QDiff: Differential Testing of Quantum Software Stacks

Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, Miryung Kim
University of California, Los Angeles
CA, United States
{wangjiyuan, zhangqian, harryxu, miryung}@cs.ucla.edu

***Abstract*—Over the past few years, several quantum software stacks (QSS) have been developed in response to rapid hardware advances in quantum computing. A QSS includes a quantum programming language, an optimizing compiler that translates a quantum algorithm written in a high-level language into quantum gate instructions, a quantum simulator that emulates these instructions on a classical device, and a software controller that sends analog signals to a very expensive quantum hardware based on quantum circuits. In comparison to traditional compilers and architecture simulators, QSSes are difficult to tests due to the probabilistic nature of results, the lack of clear hardware specifications, and quantum programming complexity.**

This work devises a novel *differential testing* approach for QSSes, named QDIFF with three major innovations: (1) We generate input programs to be tested via semantics-preserving, source to source transformation to explore program variants. (2) We speed up differential testing by filtering out quantum circuits that are not worthwhile to execute on quantum hardware by analyzing static characteristics such as a circuit depth, 2-gate operations, gate error rates, and T1 relaxation time. (3) We design an extensible equivalence checking mechanism via distribution comparison functions such as Kolmogorov–Smirnov test and cross entropy.

We evaluate QDIFF with three widely-used open source QSSes: Qiskit from IBM, Cirq from Google, and Pyquil from Rigetti. By running QDIFF on both real hardware and quantum simulators, we found several critical bugs revealing potential instabilities in these platforms. QDIFF's source transformation is effective in producing semantically equivalent yet not-identical circuits (i.e., 34% of trials), and its filtering mechanism can speed up differential testing by 66%.

I. INTRODUCTION

Quantum computing is an emerging computing paradigm that promises unique advantages over classical computing. However, quantum programming is challenging. Quantum computation logic is expressed in qubits and quantum gates, and the states of quantum registers are measured probabilistically. Due to the physical properties of quantum mechanics, qubits and quantum gates are fundamentally different from classical bits and gate logic. For example, *quantum indeterminacy* dictates that the same quantum program can produce different results in different executions.

To this end, many quantum software stacks such as Google's Cirq [1], Rigetti's Pyquil [2], and IBM's Qiskit [3] have been developed to provide user-friendly high-level languages for quantum programming, abstracting away the underlying physical and mathematical intricacies.

A quantum software track (QSS) includes (1) APIs and language constructs to express quantum algorithms, (2) a compiler that transforms and optimizes a given input quantum algorithm at the circuit level, and (3) a backend executor that either simulates the resulting gates on classical devices or executes directly on quantum hardware. Consider Qiskit [4], which consists of four components: `QiskitTerra` that compiles and optimizes programs, `QiskitAer` that supports high-performance noisy simulations, `QiskitIgnis` for error correction, noise characterization, and hardware verification, and `QiskitAqua` that helps a developer to express a quantum algorithm or an application. Other QSSes [2] such as Pyquil have a similar set of components—*e.g.*, Pyquil uses `quilc` as a compiler and `qvm` as a quantum simulator.

As with any compiler framework, a QSS could be error-prone. Developers and users often report bugs on popular QSSes [3], [5], [6], and a simple search on StackOverflow with the keyword "quantum error" would bring up over 500 posts on various QSS components, ranging from compiler settings, simulation, and the actual hardware [7]. These posts often reveal deeper confusion that developers face due to inherent probabilistic nature of quantum measurements—*if a program produces a result that looks different from what is expected, is it due to a bug or the non-determinism inherent in quantum programs? Is there divergence beyond expected noise coming from an input program, a compiler, a simulator, and/or hardware?*

Challenges. There are three technical challenges that make it difficult to test QSSes.

The first challenge is how to generate semantically equivalent programs for testing quantum compilers [8], [9]. Though existing compiler testing techniques generate equivalent programs by manipulating code in dead regions or unexecuted branches [10], a quantum program usually does not have many unexecuted branches. Therefore, we must design a technique that can produce a large number of semantically-equivalent variant circuits that may induce divergence on hardware.

The second challenge is that compilers are not the only source of bugs in a QSS. For example, in a line of recent work [11], [12] on verified quantum compilers, compilers are guaranteed to be correct with respect to their optimization and transformation rules. However, the overall correctness of

QSSes goes beyond compiler optimization rules. We must reason about subsequent execution on quantum simulators and hardware, which are often the major sources of instabilities. In fact, Rigetti or Cirq's Github histories indicate that bugs may come from its various backends due to improper initialization [13], problematic APIs [14], or inappropriate synthesis options [15].

The third challenge is how to interpret measurements from quantum simulators or hardware, since quantum measurement operations by definition are probabilistic in nature due to quantum indeterminacy [11]. Therefore, when we execute logically equivalent program variants, comparing their measurement results is not straightforward, as we must take into account inherent noise (e.g., hardware gate errors, readout errors, and decoherence errors) and must determine whether the observed divergence is significant enough to be considered as a meaningful instability.

