $|\mathcal{I}|^2$ input combinations, which can substantially increase the number of tests required.

Simon. Next, we introduce Simon's MR:

Definition 11 Let $\overline{(\cdot)}$ denote the unary operator that:

- When applied to a bit string, inverts the string.
- When applied to a set of bit strings, returns the set obtained by inverting each element.

Definition 12 ($\mathrm{MR_S}$) Let $i_0 \in \mathcal{I}_{\mathrm{S}}$, $\mathbf{c_0} \in \mathcal{I}$ the period of i_0, and $\mathbf{c_1} = \overline{\mathbf{c_0}}$ denoted as $i_1 = \overline{i_0}$;

$$\mathrm{MR_S}(i_0, i_1, o_0, o_1) \iff o_0 = \overline{o_1}$$

In this case, the rule follows from the semantics of the problem. If the set of solutions produced by the algorithm represents a system of equations, then any permutation of the variables must be mirrored by a corresponding permutation of the solution, here, the input. In particular, the permutation that defines $\mathrm{MR_S}$ is provided by the operator $\overline{(\cdot)}$.

A summary of all algorithm MRs is given in Table 1. These rules cannot be generalised to arbitrary quantum algorithms, as they are inherently specific and tailored

R	Inputs	$R(i_0, ..., i_m, o_0, ..., o_m)$
MR^1_{DJ}	$i_0 \in \mathcal{I}_{DJ},\ i_1 := i_0 \circ g,$ $g \in \mathrm{Sym}(\{0,1\}^n)$	$o_0 = \{\|\mathbf{0}\rangle\} \iff o_1 = \{\|\mathbf{0}\rangle\}$
MR^2_{DJ}	$i_0 \in \mathcal{I}_{DJ},\ i_1 := 1 \oplus i_0$	$o_0 = \{\|\mathbf{0}\rangle\} \iff o_1 = \{\|\mathbf{0}\rangle\}$
MR^1_{BV}	$i_0, i_1 \in \mathcal{I}_{BV},\ i_2 := i_0 \oplus i_1$	$o_2 = o_0 \oplus o_1$
MR^2_{BV}	$i_0 \in \mathcal{I}_{BV},\ i_1 := i_0 \oplus \|\mathbf{1}\rangle$	$\{\|\mathbf{1}\rangle\} = o_0 \oplus o_1$
MR^3_{BV}	$i_0, i_1 \in \mathcal{I}_{BV},\ i_2 := i_0 \cdot i_1$	$o_2 = o_0 \oplus o_1$
MR^1_{S}	$i_0 \in \mathcal{I}_{S},\ i_1 := \overline{i_0}$	$o_0 = \overline{o_1}$

Table 1 MRs summary.

to the algorithm under study. This reflects a well-known limitation of MT in classical computing, which persists in the quantum setting. Nevertheless, as a general guideline, the problem semantics and a sound understanding of the circuit's operational behaviour should provide insight into logical properties that are expected to hold. In our case, the rules are derived from the problem preconditions, the mathematical formulation of the algorithm, and the underlying problem semantics. The proposed MRs have been formally proved, although the proofs are omitted here for conciseness.

3.2 Extending QCRMut for MT

Once the MRs used in this study have been introduced, they need to be integrated into the mutation-testing process. For this purpose, we use QCRMut to generate and execute mutants and extend its testing functionality to support the evaluation of the proposed MRs. Figure 1 presents the general workflow. Given a quantum circuit, QCRMut generates the mutants, while the inputs required by the corresponding MR are generated separately. Each mutant is then evaluated using these inputs, and the results are stored for later analysis.

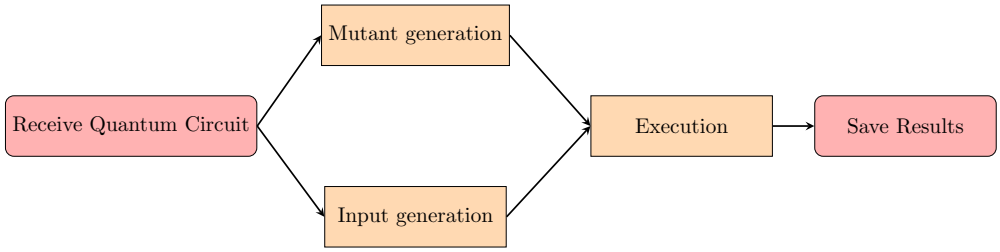


Fig. 1 Simplified Mutation Testing Workflow

The execution of each individual mutant is detailed in Figure 2. For every input, a test circuit is constructed from the mutant and executed. The resulting information is

then analysed according to the corresponding MR to determine whether the mutant has been killed. If the mutant survives and further inputs are available, the process is repeated with the next input. Otherwise, the mutant is classified as surviving.

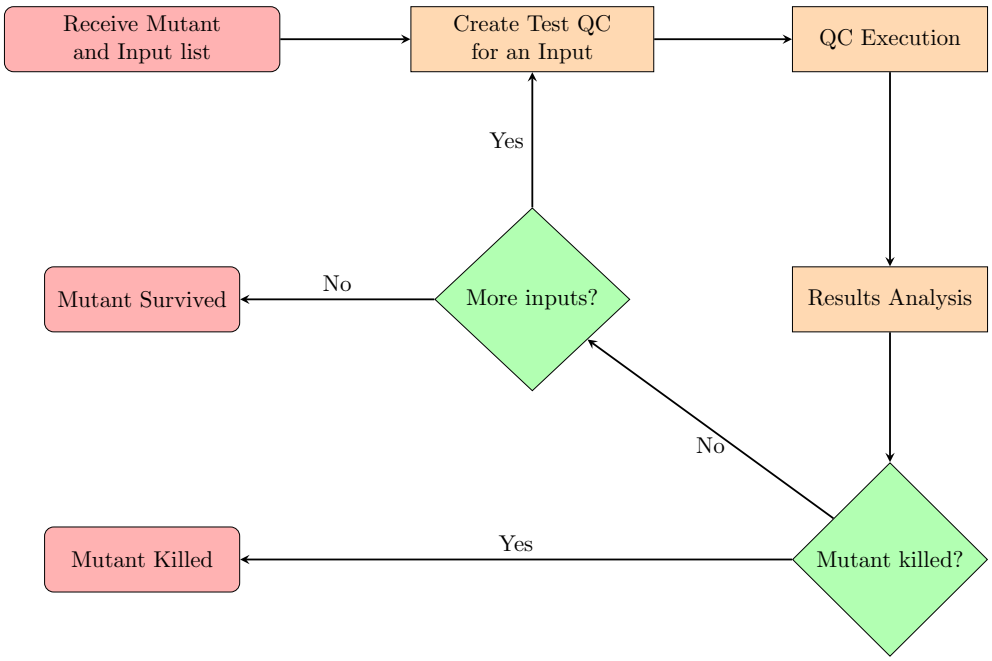


Fig. 2 Execution Workflow for a Mutant

The original QCRMut testing process determines the expected oracle output before circuit execution and evaluates the measured results accordingly. In this work, we extend this process to support the proposed MRs, allowing certain properties to be verified before measurement and minimising dependence on statistical analysis of measurement outcomes. To achieve this, we incorporated a more flexible oracle-analysis mechanism into QCRMut, allowing results to be evaluated according to the requirements of each MR rather than relying exclusively on a predefined expected output. We also generalised the execution process to accommodate different testing requirements.

We also adapted the test-circuit construction process to accommodate the different requirements of the proposed MRs. Circuits are now prepared for each generated input, allowing mutants and their corresponding inputs to be combined during execution. The measurement and result-storage procedures were also modified to support different circuit configurations and avoid storing redundant results. Together, these extensions allow the proposed MRs to be integrated into QCRMut without altering its mutant-generation process, providing the MR-specific input generation and analysis required to evaluate their fault-detection capability.

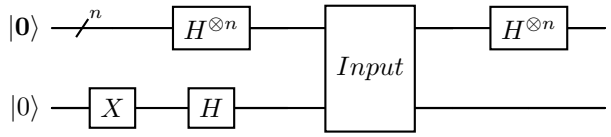
4 From MRs to Testing Structures

We now address one of the main objectives of this work: assessing the effectiveness and fault-detection capability of the proposed MRs. To this end, we translate their mathematical definitions into testing structures that allow the testing process to be automated. The resulting structures are then used in the experiments presented in the next section. We first describe how the proposed MRs are instantiated. Each instantiation provides the functions required for the different stages of the testing process: mutant generation, input generation, test-circuit construction, execution, and result analysis. Two of these five elements are common to all MRs:

Mutant generator: We use QCRMut because it provides flexibility in the mutation process by operating directly on Qiskit's high-level circuit representation. In particular, its immutable positions allow the structure of the algorithms to be preserved during mutation. This is essential when applying our MRs, as we require consistent input locations and well-defined correspondences between inputs.
Execution: The resulting quantum circuits, QCs, are executed within the extended QCRMut testing structure using Qiskit's `AerSimulator` with a fixed seed to ensure reproducibility and the default number of 1024 shots per execution.

The preparation of the algorithm implementations is also common to all MRs. For each algorithm, the instruction corresponding to the input is replaced by an instance of QCRMut's `PlaceHolder` instruction. This preserves the input location during mutant generation and allows the appropriate input to be inserted when constructing each test circuit. Measurements are omitted at this stage, as the test-circuit generator is responsible for constructing the circuit and introducing the measurements required by the corresponding MR.

Example 1 (DJ implementation circuit)



The inputs of the algorithms considered in this work are functions satisfying specific properties imposed by the corresponding problem. To use these functions within the testing process, they must be represented by the information required to construct their corresponding oracle circuits. We refer to this representation as the *computational input* and formalise the transformation from the input space of an algorithm to its oracle implementation as follows.

Definition 13 (Computational input) Let A be a quantum algorithm with input space $\mathcal{I}_{\mathrm{A}}$. We denote by $\widehat{\mathcal{I}}_{\mathrm{A}}$ the *computational input space* of A. A *computational input* is an element $\widehat{i} \in \widehat{\mathcal{I}}_{\mathrm{A}}$ containing the information required to construct an oracle corresponding to $i \in \mathcal{I}_{\mathrm{A}}$.

Definition 14 (Computational input generator) Let $\mathcal{I}_\mathrm{A}$ and $\widehat{\mathcal{I}}_\mathrm{A}$ be the input and computational input spaces of an algorithm A, respectively. A *computational input generator* is a function $\rho_\mathrm{A} : \mathcal{I}_\mathrm{A} \to \widehat{\mathcal{I}}_\mathrm{A}$, which maps an input $i \in \mathcal{I}_\mathrm{A}$ to its computational input $\widehat{i} = \rho_\mathrm{A}(i)$.

Definition 15 (Oracle generator) Let $\widehat{\mathcal{I}}_\mathrm{A}$ be the computational input space of an algorithm A. An *oracle generator* is a function $\omega_\mathrm{A} : \widehat{\mathcal{I}}_\mathrm{A} \to \mathcal{C}_A$, where $\mathcal{C}_A$ denotes the set of quantum circuits implementing valid oracles for A. Given an input $i \in \mathcal{I}_\mathrm{A}$, its corresponding oracle circuit is $U_i = \omega_\mathrm{A}(\rho_\mathrm{A}(i)) = \omega_\mathrm{A}(\widehat{i}\,)$.

Here, the term *oracle* refers to the quantum oracle used by the algorithm, rather than a test oracle that determines the correctness of an execution. A quantum oracle encodes a function as a quantum operation, which the algorithm accesses as a black box. The oracle generator ω_A constructs this operation from the computational input $\widehat{i}$. Thus, the transformation from an input of algorithm A to its corresponding oracle circuit is represented by

$$\mathcal{I}_\mathrm{A} \xrightarrow{\rho_\mathrm{A}} \widehat{\mathcal{I}}_\mathrm{A} \xrightarrow{\omega_\mathrm{A}} \mathcal{C}_A,$$

or, for a particular input $i \in \mathcal{I}_\mathrm{A}$,

$$i \xmapsto{\rho_\mathrm{A}} \widehat{i} \xmapsto{\omega_\mathrm{A}} U_i.$$

In our implementation, the oracle generators ω_A are based on the oracle constructions provided in Qiskit's learning materials[1]. When necessary, these constructions are adapted so that the information required to construct the oracle can be provided explicitly through the computational input $\widehat{i}$.

The computational input space $\widehat{\mathcal{I}}_\mathrm{A}$ and the computational input generator ρ_A depend on the algorithm and the adopted oracle construction. Other elements of the testing structure, such as test-circuit generation and oracle analysis, also depend on the requirements of each MR. In the following subsections, we describe the complete testing structure required to apply the proposed MRs to each algorithm.

4.1 Application of the DJ MRs

We begin with the DJ algorithm. The original circuit is prepared as described above, replacing its input location with a `PlaceHolder` instruction. We then instantiate the computational input space $\widehat{\mathcal{I}}_\mathrm{DJ}$ and its corresponding generator ρ_DJ.

The Qiskit construction adopted for the DJ oracle generator ω_DJ includes random choices during oracle construction. We adapt it so that these choices are explicitly determined by the computational input $\widehat{i}$. In particular, the construction first determines whether the output qubit is flipped and whether the generated function is constant or balanced. In the balanced case, it additionally selects half of the computational basis elements. These three decisions determine the information that must be provided by a DJ computational input.

[1] https://learning.quantum.ibm.com/course/fundamentals-of-quantum-algorithms/quantum-query-algorithms

Definition 16 (DJ computational input) Let n be the number of input qubits of the DJ algorithm. A *DJ computational input* has the form

$$\widehat{i} = (a, b, S),$$

where $a, b \in \{0, 1\}$ and $S \subseteq \{0, \ldots, 2^n - 1\}$, $|S| = 2^{n-1}$. The value a determines whether the output values are flipped, b determines whether the function is constant, $b = 1$, or balanced, $b = 0$, and S contains the basis elements selected by the oracle construction when the function is balanced.

The constraints imposed by the DJ problem restrict the possible values of these components. For a constant function, S is not required and a determines whether all inputs are mapped to 0 or to 1. For a balanced function, S contains exactly half of the 2^n basis elements. Consequently, the computational input space is defined as follows.

Definition 17 (DJ computational input space) Let n be the number of input qubits of the DJ algorithm. The *DJ computational input space* is

$$\widehat{\mathcal{I}}_{\mathrm{DJ}}^{n} = \Big\{(a, 1, \emptyset) \;\Big|\; a \in \{0, 1\}\Big\} \cup \Big\{(a, 0, S) \;\Big|\; a \in \{0, 1\},\ S \subseteq \{0, \ldots, 2^n - 1\},\ |S| = 2^{n-1}\Big\}.$$

The first set represents the constant functions, where a determines the value to which all inputs are mapped. The second set represents the balanced functions, where S identifies the selected basis elements used in the oracle construction and a determines whether the corresponding output values are flipped. Note that different computational inputs may represent the same function: mapping the elements of S to 0 is equivalent to mapping those of its complement to 1. Consequently, the number of distinct balanced functions is smaller than the number of possible computational input representations, allowing redundant cases to be excluded from the experimental evaluation.

The procedure described above instantiates the computational input generator $\rho_{\mathrm{DJ}} : \mathcal{I}_{\mathrm{DJ}} \to \widehat{\mathcal{I}}_{\mathrm{DJ}}$, which maps each DJ input $i \in \mathcal{I}_{\mathrm{DJ}}$ to its corresponding computational input $\widehat{i} = \rho_{\mathrm{DJ}}(i)$.

In the implementation, a computational input $\widehat{i} = (a, b, S)$ is encoded as `(a, b, list[int])`, with the third component omitted for constant functions. The following examples illustrate this encoding for both constant and balanced inputs.

Example 2 (DJ computational input examples) Let $n = 3$ for balance functions,

- **Constant:** There are only two constant functions. Therefore, the computational input `(0, 1, -)` represents the function that maps every basis element to 0, whereas `(1, 1, -)` represents the function that maps every basis element to 1.
- **Balanced with a = 0:** The computational input `(0, 0, [2, 3, 6, 7])` represents a balanced function in which the basis elements $(2, 3, 6, 7)$ are mapped to 1 and the remaining basis elements $(0, 1, 4, 5)$ are mapped to 0.
- **Balanced with a = 1:** Similarly, `(1, 0, [2, 3, 6, 7])` represents the complementary balanced function: the basis elements $(2, 3, 6, 7)$ are mapped to 0, while the remaining basis elements $(0, 1, 4, 5)$ are mapped to 1.

Note that different computational inputs may represent the same function. For example, complementing both a and the selected set of basis elements preserves the function's mapping. However, these inputs may produce different oracle circuits, involving different gate operations and potentially exercising different parts of the circuit during testing.

Once the source computational input $\widehat{i}_0$ has been generated, we define the follow-up computational input $\widehat{i}_1$ for each MR. For $\mathrm{MR}^1_{\mathrm{DJ}}$, the bijection g must be considered when constructing the follow-up test circuit. We implement g by permuting the input qubits before applying the oracle U_f. The inverse permutation should then be applied after U_f. However, since the MR only checks whether the resulting state is $|\mathbf{0}\rangle$, the qubit order does not affect the outcome. We therefore omit the inverse permutation, reducing the number of gates in the test circuit. The initial permutation is implemented using Qiskit's built-in permutation functionality.

Definition 18 (DJ follow-up computational inputs) Let $\widehat{i}_0 = (a, b, S) \in \widehat{\mathcal{I}}_{\mathrm{DJ}}$ be a source computational input. The follow-up computational inputs are constructed as follows:

$\mathbf{MR^1_{DJ}}$: The follow-up computational input is defined as $(\widehat{i}_0, P_g)$, where P represents the permutation induced by the bijection g. In the implementation, this is represented as `(i0, P(g))`.
$\mathbf{MR^2_{DJ}}$: The follow-up computational input is defined as $\widehat{i}_1 = (1 - a, b, S)$. In the implementation, this is represented as `(1 - a, b, list(S))`.

For $\mathrm{MR}^1_{\mathrm{DJ}}$, the permutation induced by g reorders the basis elements associated with the source input. Therefore, although the follow-up input could also be expressed using the computational input representation defined by $\widehat{\mathcal{I}}_{\mathrm{DJ}}$, this conversion is unnecessary, as the permutation is applied directly at test-circuit construction.

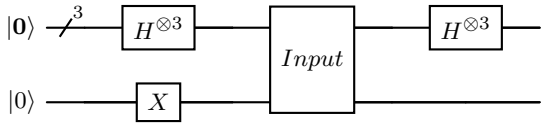
Example 3 (DJ follow-up input examples) Let $\widehat{i}_0 \in \widehat{\mathcal{I}}_{\mathrm{DJ}}$, with representation `i0 = (0, 0, [2, 3, 6, 7])`.