This Work. QDiff is a novel differential testing technique for QSSes. QDiff takes as input a seed quantum program and reports potential bugs with a pair of *witness programs* (i.e., logically equivalent programs that are supposed to produce identical behavior) but triggers divergent behavior on the target QSS (i.e., meaningful instabilities beyond expected noise). QDiff overcomes the aforementioned challenges with three technical innovations below.

First, QDiff generates logically equivalent quantum programs using a set of *equivalent gate transformation* rules. This is based on the insight that each quantum gate can be represented mathematically as a unitary matrix; a sequence of gates essentially corresponds to the multiplication of their unitary matrices, which also yields a unitary matrix. As such, one sequence of gates is semantically equivalent to another if the two sequences yield the same unitary matrix. We leverage seven gate transformation rules to generate different gate sequences that are guaranteed to yield the same unitary matrix. These sequences essentially define logically equivalent program variants that are supposed to yield identical behavior. Such variants are then executed on a simulator or hardware and their measurements are compared.

Second, with multiple logically equivalent variants generated after optimization, QDiff selects a subset of those circuits that are worthwhile to run on a noisy simulator or quantum hardware. This selection and filtering process is crucial for both *speedup* and *potential cost saving*, because there are only a very few quantum hardware devices in the world, and they are extremely expensive resources—D-wave's quantum computer is reported to be \$15 million in 2017 [16]. IBM Qiskit allows public hardware access up to 5 qubits, but a higher number of qubit usage is strictly restricted. Noisy simulators are extremely compute-intensive to run on classical computers as well, taking on more than 4 minutes to do 3000 measurements on a simple four-qubit circuit. This noisy simulation cost becomes extremely unmanageable with the higher number of qubits.

To achieve speed up, QDiff analyzes the static characteristic of logically equivalent circuits such as the circuit depth (i.e., *moments*), the number of 2-qubit gates, the known error rates of each gate kind, and T1 relaxation time for quantum hardware. QDiff obviates the need of executing certain circuits on hardware or noisy simulators, if the logically equivalent circuits are incapable of revealing *meaningful instabilities* in QSS.

Third, in order to compare executions of logically equivalent circuits, QDiff first identifies how many measurements are needed for reliable comparison. Based on the literature of distribution sampling, QDiff adapts closeness testing in $L1$ norm [17] to estimate the required number of measurements given confidence level $p = 2/3$. Using either a noisy simulator or directly on hardware, it then performs the required number of measurements. To compare two sets of measurements, QDiff uses distribution comparison methods. Two methods based on Kolmogorov-Smirnov (K-S) test [18], [19] and Cross Entropy [20], [21] are currently supported. This comparison method is easily extensible in QDiff, as a user can define a new statistical comparison method [17] in QDiff.

Results. We evaluated QDiff with the latest versions of three widely-used QSSes: Qiskit, Cirq, and Pyquil. With six seed quantum algorithms, QDiff generates 730 variant algorithms through semantics-modifying mutations. Starting from the generated algorithms, it generates a total of 14799 program variants using semantics-preserving source transformations. This generation process took QDiff 14 hours. With the filtering mechanism, QDiff reduces its testing time by 66%. Using QDiff, we determined total 6 sources of instabilities. These include 4 software crash bugs in Pyquil and Cirq simulation, and 2 potential root causes that may explain 25 out of 29 cases of divergence beyond expected noise on IBM hardware.

The main contributions of this work include:

- We present the first differential testing framework for QSSes. This framework is integrated with three widely used QSSes from Google, IBM, and Rigetti.
- We embody three technical innovations in QDiff to make differential testing fast and to increase the chance of finding meaningful instabilities beyond the usual expected noise: (1) auto-generation of logically equivalent variants, (2) auto-filtering of circuits that are worthwhile to run on hardware (or a noisy simulator), and (3) pluggable methods of comparing measurements in terms of statistical distribution. QDiff obviates the need of invoking quantum back-ends by 66%. Using QDiff, we determined 6 sources of instabilities in three widely used QSSes: 4 crash bugs and 2 root causes for divergences beyond expected noise on IBM hardware.

We provide access to the artifacts of QDiff at https://github.com/wjy99-c/QDiff.

II. Background

Quantum Computing. Quantum computing emerges as a promising technology for many domains where quantum computers have been demonstrated to outperform classical com-

TABLE I: Different layers that bugs appear

Layers	Percentage	Example
Compiler (optimizations & settings)	53.9%	`MergeInteractions()` returns different results for the same circuit [22].
Backend (simulators & hardware)	19.7%	Simulators change a global random state on Cirq [23].
API and quantum gate	11.8%	PhasedXPowGate raised to a symbol power fails the is_parameterized protocol [24].

puters by a large margin. For example, Grover's algorithm [25] can find a given item in an unsorted database with a $\sqrt{N}$-times speedup when running on a quantum computer, compared to a classical computer. Similar to programs executed on heterogeneous devices such as FPGAs, a quantum application is comprised of *host code* that runs on CPU and *quantum code* that runs on quantum hardware. Generally, quantum computers work as accelerators to execute the compute-intensive parts of the original application. For example, the Shor algorithm [26] can achieve an exponential speedup by decomposing integer factorization into a reduction to be executed on a classical computer and an order finding problem to be executed on quantum hardware.