$\mathbf{MR^1_{DJ}}$: If $i_1 = i_0 \cdot g$ and g is represented by the permutation $P = (0, 1, 2, 4, 3, 5, 6, 7)$. Then `i1 = ((0, 0, [2, 3, 6, 7]), P)` represents $\widehat{i}_1$. In computational-input form, this can be rewritten as $\widehat{i}_1 = (0, 0, (2, 4, 6, 7))$.
$\mathbf{MR^2_{DJ}}$: The follow-up input representation is `i1 = (1, 0, [2, 3, 6, 7])`.

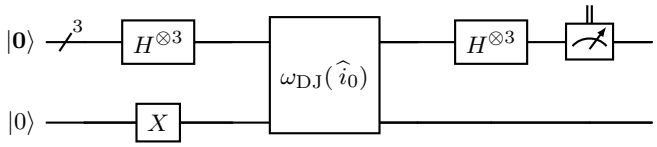
Once the mutants and computational inputs have been generated, the test circuits are constructed. For $\mathrm{MR}^2_{\mathrm{DJ}}$, the placeHolder instruction is replaced by $\omega_{\mathrm{DJ}}(\widehat{i}_0)$ and $\omega_{\mathrm{DJ}}(\widehat{i}_1)$ in the source and follow-up circuits, respectively. For $\mathrm{MR}^1_{\mathrm{DJ}}$, it is replaced by $\omega_{\mathrm{DJ}}(\widehat{i}_0)$ in the source circuit and by the permutation P followed by $\omega_{\mathrm{DJ}}(\widehat{i}_0)$ in the follow-up circuit. Here, ω_{DJ} denotes the Qiskit-based DJ oracle generator. In both cases, measurements are added after constructing the circuits. The following example illustrates the test-circuit construction for $\mathrm{MR}^1_{\mathrm{DJ}}$.

Example 4 (DJ MR1 test-circuit construction) Let M be a mutant of the DJ implementation presented in Example 1, with $n = 3$. Let $i_0 \in \mathcal{I}_{\mathrm{DJ}}$ be the source input, $\widehat{i_0} = \rho_{\mathrm{DJ}}(i_0)$ its computational input, and $g \in \mathrm{Sim}(\{0,1\}^3)$ a bijection. The follow-up input is $i_1 = i_0 \cdot g$, and P denotes the permutation implementing g in the test circuit.

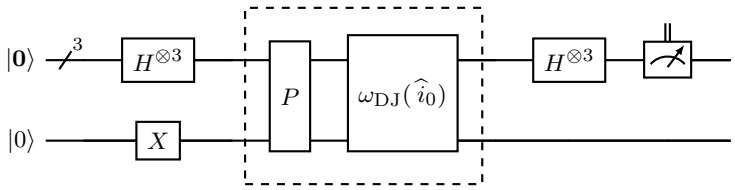
Consider the mutant M:



The source test circuit replaces the input with $\omega_{\mathrm{DJ}}(\widehat{i_0})$ and adds the measurements:



The follow-up test circuit replaces the input with P followed by $\omega_{\mathrm{DJ}}(\widehat{i_0})$, which implements the transformation corresponding to $\widehat{i_1}$, and adds the measurement:



The measurement outcomes of the resulting circuits are then analysed to determine whether the expected relation holds. Recall that, for the DJ algorithm, a constant function produces $|\mathbf{0}\rangle$ as its only possible measurement outcome, whereas a balanced function never produces $|\mathbf{0}\rangle$. We use these properties to interpret the observed results, while accounting for mutants that may exhibit neither behaviour. The analysis distinguishes three cases based on the measurement outcomes of the first circuit:

- If $|\mathbf{0}\rangle$ is the only outcome of the first circuit, its results are consistent with a constant function. The mutant is killed if the second circuit produces any outcome other than $|\mathbf{0}\rangle$, as its results would no longer be consistent with a constant function.
- If $|\mathbf{0}\rangle$ does not appear among the outcomes of the first circuit, its results are consistent with a balanced function. The mutant is killed if $|\mathbf{0}\rangle$ appears among the outcomes of the second circuit, whose results would then be consistent with a constant function or with neither a constant nor a balanced function.

Result $\widehat{i}_1$ \ Result $\widehat{i}_0$	Constant	Balanced	Neither
Constant	**Survived**	Killed	Killed
Balance	Killed	**Survived**	Killed
Neither	Killed	Killed	Killed

Table 2 DJ Analysis summary.

- If the first circuit produces multiple outcomes including $|\mathbf{0}\rangle$, its results are consistent with neither a constant nor a balanced function. The mutant is therefore killed regardless of the outcomes of the second circuit.

These cases define the DJ analysis function, summarised in Table 2. Finally, we construct the testing structures using the functions defined above. The two DJ MRs differ in their follow-up input generation and test-circuit construction. Table 3 summarises the resulting testing structures.

4.2 Application of the BV MRs

Following the same structure, we now instantiate the testing components required for the BV MRs, omitting those analogous to the components already introduced for DJ, including the preparation of the original circuit. Each BV input $i \in \mathcal{I}_{\mathrm{BV}}$ is determined by a binary string of length n, which directly constitutes the computational input, as it contains all the information required to construct the corresponding oracle.

Definition 19 (BV computational input) Let n be the number of input qubits of the BV algorithm. A BV computational input is a binary string $\widehat{i} \in \{0,1\}^n$. Consequently, the computational input space is $\widehat{\mathcal{I}}_{\mathrm{BV}} = \{0,1\}^n = \mathcal{I}$.

The computational input generator $\rho_{\mathrm{BV}} : \mathcal{I}_{\mathrm{BV}} \to \widehat{\mathcal{I}}_{\mathrm{BV}}$ maps each BV input i to the binary string $\widehat{i} = \rho_{\mathrm{BV}}(i)$ that determines its oracle. The Qiskit-based oracle generator ω_{BV} then constructs the corresponding oracle from $\widehat{i}$. Once the source computational inputs have been generated, the remaining computational inputs are constructed according to each MR.

Definition 20 (BV follow-up computational inputs) Let $\widehat{i}_0, \widehat{i}_1 \in \widehat{\mathcal{I}}_{\mathrm{BV}}$. The additional computational inputs are constructed as follows:

$\mathbf{MR^1_{BV}}$: $\widehat{i}_2 = \widehat{i}_0 \oplus \widehat{i}_1$.
$\mathbf{MR^2_{BV}}$: $\widehat{i}_1 = \widehat{i}_0 \oplus \mathbf{1}$.
$\mathbf{MR^3_{BV}}$: $\widehat{i}_2 = \widehat{i}_0 \cdot \widehat{i}_1$, where $\cdot$ represents the successive application of $\omega_{\mathrm{BV}}(\widehat{i}_0)$ and $\omega_{\mathrm{BV}}(\widehat{i}_1)$. This is implemented directly during test-circuit construction.

The following examples illustrate the construction of the follow-up computational inputs for $\mathrm{MR}^1_{\mathrm{BV}}$ and $\mathrm{MR}^2_{\mathrm{BV}}$. $\mathrm{MR}^3_{\mathrm{BV}}$ is omitted, as its composition is implemented directly during test-circuit construction.

Example 5 (BV follow-up computational input examples) Consider BV computational inputs with $n = 4$.

$\mathbf{MR^1_{BV}}$: Let $\widehat{i}_0 = \texttt{1010}$ and $\widehat{i}_1 = \texttt{0110}$. Then $\widehat{i}_2 = \widehat{i}_0 \oplus \widehat{i}_1 = \texttt{1100}$.
$\mathbf{MR^2_{BV}}$: Let $\widehat{i}_0 = \texttt{1010}$. Its bitwise complement gives the follow-up computational input $\widehat{i}_1 = \widehat{i}_0 \oplus \mathbf{1} = \texttt{0101}$.

Following the generation of computational inputs, the test circuits are constructed by replacing the placeHolder with the corresponding Qiskit-based oracles $\omega_{\mathrm{BV}}(\widehat{i})$. Unlike the DJ MRs, which require analysing the measurement outcomes of separate circuits, the BV MRs combine the outputs within a single quantum circuit before measurement. This is achieved using CNOT gates to perform bitwise XOR operations between the output registers, transforming the expected relation into a condition that can be verified directly through measurement. By performing this comparison within the quantum circuit, the testing process avoids the need to measure each output separately and statistically analyse their relationships, minimising dependence on the probabilistic nature of measurement.

The construction relies on the property that, for $\mathbf{a}, \mathbf{b} \in \{0,1\}^n$,

$$\mathbf{a} = \mathbf{b} \iff \mathbf{a} \oplus \mathbf{b} = \mathbf{0}.$$

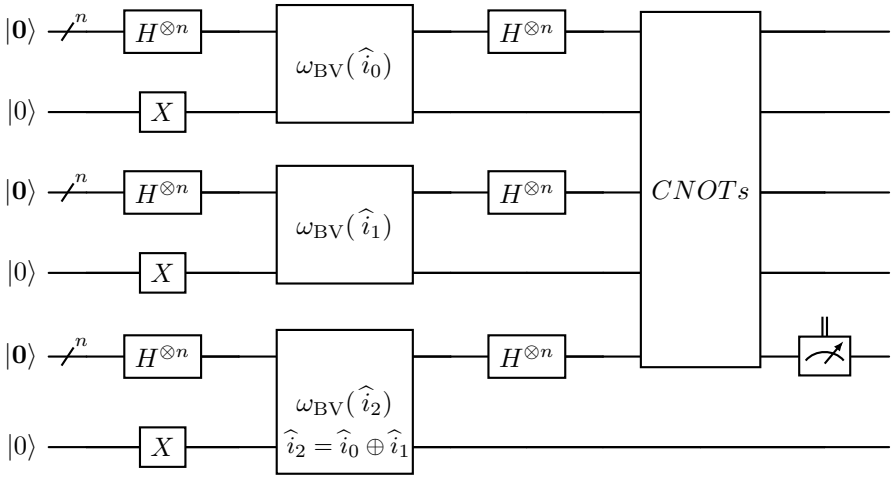
For instance, $\mathrm{MR}^1_{\mathrm{BV}}$ requires $\mathbf{o}_2 = \mathbf{o}_0 \oplus \mathbf{o}_1$, and therefore $\mathbf{o}_2 \oplus \mathbf{o}_0 \oplus \mathbf{o}_1 = \mathbf{0}$.

The test circuits are constructed so that this principle is applied to the relation defined by each MR. Consequently, when an MR is satisfied, the measured register yields $|\mathbf{0}\rangle$ with probability 1. A mutant is therefore killed if $|\mathbf{0}\rangle$ is not the sole measurement outcome. The next example shows the BV MRs test-circuit construction.

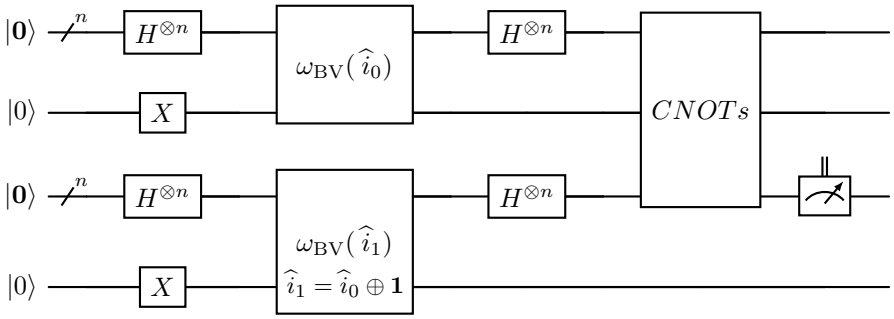
The three BV MRs differ in their follow-up input generation and test-circuit construction, while sharing the same analysis criterion. The resulting testing structures are summarised in Table 3. We now proceed to the final algorithm considered in this work, Simon's algorithm.

Example 6 (BV MRs test-circuit construction) Let M be a mutant of the BV implementation. Let $i_0, i_1 \in \mathcal{I}_{\mathrm{BV}}$ be source inputs, when needed, with computational inputs $\widehat{i}_0 = \rho_{\mathrm{BV}}(i_0)$ and $\widehat{i}_1 = \rho_{\mathrm{BV}}(i_1)$. The corresponding follow-up inputs are constructed according to each MR as defined above. We consider M to have the same structure as the mutant in Example 4. The test circuits for each MR are constructed by replacing the input with the corresponding oracles and combining their outputs according to the relation defined by each MR. The resulting circuits are shown below.

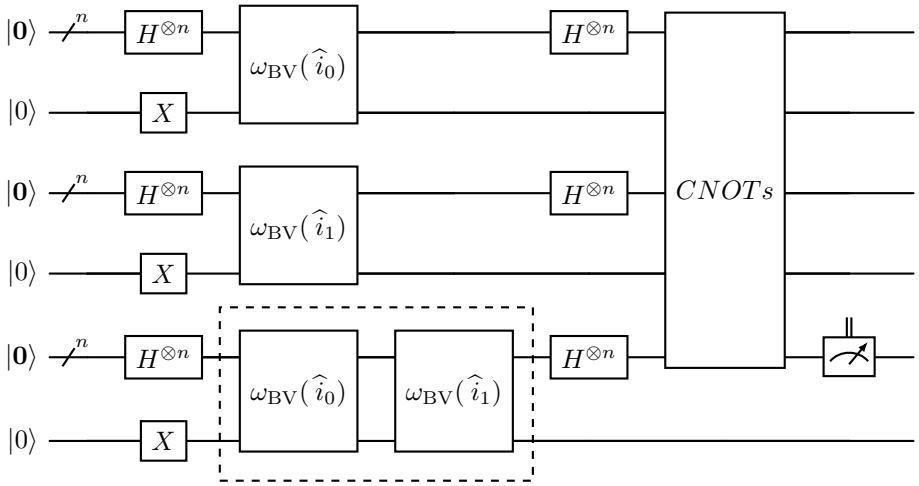
$\mathbf{MR^1_{BV}}$:



$\mathbf{MR^2_{BV}}$:



$\mathbf{MR^3_{BV}}$:



4.3 Application of the Simon MRs

We finally instantiate the testing components required for the Simon MR. As for the previous algorithms, the original circuit is prepared using the procedure described above. The oracle generator ω_S is based on the implementation provided in Qiskit's learning materials [34].

For Simon's algorithm, each input function is determined by a non-zero binary string that defines its hidden period. As this string contains the information required to construct the corresponding oracle, it directly constitutes the computational input.

Definition 21 (Simon computational input) Let n be the number of input qubits of Simon's algorithm. A Simon computational input is a binary string $\widehat{i} \in \{0,1\}^n \setminus \{\mathbf{0}\}$. Consequently, the computational input space is $\widehat{\mathcal{I}}_S = \{0,1\}^n \setminus \{\mathbf{0}\}$.

The computational input generator $\rho_S : \mathcal{I}_S \to \widehat{\mathcal{I}}_S$ maps each Simon input i to the binary string $\widehat{i} = \rho_S(i)$ that determines its hidden period. The Qiskit-based oracle generator ω_S then constructs the corresponding oracle from $\widehat{i}$.

Once the source computational input $\widehat{i}_0$ has been generated, the follow-up computational input is constructed by applying the inversion operator defined in Definition 11.

Definition 22 (Simon follow-up computational input) Let $\widehat{i}_0 \in \widehat{\mathcal{I}}_S$ be a source computational input. The follow-up computational input for MR_S^1 is $\widehat{i}_1 = \overline{\widehat{i}_0}$.

The following example illustrates this transformation.

Example 7 (Simon follow-up computational input example) Let $n = 4$ and $\widehat{i}_0 = \texttt{1011}$. The follow-up computational input is

$$\widehat{i}_1 = \overline{\widehat{i}_0} = \texttt{1101}.$$

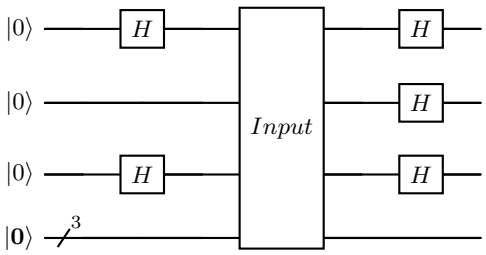
The test circuits are constructed by replacing the placeHolder with $\omega_S(\widehat{i}_0)$ and $\omega_S(\widehat{i}_1)$, respectively, and adding the required measurements. As MR_S^1 requires $o_0 = \overline{o_1}$, the inversion of the follow-up output is performed directly during measurement. The following example illustrates this construction.

The analysis can then compare the two sets of measurement outcomes directly. Qiskit returns the results as dictionaries whose keys are the observed bit strings and whose values are their repetition counts. Since MR_S^1 concerns the set of possible outcomes rather than their frequencies, only the sets of keys are compared. The mutant survives if both sets are equal and is killed otherwise.

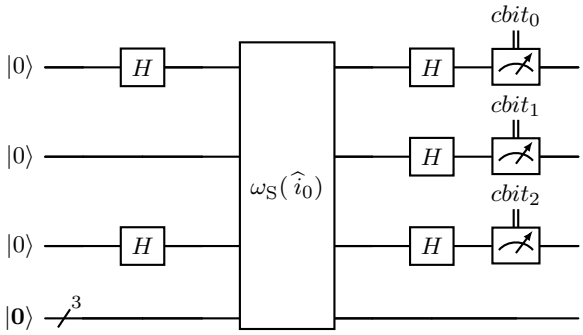
Example 8 (Simon MR test-circuit construction) Let M be a mutant of the Simon implementation with $n = 3$, and let $i_0 \in \mathcal{I}_S$ be a source input with computational input $\widehat{i}_0 = \rho_S(i_0)$.

The follow-up computational input is $\widehat{i}_1 = \overline{\widehat{i}_0}$. The source and follow-up test circuits replace the input with $\omega_S(\widehat{i}0)$ and $\omega_S(\widehat{i}_1)$, respectively.

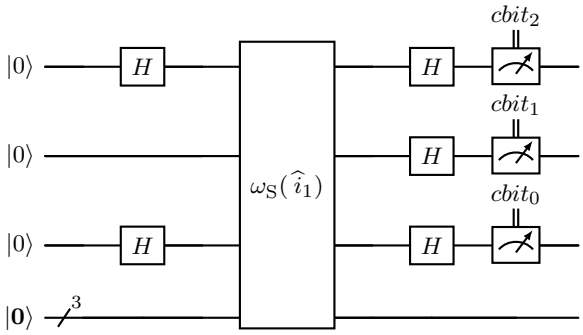
Consider the mutant M:



The source test circuit replaces the input with $\omega_S(\widehat{i}_0)$ and adds the measurements:



The follow-up test circuit replaces the input with $\omega_S(\widehat{i}_1)$ and adds the measurements:



In the source circuit, the qubits are measured into the classical register in their original order. In the follow-up circuit, they are measured in reverse order. Consequently, the measurement outcomes of the follow-up circuit directly represent $\overline{o_1}$, as required by MR_S^1.

The summary of all our MRs quantum testing structures can be found in Table 3.

Alg	MR	Input generator	Circuit generator	MR Analysis
DJ	1	DJ input generator and permutation	DJ MR1	DJ Analysis
	2	DJ input generator	DJ MR2	
BV	1	BV pair input	BV MR1	BV Analysis
	2	BV xor input	BV MR2	
	3	BV pair input	BV MR3	
Simon	1	Single input generator	Simon MR1	Simon Analysis

Table 3 MRs structure summary.