Quantum Software Stack. Currently, three most widely used QSSes are Qiskit, Cirq, and Pyquil [27].

- **Qiskit [3]** is an open-source framework developed by IBM. Qiskit provides a software stack that is easy to use quantum computers and facilitates quantum computing research. Qiskit consists of `Qiskit Terra` (compiler), `Qiskit Aer` (several quantum simulators), `Qiskit Ignis` (which supports error correction and noise characterization), and `Qiskit Aqua` (APIs to help developers write applications).
- **Cirq [28]** is an open-source Python framework from Google. It enables a developer to create and simulate Noisy Intermediate-Scale Quantum (NISQ) circuits. Cirq consists of an `optimization` component to compile and transform circuits, a `simulator` component for emulating quantum computation on classical devices, and other `development` libraries for application development.
- **Pyquil [29]** is an open-source Python framework developed by Rigetti. It builds on Quil, an open quantum instruction language for near-term quantum computers, and uses a combined classical/quantum memory model. PyQuil is the main component of Forest, the overarching platform for Rigetti's quantum software. Pyquil consists of: `quilc` (compiler), `qvm` (quantum virtual machine with several kinds of simulators), and `pyquil` (a library to help users write and run quantum applications).

All three QSSes are similar to one another in that each includes a quantum programming language, an optimizing compiler that outputs quantum gate instructions, a quantum simulator that emulates these instructions on a classical device, and a software controller that executes gate instructions on quantum hardware.

Quantum Bit. A *quantum bit*, or *qubit* for short, is the basic unit of quantum computation. Unlike a classical bit that is either 0 or 1, a qubit's state is a probabilistic function of $|0\rangle$ and $|1\rangle$, represented as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \text{where } |\alpha|^2 + |\beta|^2 = 1 \tag{1}$$

The state of a qubit is unknown until a *measurement* is completed, resulting in $|\alpha|^2$ to be 0 and $|\beta|^2$ to be 1. Naturally, the sum of the probabilities (i.e., the modulus squared of amplitudes) is 1: $|\alpha|^2 + |\beta|^2 = 1$.

Quantum Gate. Classical computers use logic gates to transform signals from input wires. For example, a *NOT* gate, also known as an inverter, takes a single signal as input and outputs the opposite value. The quantum gate analogous to *NOT* is an *X* gate, which transforms a qubit $\alpha|0\rangle + \beta|1\rangle$ to $\beta|0\rangle + \alpha|1\rangle$.

An *X* gate has the following matrix-based representation:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2}$$

Using a vector to represent the quantum state $\alpha|0\rangle + \beta|1\rangle$, applying an *X* gate has the following effect:

$$X\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \begin{bmatrix} \alpha' \\ \beta' \end{bmatrix} \tag{3}$$

Other commonly-used quantum gates include H, T, $CNOT$, Z, *CZ*, S, *U1*, and *U3*; a full explanation of these gates is elsewhere [30].

Since all quantum gates can be represented as *matrices*, we can transform a sequence of gates to another logically equivalent sequence without altering its outcome, as long as *the multiplication of the matrices for gates in each sequence* produces the same result. As an example, a *SWAP* gate, a two-qubit gate that swaps the two qubits' states, is semantically equivalent to a sequence of three *CNOT* gates.

$$\mathit{SWAP}(q_1, q_2) = \mathit{CNOT}(q_1, q_2)\mathit{CNOT}(q_2, q_1)\mathit{CNOT}(q_1, q_2) \tag{4}$$

This observation forms the foundation of QDIFF's program variant generation procedure, detailed in Section IV.

Quantum Circuit. A quantum circuit consists of a set of connected quantum gates. Since quantum gates can be represented as unitary matrices, a quantum circuit is essentially the multiplication of the matrices.

Figure 1a shows a program in IBM's Qiskit [3]. It first registers two qubits and initializes them to $|0\rangle$. Next, an *H* gate sets the first qubit into state $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and an *X* gate flips the second qubit from $|0\rangle$ to $|1\rangle$. Finally, a measurement is performed and stored in a classical array. The function returns this circuit as a function-type value.

Figure 1b shows host code that calls this quantum circuit. Users can execute this circuit on different backends such as real quantum hardware or simulators. The circuit is executed on *qasm_simulator* for a thousand times (`shots=1000`). Since the first qubit state is $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and the second qubit state

```
def make_circuit() -> QuantumCircuit:
    qubit = QuantumRegister(2,"qc")
    bit = ClassicalRegister(2, "qm")
    prog = QuantumCircuit(qubit, bit)
    prog.h(qubit[0])
    prog.x(qubit[1])
    for i in range(2):
        prog.measure(qubit[i], bit[i])
    return prog
```

(a) A quantum circuit with an H gate and an X gate.

```
    prog = make_circuit()
    backend = BasicAer.get_backend('qasm_simulator')
    info = execute(prog, backend=backend, shots=1000).
          result().get_counts()
```

(b) The host code using the circuit in (a).

Fig. 1: An example Qiskit program.

is $|1\rangle$, each run produces a result of either $|01\rangle$ or $|11\rangle$ with the equal probability of $0.5 = (\frac{1}{\sqrt{2}})^2$.

T1 relaxation time. In quantum computing, a qubit can retain data for only a limited amount of time, referred to as *relaxation Time* because a qubit in a high-energy state (state $|1\rangle$) naturally decays to a low-energy state (state $|0\rangle$). The time span for this decay is referred to as *T1 Relaxation Time*. For a physical circuit, its measurement results are unreliable if its execution time is longer than *T1*.

III. Motivation

To understand real-world QSS bugs, we collected 76 latest issues reported on Github Pyquil, Cirq, and Qiskit. After excluding 11 issues related to installation and other tools, we cagegorized the remaining 65 issues by the layer where each issue appears: compilers, backends, and APIs.

Table I summarizes the percentage of the corresponding layer and a representative example. Most bugs appear at the compiler level—compiler optimizations and settings. For example, Cirq produces incorrect circuits when using `MergeInteractions()` [22]. The second most common bugs appear at the backend level—simulators and hardware execution, *e.g.*, the initial state is unexpectedly modified by the Cirq simulator in [23]. The other issues are with respect to the API implementation of high-level gates. For example, in Cirq, when `cirq.PhasedXPowGate` is invoked with an input argument `exponent`, invoking `is_parameterized` on the resulting gate should return `true` but returns `false` due to a bug in `cirq.PhasedXPowGate`.

The aforementioned bugs are hard to find due to quantum indeterminacy. The following excerpt illustrates a concrete example of how one google tutorial user is confused whether a problem is a bug or due to inherent non-determinism. Cirq's VQE (Variational-Quantum-Eigensolver) tutorial had a mistake—the orders of mix layer and cost layer were described in a wrong order. The tutorial user `D` (shortened) posted a question on StackExchange [31] with the embedded code from Cirq's VQE tutorial. A Cirq developer `C` (shortened) noticed that "The strange thing is the example output shows the output probabilities varying with *gamma* (the CZ parameter)", where *gamma* should not have any effect on the measurement results." `D` actually believed the disagreement is due to quantum indeterminacy, until `C` explicitly labeled it as a compiler bug.

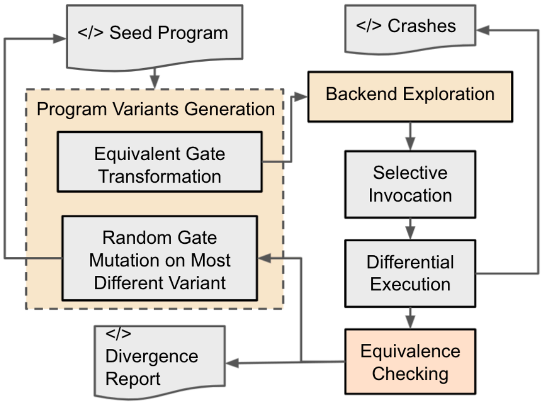


Fig. 2: QDiff Overview

Key Takeway. This study of Github issues shows that QSS bugs are not just traditional compiler bugs and may come from various sources at different layers: API implementation of high-level gates, backend simulators, or hardware execution. This problem of detecting QSS bugs is further complicated by the probabilistic and noisy nature of quantum indeterminacy.

IV. QDiff Approach

QDiff contains three novel components to detect meaningful instabilities in QSSes. Figure 2 shows *program variant generation* using equivalent gate transformation and mutations (Section IV-A); *backend exploration* leveraging selective invocation of quantum simulators and hardware (Section IV-B); and *equivalence checking* via distribution comparison (Section IV-C). Starting from a seed program, QDiff iterates these steps until a time limit is reached. Our key insight is that (1) we can generate semantically equivalent but syntactically different circuits, and (2) we can speed-up the differential execution process by filtering out certain circuits, because they will definitely lead to unreliable divergence and thus are *not worthwhile to run* on hardware or noisy simulators.

A. Program Variant Generation

Prior work [10] finds that testing compilers with equivalent programs is highly effective. QDiff adapts this idea to the domain of quantum compiler testing by creating logically equivalent gate sequences. It then checks whether the corresponding equivalent circuits produce the same results (in this case, a similar statistical distribution with some noise) on quantum simulators or hardware. For this purpose, QDiff generates program variants by repeating the two-fold process of *applying semantics-preserving gate transformation* to each generated program in each iteration and *applying semantics-modifying mutations* to diversify the pool of input programs in the next iteration.