5 Experiments and Results

This section evaluates the validity and fault-detection effectiveness of the proposed MRs through mutation testing, as well as the potential for reducing the number of inputs required. To this end, we conduct three experiments using the testing structures introduced in the previous section and mutants generated by QCRMut. We first assess the proposed MRs through exhaustive evaluation, establishing reference mutation scores. We then investigate whether these scores can be reproduced using fewer inputs and examine their consistency across random input selections. Finally, we extend the analysis to larger configurations to investigate mutation-score stabilisation without an exhaustive reference.

All experimental results, testing structures, and replication instructions are available in the provided GitHub repository[2]. To ensure reproducibility, fixed seeds were used for both mutant generation and circuit execution. The experiments were executed using Python 3.12.2 and Qiskit 2.3.0 on Debian 12.2.0-14, on a system with an Intel(R) Xeon W-2235 CPU (6 physical and 12 virtual cores) and 128 GB of RAM.

5.1 MR Quality Experiments

In Experiment I, we exhaustively evaluate the input combinations defined by each MR to verify their validity for the original quantum programs and assess their fault-detection effectiveness through mutation testing. We consider BV and Simon with 3, 4, 5, and 6 input qubits, whereas DJ is evaluated with 3 and 4 input qubits. Larger DJ configurations are excluded from exhaustive evaluation due to the rapid combinatorial growth of their computational input space. The resulting mutation scores serve as references for the input-reduction analysis in Experiment II.

Rather than fixing an arbitrary number of mutants, we determine it using the coverability criterion proposed for QCRMut [26]. Full *coverability* is achieved when every gate and applicable mutation operator combination is represented by at least one mutant. QCRMut estimates the number of mutants required to achieve this criterion using a formula derived from the *Coupon Collector's problem* [35]. This estimate

[2] https://anonymous.4open.science/r/MetTesting-52B8/README.md

Algorithm	# Input qubits (n)	# Algorithm qubits	# Gates	# Mutants	$\lvert\widehat{\mathcal{I}}_{\mathrm{A}}\rvert$
DJ	3	4	9	256	72
	4	5	11	256	12872
	5	6	13	256	601080392
	6	7	15	288	$\binom{2^n}{2^{n-1}} + 2$
	7	8	17	320	
BV	3	4	9	256	8
	4	5	11	256	16
	5	6	13	256	32
	6	7	15	288	64
	7	8	17	320	128
Simon	3	6	7	256	7
	4	8	9	256	15
	5	10	11	256	31
	6	12	13	256	63
	7	14	15	288	127

Table 4 Experimental configurations considered across Experiments I–III, including circuit size, number of generated mutants, and source computational input-space size.

Algorithm	MR	# Circuits	# Qubits per circuit	# Input combinations
DJ	1	2	$n+1$	$\lvert\widehat{\mathcal{I}}_{\mathrm{DJ}}\rvert \cdot n!$
	2		$n+1$	$\lvert\widehat{\mathcal{I}}_{\mathrm{DJ}}\rvert$
BV	1	1	$3(n+1)$	$\lvert\widehat{\mathcal{I}}_{\mathrm{BV}}\rvert^2$
	2		$2(n+1)$	$\lvert\widehat{\mathcal{I}}_{\mathrm{BV}}\rvert$
	3		$3(n+1)$	$\lvert\widehat{\mathcal{I}}_{\mathrm{BV}}\rvert^2$
Simon	1	2	$2n$	$\lvert\widehat{\mathcal{I}}_{\mathrm{S}}\rvert$

Table 5 Testing resources required by each MR, where n denotes the number of input qubits of the original algorithm. Input combinations include the source and follow-up computational inputs defined by each MR.

corresponds to the mean of the resulting distribution. Table 4 presents the configurations considered across the experiments, including the size of the source computational input space and the number of mutants generated for each configuration.

The testing structures introduce additional computational requirements in terms of both the number of qubits and the number of input combinations. Let n denote the number of input qubits of the original algorithm. Depending on the MR, several algorithm executions are combined within a single test circuit, increasing its number of qubits with respect to the original algorithm, a limitation already identified in previous work [28, 36]. Furthermore, the generation of follow-up computational inputs can increase the number of input combinations that must be evaluated. Table 5 summarises the number of circuits, qubits per circuit, and total input combinations defined by each MR, considering both source and follow-up computational inputs.

Alg	Rule	#Qbits	#Inp[1]	MS (%)	Execution time (s)
DJ	MR1	3	2	77.34	1549.72
		4	2	69.92	1458612.72
	MR2	3	2	77.34	268.92
		4	2	69.92	60884.56
BV	MR1	3	4	83.20	87.41
		4	7	75.39	2280.49
		5	7	77.73	11864.75
		6	8	75.35	70617.00
	MR2	3	2	83.59	16.68
		4	2	76.17	35.76
		5	3	79.30	53.74
		6	3	75.69	140.89
	MR3	3	4	83.20	87.09
		4	7	75.39	2285.13
		5	7	77.73	10944.78
		6	8	75.35	72200.01
Simon	MR1	3	2	41.80	60.54
		4	4	56.64	98.83
		5	7	49.61	219.55
		6	6	60.16	346.57

[1] Number of inputs at which the final mutation score was first obtained in Experiment I.

Table 6 Experiment I: Mutation scores obtained through exhaustive evaluation of the input space for each MR and algorithm configuration.

These requirements become particularly relevant when using simulation. An n-qubit system is represented by a state vector with 2^n amplitudes, while an operator acting on the complete system has dimension $2^n \times 2^n$. Consequently, increasing the number of qubits can considerably increase the computational and memory requirements of simulation, while increasing the number of input combinations results in a larger number of test executions. These factors become increasingly significant as n grows, eventually making exhaustive evaluation impractical.

Having defined the experimental configurations and their computational requirements, we evaluate the proposed MRs across the complete input space, verifying their validity for the original quantum programs and assessing their fault-detection effectiveness through mutation testing. The resulting mutation scores are presented in Table 6, together with the number of inputs used to obtain each score and the corresponding execution time. These mutation scores establish the exhaustive references used in Experiment II. These results allow us to address **RQ1: How effective are the proposed MRs at detecting faults in the selected quantum programs?**

The results confirm that the proposed MRs are satisfied by the original quantum programs across all evaluated input combinations and are capable of detecting faults in the generated mutants, although their effectiveness varies across algorithms and configurations. The BV MRs achieve the highest mutation scores, ranging from 75.35%

to 83.59%, with similar results across the three MRs. The DJ MRs also exhibit consistent fault-detection capability, with mutation scores between 69.92% and 77.34%. Simon's MR obtains lower scores, ranging from 41.80% to 60.16%. Overall, the results provide a positive answer to RQ1, demonstrating the fault-detection effectiveness of the proposed MRs while showing that it depends on the algorithm under test.

5.2 Input Reduction Experiments

While the previous experiment establishes the fault-detection capability of the proposed MRs, Table 6 also shows that at most eight inputs were needed to obtain the reported mutation scores during exhaustive evaluation. This motivates investigating whether randomly selected, reduced input sets can consistently reproduce those results, and whether a similar pattern is observed as the input space grows.

We first consider the configurations for which exhaustive results are available: 3 and 4 input qubits for DJ, and 3 to 6 input qubits for BV and Simon, Experiment II. For each configuration, inputs are randomly selected using 20 different seeds, progressively increasing their number from 2 to 20. For DJ, the selection is constrained to include at least one constant-function input, ensuring representativity, while the remaining inputs are randomly selected. The exhaustive mutation scores from Experiment I provide reference values: reaching the corresponding score means that the selected inputs detect all mutants identified as killable through exhaustive evaluation. To address RQ2.1, we determine the minimum number of inputs required to reach this reference for each seed, reporting its median and interquartile range, IQR. To address RQ2.2, we analyse variability across seeds and the proportion reaching the reference at each input count, accompanied by 95% Wilson confidence intervals to quantify uncertainty. We also examine the evolution of the mean mutation score and its standard deviation as additional inputs are considered.

Figure 3 shows the evolution of median mutation scores for BV, with shaded IQR bands illustrating variation across random input selections. Figure 4 shows the proportion of selections reaching the exhaustive reference as the number of inputs increases. Table 7 complements these plots with the mean mutation scores at selected input counts, the minimum input counts required, and the success rates at 10 inputs. These results allow us to address **RQ2.1: How many inputs are required to achieve the mutation score obtained using the complete input space?**

The evolution plots and Table 7 show that the exhaustive scores can be reproduced with substantially fewer inputs. Both DJ MRs require three inputs across all seeds and configurations, while the BV MRs require a median of two to four. Simon needs more inputs in some configurations: the median is four for three qubits, six for four qubits, 11.5 for five qubits, and eight for six qubits. Nevertheless, all 20 seeds reach the exhaustive score by 15 inputs in every evaluated configuration. RQ2.1 is therefore answered positively for these configurations: reduced input sets can reproduce the full set of mutant detections established through exhaustive evaluation.