Equivalent Gate Transformation (EGT). As discussed earlier

TABLE II: Explored Compiler Configurations

Framework	Options	Description
Qiskit	`BasicSwap, LookaheadSwap, StochasticSwap`	Specify how swaps should be inserted to make the circuit compatible with the coupling map. QDiff checks if BasicSwap has the most $SWAP$ gates.
	`Optimization_level=0,1,2,3`	Specify the optimization level—the higher the level, the simpler the resulting circuit. QDiff checks if a higher-level optimization generates a more complex circuit.
Cirq	`DropEmptyMoment()`	Remove empty moments from a circuit. QDiff checks empty moments in the circuit.
	`MergeInteractions()`	Merge adjacent gates; QDiff checks the applicability of G6 in Table III.
	`PointOptimizationSummary()`	User-defined optimization.
Pyquil	`PRAGMA INITIAL_REWIRING {"NAIVE" ,"GREEDY", "PARITIAL"}`	Change the optimization/mapping style
	`PRAGMA {COMMUTING_BLOCKS ,PERSERVE_BLOCKS}`	Change the optimization style for certain part of the code.

TABLE III: QDiff generates program variants based on EGT rules G1-G7.

Rule ID	Original Construct	Equivalent Construct

G1	SWAP(q_1,q_2)	CNOT(q_1,q_2)
		CNOT(q_2,q_1)
		CNOT(q_1,q_2)

G2	H(q_1)H(q_1)	Merged to Identity Matrix
G3	X(q_1)	H(q_1)S(q_1)S(q_1)H(q_1)
G4	Z(q_1)	S(q_1)S(q_1)
G5	CZ(q_1,q_2)	H(q_2)CNOT(q_1.q_2)H(q_2)
G6	CZ(q_1,q_2)CZ(q_1,q_2)	Merged to Identity Matrix
G7	CCNOT(q_1,q_2,q_3)	6 CNOT gates with 9 one-qubit gates

in Section II, one quantum gate sequence is semantically equivalent to another sequence, if they both yield the same unitary-matrix representation. In fact, complex quantum gates, without altering the outcome, can be described as a combination of basic quantum gates. QDiff leverages seven gate transformation rules to map complex gates into sequences of simple gates, as shown in Table III. In G7, a Toffoli gate (*i.e.* $CCNOT$) can be replaced with 6 $CNOT$ gates and 9 one-qubit gates. To explore the alternative representation of a given gate sequence, QDiff finds the applicable transformation rules by matching the gate names. For an example circuit $S(q2)Z(q1)$, QDiff identifies the complex gate $Z(q1)$, applies rule G2 to construct gate $Z(q1)$ as a sequence $S(q1)S(q1)$, and generates a final variant $S(q2)S(q1)S(q1)$. As opposed to an optimizing compiler that applies transformation rules to reduce the number gates or the total gate depth [3], [11], QDiff aims to diversify the pool of input programs through source to source transformation.

Seed Diversification via Mutations. While generation of logically equivalent programs can find bugs through observing disagreements, they are unlikely to exercise various programming constructs. To diversify the seed input programs and to explore hard-to-reach corner cases in quantum compilers. QDiff borrows the idea of mutation-based fuzz testing [32] and designs a set of semantics-modifying mutations.

After generating multiple logically equivalent programs in each iteration, QDiff calculates the average distributions among the equivalent programs as the reference distribution, then picks the variant that leads to the largest comparison distance (e.g., K-S distance) from the reference distribution, and randomly applies one of the following four mutation operations to the variant in order to generate a new algorithm. This is based on the insight that the program with the most deviating results has a higher chance to expose unseen behavior [10], [33]. Please note that these mutations do not preserve semantics; instead, the goal of mutations is to resume the next round of differential testing with a different algorithm. We start with four mutation operators listed below, used in quantum mutant generation [34], [35].

- **Gate Insertion/ Deletion (M1)** inserts/deletes random quantum gates: *e.g.*, insert `prog.x(qubit[1])`;
- **Gate Change (M2)** changes a quantum gate to another gate: *e.g.*, from `prog.x(qubit[1])` to `prog.h(qubit[1])`;
- **Gate Swap (M3)** swaps two quantum gates;
- **Qubit Change (M4)** changes the qubits: *e.g.*, from `prog.x(qubit[1])` to `prog.x(qubit[2])`.

B. Quantum Simulation and Hardware Execution

Compiler Configuration Exploration. QDiff automatically explores different compiler configurations. In Qiskit, compiler settings can be specified by the arguments passed to the backends e.g., users can apply `optimization level=1` to collapse adjacent gates via light-weight optimization, while `optimization level=3` does heavy-weight optimization to resynthesize two-qubit blocks in the circuit. In Cirq, compiler settings must be specified using API invocations: e.g., users can write their own optimization with `PointOptimizationSummary()`. In Pyquil, compiler settings are specified using inlined pragmas similar to how FPGA developers specify high level synthesis options using pre-processor directives, e.g., a region denoted by `PRAGMA PRESERVE_BLOCK` will not be modified by a compiler. There are in total 2, 3, and 2 configuration types for Qiskit, Cirq, and Pyquil respectively, as shown in Table II. When executing a variant with a specific compiler setting, QDiff records both thrown exceptions and program timeouts.

Backend Exploration. Backend exploration runs the same input program on different backends, shown in Table V. In terms of real hardware execution, QDiff uses the free version of IBM hardware only, because other platforms are currently proprietary. QDiff is extensible by specifying a different backend configuration. Noisy simulators and state-vector simulators are both included in QDiff's backend exploration.

Filtering and Selective Invocation on Hardware. QDiff is focused on isolating software defects, not hardware defects. Because hardware imperfections such as decoherence

TABLE IV: IBM quantum hardware's gate-level error rates, gate time, and T1 relaxation (decoherence) time.

IBM quantum computer	T1 time $/\mu s$	Gate time $/ns$	2-qubit gate error rate	t_m	δ_{2qubit}
ibm_santiago	155.19	408.89	1.79%	379	5
ibm_yorktown	56.81	476.44	2.03%	119	5
ibm_16_melbourne	53.37	928.71	3.29%	57	3
ibm_belem	74.45	552.89	1.07%	135	11
ibem_quito	85.35	353.78	1.03%	241	11

TABLE V: Explored Backends (Simulators and Hardware)

Framework	Backends	Description
Qiskit	`statevector_simulator`	noiseless sim.
	`qasm_simulator`	noiseless sim.

	`FakeSantiago`
	`FakeYorktown`
	`FakeMelbourne`
		noisy sim.

	`ibmq_santiago`
	`ibmq_yorktown`
	`ibmq_16_melbourne`
		quantum hardware

Cirq	`Simulator`	noiseless/noisy sim.
	`DensityMatrixSimulator`	noiseless/noisy sim.
Pyquil	`Aspen-x-yQ-noisy-qvm`	noisy sim.
	`WavefunctionSimulator`	noiseless sim.
	`Aspen-x-yQ-qvm,Pyqvm`	noiseless sim.

TABLE VI: Cumulative probability of KS test

Measurement	State		Cumulative	State	
Distribution	'0'	'1'	Probability	'0'	'1'
A_1	464	546	EDF_{A1}	0.464	1
A_2	500	500	EDF_{A2}	0.500	1

is present, it is important to filter out circuits that would invoke errors due to the inherent hardware limitations. With the above observations, QDiff filters out unnecessary circuits in two steps. First, QDiff examines the final gate sequences after all compiler optimizations and logical-to-physical mappings, filtering out exactly identical physical circuits by moment-by-moment comparison. Second, QDiff analyzes the static characteristics of circuits to remove those that certainly produce unreliable executions (i.e. results dominated by hardware-level noise such as gate errors and relaxation errors and hence unreliable), while leaving those that may produce *meaningful divergences* using Definition IV.1.

As discussed in Section II, for a physical circuit, its measurement results are unreliable if its execution time is longer than T1, implying that the number of circuit moments n_m (the depth of the circuit) in any circuit should not exceed a threshold t_m. Moreover, different kinds of quantum gates have different inherent error rates. For publicly available IBM quantum computers, error rates of single-qubit operations are in the order of 10^{-3}, while error rates of 2-qubit gate operations are in the order of 10^{-2}. A typical quantum program contains a significant number of 2-qubit gates, whose errors contribute the most to the overall error rate because such gates are more error-prone than 1-qubit gates [36]. Taking this into consideration, d_{2qubit} (the difference in the number of 2-qubit gates from the original circuit) should not exceed an application-specific threshold δ_{2qubit} to avoid unreliable results. Leveraging the above observations, we define the *worthiness* of invoking a quantum circuit.

Definition IV.1. *A circuit is worth invoking on quantum hardware or noisy simulator, if it satisfies the following condition:* $n_m < t_m$ *and* $d_{2qubit} < \delta_{2qubit}$.

The threshold t_m is determined empirically by two factors: (1) IBM computers' average $T1$ time, and (2) the average *gate* execution time for all gates, as listed in Table IV. QDiff computes t_m by dividing $T1$ by the average *gate* execution time. It then filters out those whose n_m is greater than t_m. Take `ibm_16_melbourne` as an example: with $T1$ = $53.57\mu s$ and the average gate execution time = $928ns$, t_m is 53570/928=57. A circuit whose total number of moments is above 57 is filtered out for `ibm_16_melbourne`.

The threshold δ_{2qubit} is determined in the following way. Suppose a user is willing to tolerate an addition error rate of t for the entire quantum program's final measurements (with 0.1 as the default). Using t, we compute δ_{2qubit} as the maximum number of 2-qubit gates to be added or deleted from the number of 2-qubit gates in the original circuit. Suppose that CNOT's error rate for this IBM computer is 1.07%. If CNOT is used d_{2qubit} times in a row additionally, its updated error rate would be by $1-(1-e)(1-0.0107)^{d_{2qubit}}$, where e is the original error rate. Since both e and $(1-0.0107)^{d_{2qubit}}$ are relatively small, we can regard the error rate change as $1-(1-0.0107)^{d_{2qubit}}$. Therefore $|d_{2qubit}-\delta_{2qubit}|$ should be less than $log_{(1-0.0107)}(1-t)$. δ_{2qubit} is 11 when t is 0.1. The above thresholds and filtering condition in Definition IV.1 are customizable according to hardware's published error rates, supported gate types, and T1 relaxation time.