These results also allow us to address **RQ2.2: How consistent is the required number of inputs across different input selections?**

Figure 4 and the IQR bands in Figure 3 shows that consistency depends on the algorithm and configuration. DJ exhibits no variation, and most BV configurations

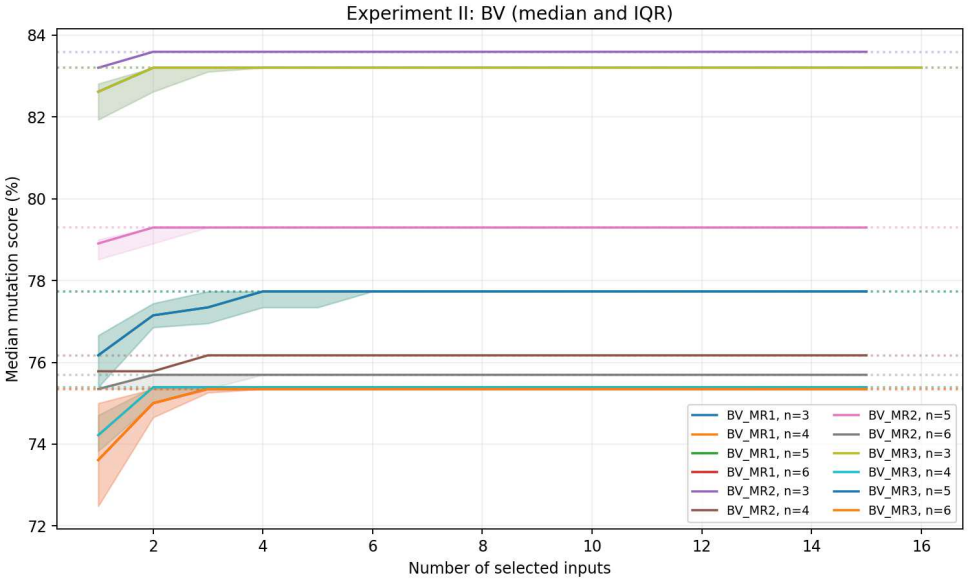


Fig. 3 Experiment II: Median mutation-score evolution for the BV MRs across 20 random input selections; shaded regions represent the IQR.

reach their exhaustive scores within 10 inputs for every seed; the exceptions are BV MR1 and MR3, where 18 or 19 of the 20 seeds succeed in some configurations. Simon is more sensitive to input selection: at five and six qubits, respectively, only 8/20 and 16/20 seeds reach the reference within 10 inputs. These proportions correspond to approximately 95% Wilson intervals of [21.9%, 61.3%] and [58.4%, 91.9%]. Thus, RQ2.2 indicates high consistency for DJ and most BV configurations, but greater variation for Simon. The results across 20 seeds do not guarantee the same input counts for every possible random selection.

Having established that reduced input sets can reproduce exhaustive mutation scores, we extend the analysis to larger configurations: 5, 6, and 7 input qubits for DJ, and 7 input qubits for BV and Simon, Experiment III. As the input space grows, the computational cost of exhaustive evaluation increases and will eventually become impractical. We therefore investigate the behaviour of 20 random input selections, using between 2 and 20 inputs, *without* an exhaustive score as a stopping reference. RQ3 focuses on mutation-score evolution, between-seed variability, and whether further mutant detections occur after apparent plateaus.

Figure 5 present the median mutation scores and IQR bands for Simon. DJ is again omitted because its scores remain constant across the evaluated input counts. Figure 6 highlights the gains obtained between 10 and 15 inputs, while Table 8 reports mean scores at selected input counts and the number of seeds detecting additional mutants between 10 and 20 inputs.

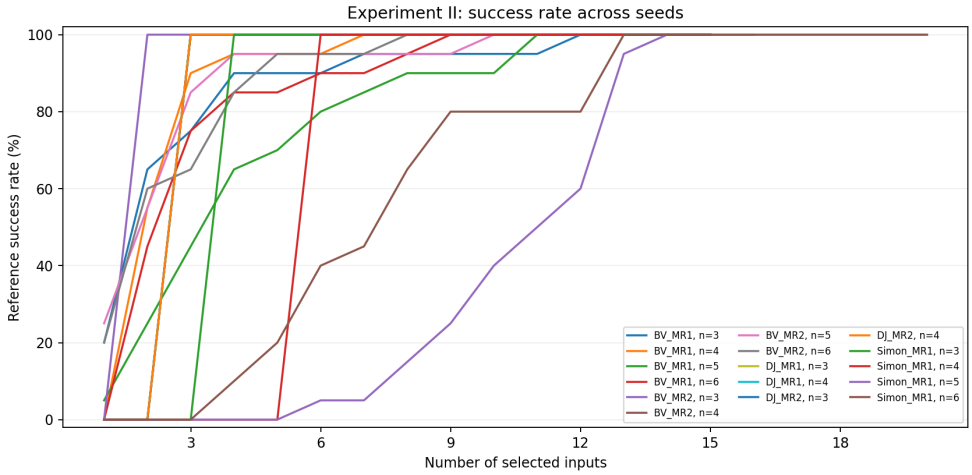


Fig. 4 Experiment II: Proportion of random input selections reaching the exhaustive mutation score as the number of inputs increases.

These results allow us to address **RQ3: Does the mutation score stabilise with a limited number of inputs for larger quantum programs?**

Figure 5, together with Table 8, show evidence of stabilisation within the evaluated range, although the rate differs by algorithm. Both DJ MRs show no additional detections beyond five inputs, and BV MR2 exhibits similarly limited variation. For BV MR1 and MR3, one seed in each case detects additional mutants after 10 inputs. Simon shows the clearest gains from additional inputs, as illustrated in Figure 6: its mean mutation score rises from 54.17% at 10 inputs to 54.51% at 15, and 6 of the 20 seeds detect further mutants between 10 and 20 inputs. Thus, an apparent plateau in an individual selection need not indicate that no further mutants can be detected.

By 15 inputs, all evaluated seeds within each configuration have the same mutation score, with no additional detections up to 20 inputs. Their variability also decreases as inputs are added; for Simon, for example, the standard deviation falls from 2.03 at five inputs to 0.81 at ten and 0.00 at fifteen. These observations support RQ3 by providing evidence that mutation scores can stabilise with a limited number of inputs. However, unlike Experiment II, there is no exhaustive reference against which to establish that every killable mutant has been detected.

Overall, the experiments show that the proposed MRs detect a substantial proportion of the generated faults, RQ1, that reduced input sets can reproduce the exhaustive results with varying consistency across algorithms and random selections, RQ2, and that mutation scores show evidence of stabilisation for larger configurations without an exhaustive stopping criterion, RQ3. The threats to the validity of these findings are discussed in the following subsection.

Alg	MR	n	MS_{exh}	Mean MS (%)[1]			k^*[2]		SR_{10}[3]
				5 inputs	10 inputs	15 inputs	Median	IQR	
DJ	MR1	3	77.34	77.34 ± 0.00	77.34 ± 0.00	77.34 ± 0.00	3	[3, 3]	20/20
		4	69.92	69.92 ± 0.00	69.92 ± 0.00	69.92 ± 0.00	3	[3, 3]	20/20
	MR2	3	77.34	77.34 ± 0.00	77.34 ± 0.00	77.34 ± 0.00	3	[3, 3]	20/20
		4	69.92	69.92 ± 0.00	69.92 ± 0.00	69.92 ± 0.00	3	[3, 3]	20/20
BV	MR1	3	83.20	83.09 ± 0.36	83.18 ± 0.09	83.20 ± 0.00	2	[2, 3.25]	19/20
		4	75.39	75.37 ± 0.09	75.39 ± 0.00	75.39 ± 0.00	2	[2, 3]	20/20
		5	77.73	77.58 ± 0.27	77.70 ± 0.12	77.73 ± 0.00	4	[2.75, 6]	18/20
		6	75.35	75.30 ± 0.13	75.35 ± 0.00	75.35 ± 0.00	3	[2, 3.25]	20/20
	MR2	3	83.59	83.59 ± 0.00	83.59 ± 0.00	83.59 ± 0.00	2	[2, 2]	20/20
		4	76.17	76.17 ± 0.00	76.17 ± 0.00	76.17 ± 0.00	3	[3, 3]	20/20
		5	79.30	79.28 ± 0.09	79.30 ± 0.00	79.30 ± 0.00	2	[2, 3]	20/20
		6	75.69	75.68 ± 0.08	75.69 ± 0.00	75.69 ± 0.00	2	[2, 4]	20/20
	MR3	3	83.20	83.09 ± 0.36	83.18 ± 0.09	83.20 ± 0.00	2	[2, 3.25]	19/20
		4	75.39	75.37 ± 0.09	75.39 ± 0.00	75.39 ± 0.00	2	[2, 3]	20/20
		5	77.73	77.58 ± 0.27	77.70 ± 0.12	77.73 ± 0.00	4	[2.75, 6]	18/20
		6	75.35	75.30 ± 0.13	75.35 ± 0.00	75.35 ± 0.00	3	[2, 3.25]	20/20
Simon	MR1	3	41.80	41.80 ± 0.00	41.80 ± 0.00	41.80 ± 0.00	4	[4, 4]	20/20
		4	56.64	55.47 ± 0.00	56.64 ± 0.00	56.64 ± 0.00	6	[6, 6]	20/20
		5	49.61	46.97 ± 4.08	49.38 ± 0.20	49.61 ± 0.00	11.5	[9.75, 13]	8/20
		6	60.16	59.04 ± 2.35	60.08 ± 0.16	60.16 ± 0.00	8	[6, 9]	16/20

[1] Mean mutation scores are reported as mean ± SD across 20 seeds.
[2] k^*: minimum inputs required to reach MS_{full}; IQR: interquartile range.
[3] SR_{10}: seeds reaching MS_{full} with at most 10 inputs. All seeds reach it by 15 inputs.

Table 7 Input-reduction results for Experiment II, including the exhaustive mutation scores obtained in Experiment I, Table 6.