C. Equivalence Checking via Distribution Comparison

Nondeterministic nature of quantum programs makes it difficult for equivalence checking. Developers usually reason about the output of a quantum circuit by executing it multiple times to obtain a distribution. While numerous distribution comparison methods are well studied in statistics, one consequent yet over-looked question for quantum computing is that *how many measurements do we need for a reliable evaluation to ensure the relative error between two distributions is within a given threshold* t *with confidence* p? We design a novel equivalence checking component, which consists of: (1) a particular distribution comparison method C, and (2) an estimation of the required number of measurements for C, given a threshold t and confidence p. QDiff is equipped with *K-S test* and *Cross Entropy*, but is also extensible to other comparison methods by providing a new comparison-specific measurement estimation.

K-S Test [18] has been used to check the equality of distributions by measuring the largest vertical distance between empirical distribution functions (EDFs) in two steps. First,

```
backend = BasicAer.get_backend('qasm_simulator')
info = execute(prog, backend=backend, shots=1024).result
    ().get_counts()
```

```
Result: {'0': 475, '1': 549}
```

(a) Quantum simulator.

```
backend = BasicAer.get_backend('statevector_simulator')
info = execute(prog, backend=backend).result().
    get_statevector()
```

```
Result: [0.70710678+0.j, 0.70710678+0.j]
```

(b) State-Vector simulator.

Fig. 3: A quantum circuit in Qiskit with two backends: quantum simulator and state-vector simulator.

it creates the EDF for a given distribution by calculating the cumulative probability of different outcome states with respect to the total number of samples. In Table VI, A_1 and A_2 are two state distributions for 1000 samples. A_1 has 464 samples in state '0' while 546 samples in state '1'. Thus, the consequent EDF_{A1} indicates that the cumulative probability of A_1 samples in state '0' and state '1' are 0.464 (464/1000) and 1 ((464+546)/1000) respectively. Next, K-S test calculates the largest distance D of EDFs for each state, and uses such distance to quantify the difference of the original distributions under comparison. In Table VI, the largest distance is 0.06 ($|0.464 - 0.500|$) for state '0'.

QDiff evaluates this K-S distance with a user-defined threshold t. If the K-S distance of two results is less than t, QDiff regards them as similar results. QDiff provides a statistical guarantee on this comparison by estimating the required number of samples. For two distributions $d1$ and $d2$ over m outcome states, prior work [17] theoretically ensures that, with $n = \Omega(m^{1/2} \cdot t^{-2})$ samples, a sample-optimal tester can check if the relative error of $L1$ distance is within a threshold t when using a default confidence level of p=2/3. QDiff estimates n using Equation 5. This estimated number is directly applicable to QDiff because K-S distance is bounded by $L1$ distance [18]. In other words, Equation 5 calculates the number of measurements required, parameterized with respect to p. We empirically set p as 2/3, as it is a commonly used default in quantum volume measurement [37], [38], bioinformatics, and other statistics comparison.

$$n = A \cdot \frac{1}{\sqrt{1-p}} \cdot m^{1/2} \cdot t^{-2} \tag{5}$$

where A is a platform-related constant and m is the number of qubit states. In Figure 3, 2828 measurement samples are needed, when we empirically set $t = 0.1$, $p = 2/3$, and $A = 12$ for Qiskit. In our evaluation, we empirically measure the constant A for each platform by repeatedly running the same programs and compare the results.

Cross Entropy measures the difference between distributions via the total entropy. It represents the average number of bits needed to encode data coming from an underlying distribution q_1 when we use an estimated target distribution q_2.

$$H(q_1, q_2) = -\sum_{x=0}^{max} q_1(x) \log q_2(x) \tag{6}$$

Prior work ensures that, with $n = \Omega(m^{2/3} \cdot t^{-4/3})$ samples, the expected cross entropy of two similar distributions over m outcome states can be bounded with t [17], [39]. t is the difference from $H(q_1, q_1)$. Similar to K-S test, QDiff estimates the number of required samples to reliably satisfy this bound, as shown in Equation 7.

$$n = A \cdot \frac{1}{\sqrt{1-p}} \cdot m^{2/3} \cdot t^{-4/3} \tag{7}$$

For Figure 3, we need 420 measurements with $t = 0.1$, $p = 2/3$, and $A = 7$ for Qiskit. This measurement trials are different from K-S test, because we are using different distance metrics.

Comparison with Reference Distribution. After generating a group of equivalent programs and filtering out worthless circuits, QDiff executes the remaining circuits and calculates the average distribution from their results. QDiff will regard this average distribution as the *reference distribution*. With the distribution comparison methods (eg. K-S Test, Cross-entropy, etc), QDiff compares this reference distribution with each result distribution and reports divergence when the distance is larger than the threshold t.