5.3 Threats to Validity

One of the main threats to external validity concerns the selection of quantum algorithms. By deriving and evaluating six MRs based on different properties of three quantum algorithms, we have shown the applicability of our approach across distinct algorithmic characteristics. Nevertheless, since the definition of MRs depends on the specific properties and expected behaviour of each algorithm, their application to other quantum programs may require further adaptations.

Another threat concerns the use of mutation score to assess the fault-detection effectiveness of the proposed MRs. In particular, equivalent mutants cannot be distinguished from surviving non-equivalent mutants without additional analysis, potentially underestimating the mutation score. Although QCRMut provides a systematic mutant-generation process, the results remain dependent on the faults represented by its mutation operators and the selected programs.

The input-reduction experiments introduce further threats related to random input selection. To mitigate the influence of individual selections, we evaluate each configuration using 20 different seeds and analyse the variability of the resulting mutation scores. Nevertheless, this number of repetitions provides only a limited estimate of the behaviour across all possible input selections. Consequently, the number of inputs

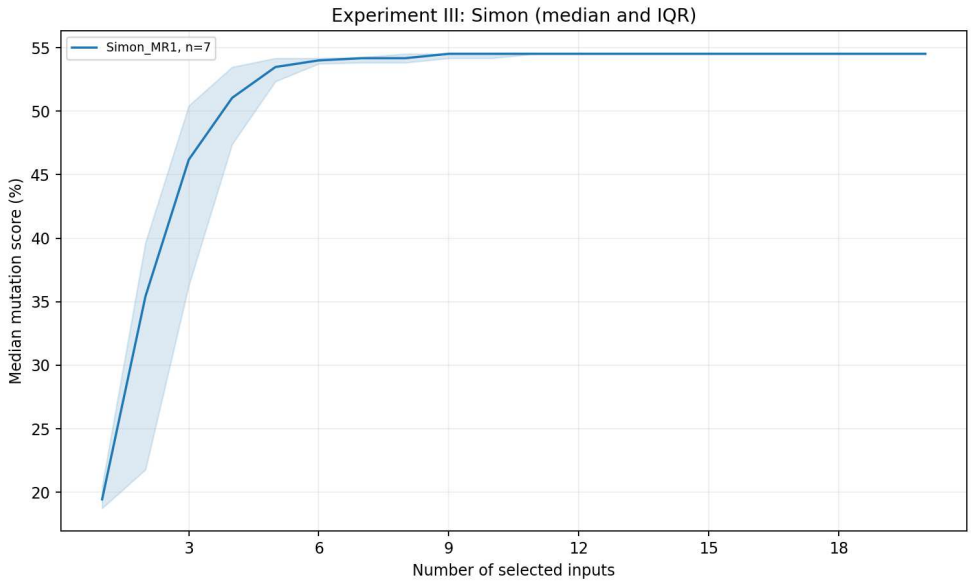


Fig. 5 Experiment III: Median mutation-score evolution for the Simon MR with seven input qubits; shaded regions represent the IQR.

required to reproduce the exhaustive mutation score may vary for random selections not covered by our experiments.

A related limitation concerns the interpretation of stabilisation in Experiment III. Due to the computational cost associated with the growth of the input space, exhaustive mutation scores are not calculated for larger configurations. Instead, we examine how mutation scores evolve across random input selections without an exhaustive reference to determine when testing can stop. Consequently, the observed stabilisation does not guarantee that further inputs would fail to detect additional mutants.

Finally, all experiments are conducted using quantum circuit simulation, which does not fully reproduce the noise, hardware constraints, and execution variability of real quantum devices. Although simulation provides a controlled environment for evaluating the proposed MRs, their behaviour on actual quantum hardware remains to be investigated. These limitations motivate further research into MR identification, equivalent-mutant detection, input-selection strategies, and the application of the proposed approach to real quantum systems and quantum algorithms employed in practical applications.

6 Conclusion and Future Work

In this paper, we investigated the applicability and fault-detection effectiveness of MT for three quantum query algorithms: Deutsch-Jozsa, Bernstein-Vazirani, and Simon. We proposed six MRs derived from the properties of these algorithms and introduced

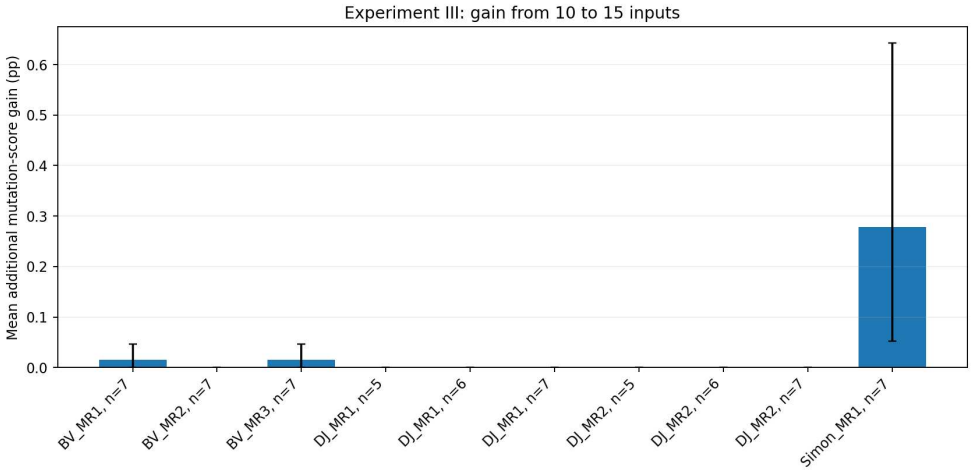


Fig. 6 Experiment III: Mutation-score gains obtained by increasing the inputs from 10 to 15.

Alg	MR	n	Mean MS (%)[1]				Seeds[2]
			5 inputs	10 inputs	15 inputs	20 inputs	
DJ	MR1	5	67.58 ± 0.00	67.58 ± 0.00	67.58 ± 0.00	67.58 ± 0.00	0/20
		6	70.71 ± 0.00	70.71 ± 0.00	70.71 ± 0.00	70.71 ± 0.00	0/20
		7	69.06 ± 0.00	69.06 ± 0.00	69.06 ± 0.00	69.06 ± 0.00	0/20
	MR2	5	67.58 ± 0.00	67.58 ± 0.00	67.58 ± 0.00	67.58 ± 0.00	0/20
		6	70.71 ± 0.00	70.71 ± 0.00	70.71 ± 0.00	70.71 ± 0.00	0/20
		7	69.06 ± 0.00	69.06 ± 0.00	69.06 ± 0.00	69.06 ± 0.00	0/20
BV	MR1	7	76.28 ± 0.14	76.55 ± 0.07	76.56 ± 0.00	76.56 ± 0.00	1/20
	MR2		77.80 ± 0.07	77.81 ± 0.00	77.81 ± 0.00	77.81 ± 0.00	0/20
	MR3		76.28 ± 0.14	76.55 ± 0.07	76.56 ± 0.00	76.56 ± 0.00	1/20
Simon	MR1	7	52.99 ± 2.03	54.17 ± 0.81	54.51 ± 0.00	54.51 ± 0.00	6/20

[1] Mean ± SD across 20 random seeds.
[2] Number of seeds detecting additional mutants between 10 and 20 inputs. No further detections were observed beyond 15 inputs.

Table 8 Experiment III: Mutation-score evolution for larger configurations without exhaustive input evaluation.

a general testing structure to translate them into executable tests. By integrating this structure with QCRMut, we automated the generation of mutants and the execution and evaluation of the resulting tests. Although the identification and definition of MRs remain algorithm-specific tasks, the proposed framework automates their application once they have been defined. Furthermore, the proposed BV MRs combine the outputs within the quantum circuit before measurement, allowing their relations

to be checked deterministically under ideal execution and minimising dependence on statistical comparisons of measurement outcomes.

The experimental evaluation first verified that the proposed MRs were satisfied by the original programs across the evaluated input spaces and assessed their fault-detection effectiveness through mutation testing, RQ1. This represents the first systematic evaluation of the fault-detection capability of MRs for quantum algorithms using automatically generated mutants. The results showed that all six MRs detected faults, with mutation scores ranging from 69.92% to 83.59% for DJ and BV, and from 41.80% to 60.16% for Simon. Although these results demonstrate their fault-detection capability, the presence of potentially equivalent mutants must be considered when interpreting the mutation scores.

We then investigated whether the exhaustive mutation scores could be reproduced using fewer inputs, RQ2. Across 20 random selections per configuration, the results showed that a relatively small number of inputs was sufficient to detect all mutants identified as killable through exhaustive evaluation. DJ consistently required three inputs, while the BV MRs required a median of between two and four inputs. Simon generally required more inputs and exhibited greater variability across random selections. Nevertheless, all evaluated selections reached their corresponding exhaustive mutation scores within 15 inputs, demonstrating the potential for substantial input reduction without compromising the observed fault-detection effectiveness. This reduction is particularly relevant to the computational cost of testing, as evaluating fewer inputs can reduce the number of required test executions and, consequently, the computational resources needed for simulation.

Finally, we extended the evaluation to larger configurations without calculating exhaustive reference scores, RQ3. The results showed that mutation scores tended to stabilise as additional inputs were considered, with no further mutant detections observed beyond 15 inputs within the evaluated range. However, some configurations, particularly Simon, required more inputs before reaching an apparent plateau. Since exhaustive evaluation was not performed for these configurations, the observed stabilisation cannot guarantee that all killable mutants have been detected.

Overall, the results support MT as a promising approach for testing quantum query programs, providing a systematic assessment of the fault-detection capability of the proposed MRs using automatically generated mutants. The possibility of verifying certain relations before measurement and reducing the number of inputs required also offers opportunities to mitigate the computational costs associated with quantum software testing. Nevertheless, the effectiveness of these reductions depends on the algorithm, the proposed MR, and the input selection.

Future work will focus on extending the proposed approach to additional quantum algorithms, particularly those used in practical applications. Since deriving MRs requires analysing the specific properties of each algorithm, identifying suitable relations remains an important research challenge. We also intend to investigate strategies for selecting inputs more effectively, reducing the number required to detect faults without relying on exhaustive reference scores. Another direction concerns the identification of equivalent mutants, which would allow a more accurate assessment of fault-detection effectiveness. Finally, we plan to evaluate the proposed approach under

realistic noise conditions, initially using noisy simulators and eventually executing tests on real quantum hardware to investigate its behaviour and computational cost in practical environments.

Declarations

Funding: This work was supported by a predoctoral contract funded by the Universidad Complutense de Madrid and Banco Santander (CT25/24), the State Research Agency (AEI) of the Spanish Ministry of Science and Innovation under grant PID2025-168053OB-I00 (NEXUS), and the Comunidad de Madrid under grant TEC-2024/COM-235 (DESAFíO-CM).

Conflict of interest: The authors declare that they have no competing interests.

Data availability: Data generated or analysed during this study are included in this published article.

Code availability: The source code and artifacts used in this study are included in this published article.

Author contribution: All authors contributed to the study conception, design, implementation, analysis, and manuscript preparation.

References

[1] Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M.: Mutation testing advances: an analysis and survey. vol. 112, pp. 275–378. Elsevier (2019). https://doi.org/10.1016/bs.adcom.2018.03.015

[2] Chen, T.Y., Kuo, F.-C., Liu, H., Poon, P.-L., Towey, D., Tse, T., Zhou, Z.Q.: Metamorphic testing: A review of challenges and opportunities. ACM Computing Surveys (CSUR) **51**(1), 1–27 (2018) https://doi.org/10.1145/3143561

[3] Almendros-Jiménez, J.M., Becerra-Terón, A., Merayo, M.G., Núñez, M.: Metamorphic testing of openstreetmap. Information and Software Technology **138**, 106631 (2021) https://doi.org/10.1016/j.infsof.2021.106631

[4] Méndez, M., Benito-Parejo, M., Ibias, A., Núñez, M.: Metamorphic testing of chess engines. Information and Software Technology **162**, 107263 (2023) https://doi.org/10.1016/j.infsof.2023.107263

[5] Ying, M.: Floyd–Hoare logic for quantum programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **33**(6), 1–49 (2012) https://doi.org/10.1145/2049706.2049708

[6] Huang, Y., Martonosi, M.: Statistical assertions for validating patterns and finding bugs in quantum programs. In: Proceedings of the 46th International Symposium on Computer Architecture, pp. 541–553 (2019). https://doi.org/10.1145/3307650.3322213

[7] Zhou, H., Byrd, G.T.: Quantum circuits for dynamic runtime assertions in quantum computation. IEEE Computer Architecture Letters **18**(2), 111–114 (2019) https://doi.org/10.1109/LCA.2019.2935049

[8] Liu, J., Byrd, G.T., Zhou, H.: Quantum circuits for dynamic runtime assertions in quantum computation. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 1017–1030 (2020). https://doi.org/10.1145/3373376.3378488

[9] Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., Xie, Y.: Projection-based runtime assertions for testing and debugging quantum programs. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–29 (2020) https://doi.org/10.1145/3428218

[10] Honarvar, S., Mousavi, M.R., Nagarajan, R.: Property-based testing of quantum programs in Q#. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pp. 430–435 (2020). https://doi.org/10.1145/3387940.3391459

[11] Ali, S., Arcaini, P., Wang, X., Yue, T.: 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), pp. 13–23 (2021). https://doi.org/10.1109/ICST49551.2021.00014 . IEEE

[12] Wang, X., Arcaini, P., Yue, T., Ali, S.: Quito: a coverage-guided test generator for quantum programs. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1237–1241 (2021). https://doi.org/10.1109/ASE51524.2021.9678798 . IEEE

[13] Wang, X., Arcaini, P., Ali, S., Yue, T.: Application of combinatorial testing to quantum programs. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 179–188 (2021). https://doi.org/10.1109/QRS54544.2021.00029 . IEEE

[14] Metwalli, S.A., Van Meter, R.: A tool for debugging quantum circuits. In: 2022 IEEE International Conference on Quantum Computing and Engineering (QCE), pp. 624–634 (2022). https://doi.org/10.1109/QCE53715.2022.00085 . IEEE

[15] Chen, Y.-F., Chung, K.-M., Lengál, O., Lin, J.-A., Tsai, W.-L.: Autoq: An automata-based quantum circuit verifier. In: International Conference on Computer Aided Verification, pp. 139–153 (2023). https://doi.org/10.1007/978-3-031-37709-9_7 . Springer

[16] Zhao, P., Wu, X., Li, Z., Zhao, J.: Qchecker: Detecting bugs in quantum programs via static analysis. In: 2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE), pp. 50–57 (2023). https://doi.org/10.1109/Q-SE59154.2023.00014 . IEEE

[17] Paltenghi, M., Pradel, M.: Analyzing quantum programs with lintq: A static analysis framework for qiskit. Proceedings of the ACM on Software Engineering **1**(FSE), 2144–2166 (2024) https://doi.org/10.1145/3660802

[18] Shang, K., Ishibuchi, H., He, L., Pang, L.M.: A survey on the hypervolume indicator in evolutionary multiobjective optimization. IEEE Transactions on Evolutionary Computation **25**(1), 1–20 (2020) https://doi.org/10.1109/TEVC.2020.3013290

[19] Serrano, M.A., Perez-Castillo, R., Piattini, M.: Quantum software engineering (2022) https://doi.org/10.1007/978-3-031-05324-5

[20] Barrera, A., Guzmán, I., Polo, M., Piattini, M.: Quantum software testing: State of the art. Journal of Software: Evolution and Process **35**(4), 2419 (2023) https://doi.org/10.1002/smr.2419

[21] Leite Ramalho, N.C., Souza, H., Lordello Chaim, M.: Testing and debugging quantum programs: The road to 2030. ACM Transactions on Software Engineering and Methodology **34**(5), 1–46 (2025) https://doi.org/10.1145/3715106

[22] Wang, J., Zhang, Q., Xu, G.H., Kim, M.: Qdiff: Differential testing of quantum software stacks. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 692–704 (2021). https://doi.org/10.1109/ASE51524.2021.9678792 . IEEE

[23] Wang, X., Yu, T., Arcaini, P., Yue, T., Ali, S.: Mutation-based test generation for quantum programs with multi-objective search. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1345–1353 (2022). https://doi.org/10.1145/3512290.3528869

[24] Mendiluze, E., Ali, S., Arcaini, P., Yue, T.: Muskit: A mutation analysis tool for quantum software testing. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1266–1270 (2021). https://doi.org/10.1109/ASE51524.2021.9678563 . IEEE

[25] Fortunato, D., Campos, J., Abreu, R.: Mutation testing of quantum programs: A case study with qiskit. IEEE Transactions on Quantum Engineering **3**, 1–17 (2022) https://doi.org/10.1109/TQE.2022.3195061

[26] García-Gil, S., Llana, L., Requeno Jarabo, J.I.: QCRMut: Quantum circuit random mutant generator tool. Available at SSRN 6211638 (2026) https://doi.org/10.2139/ssrn.6211638

[27] Paltenghi, M., Pradel, M.: Morphq: Metamorphic testing of the qiskit quantum computing platform. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 2413–2424 (2023). https://doi.org/10.1109/ICSE48619.2023.00202 . IEEE

[28] Abreu, R., Fernandes, J.P., Llana, L., Tavares, G.: Metamorphic testing of oracle quantum programs. In: Proceedings of the 3rd International Workshop on Quantum Software Engineering, pp. 16–23 (2022). https://doi.org/10.1145/3528230.3529189

[29] Costa, N., Fernandes, J.P., Abreu, R.: Asserting the correctness of shor implementations using metamorphic testing. In: Proceedings of the 1st International Workshop on Quantum Programming for Software Engineering, pp. 32–36 (2022). https://doi.org/10.1145/3549036.3562062

[30] Deutsch, D., Jozsa, R.: Rapid solution of problems by quantum computation. Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences **439**(1907), 553–558 (1992) https://doi.org/10.1098/rspa.1992.0167

[31] Bernstein, E., Vazirani, U.: Quantum complexity theory. SIAM Journal on Computing **26**, 1411–1473 (1997) https://doi.org/10.1137/S0097539796300921

[32] Yanofsky, N.S., Mannucci, M.A.: Quantum computing for computer scientists (2008) https://doi.org/10.1017/cbo9780511813887

[33] Simon, D.R.: On the power of quantum computation. SIAM journal on computing **26**(5), 1474–1483 (1997) https://doi.org/10.1137/S0097539796298637

[34] authors: Qiskit Textbook. Github (2023). https://github.com/Qiskit/textbook

[35] Flajolet, P., Gardy, D., Thimonier, L.: Birthday paradox, coupon collectors, caching algorithms and self-organizing search. Discrete Applied Mathematics **39**(3), 207–229 (1992) https://doi.org/10.1016/0166-218X(92)90177-C

[36] Barrera Amo, A., Serrano, M.A., Guzmán, I., Polo, M., Piattini, M.: Automatic generation of test circuits for the verification of quantum deterministic algorithms. In: Proceedings of the 1st International Workshop on Quantum Programming for Software Engineering, pp. 1–6 (2022). https://doi.org/10.1145/3549036.356205