Reporting Divergence Explanation. QDiff reports the potential source of the divergence in program P:

1) If P finds divergence when using different backends while keeping a frontend's options unchanged, QDiff reports this as a potential backend source;
2) If P finds divergence when using a specific backend while varying a frontend's options, QDiff reports this as a potential frontend source;
3) Otherwise, QDiff reports this as other sources, such as a potential bug in the API gate implementation.

V. Evaluation

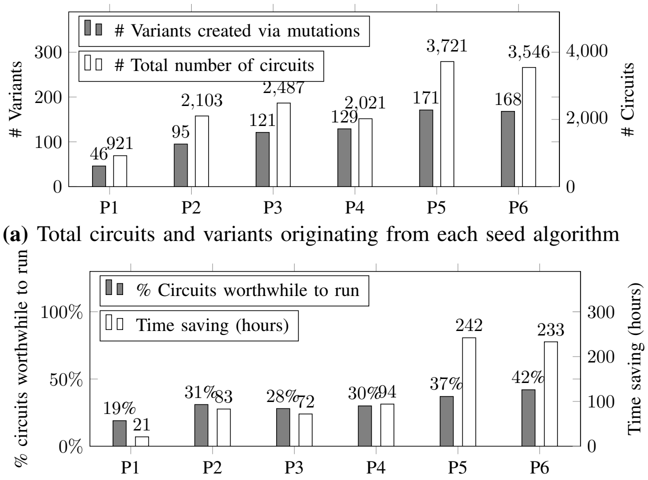
We evaluate following research questions:

RQ1 How many syntactically different programs can be generated by QDiff's mutation and equivalent gate transformation?

RQ2 How much speedup can we achieve via filtering and obviating the need of invoking a quantum simulator or hardware?

RQ3 What has QDiff found via differential testing of the widely-used QSSes?

Benchmarks. QDiff starts differential testing with five well known quantum algorithms as seed programs [25], [26], [40]–[42] (Deutsch-Jozsa, Berstein-Vazira, VQE–Variational-Quantum-Eigensolver, Grover, and QAOA–Quantum Approximate Optimization Algorithm) and one additional program X Gate listed in Table VII. We do not use the relatively large algorithms like Shor's [26] because IBM's public access can support up to 16 qubits only. Large algorithms also require



(a) Total circuits and variants originating from each seed algorithm

(b) The percentage of circuits that are worth running

Fig. 4: Statistics of QDiff-generated circuits.

many moments and 2-qubit gates, often producing unreliable results on quantum hardware.

Experimental Environment. We evaluate QDiff with three widely-used QSSes: Pyquil 2.19.0 with Quilc 1.19.0, Qiksit 0.21.0, and Cirq 0.9.0. We run the circuits on five different hardware versions based on their availability, including `ibmq_santiago`, `ibmq_yorktown`, `ibmq_16_melbourne`, `ibmq_belem`. and `ibmq_quito`. The details of hardware can be found on IBM's quantum computing website [3]. We use K-S test with $t = 0.1$ as the distribution comparison method.

A. RQ1 Variant Generation via S2S transformation

As shown in Figure 4a, for all six seed algorithms together, QDiff generates 730 program variants through semantics-modifying mutations and generates the total of 14799 circuits with equivalent gate transformation to each generated variant. This total circuit generation process takes around 14 hours.

Take P2 Deutsch-Jozsa algorithm as an example. 95 different program variants are generated through semantics-modifying mutations. For each variant program, QDiff generates 20 logically equivalent circuits. For P2, the total generation for 2103 circuits takes around 2 hours, while the rest of differential execution via simulation or hardware execution takes around 2 days. This implies that the bottleneck of testing is not about input program generation but the execution of the generated programs, which justifies our approach to select which circuits are worthwhile to run.

B. RQ2 Speed Up

As shown in Figure 4b, after filtering, only 19%-42% of the generated circuits are retained for differential execution on quantum hardware, leading to a 66% reductions in quantum hardware or noisy simulator invocations. QDiff finishes the entire testing process within around 17 days by leveraging its circuit selection process, which means QDiff would have saved additional 30 days of testing time by filtering.

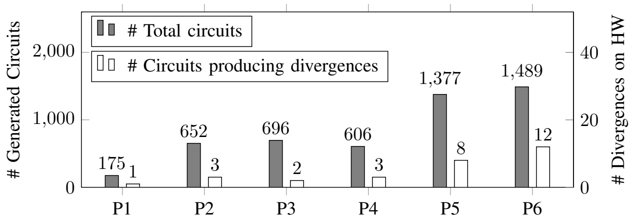


Fig. 5: Circuits producing divergences on IBM hardware

```
qc = get_qc('Aspen-0-3Q-
    A-qvm')
result_rm = qc.
    run_and_measure(p)
```

(a) `run_and_measure` measures all 16 qubits in device `Aspen-0` when a simulator allocates 3 qubits, resulting in an exception.

```
p = Program(X(0), X(1).
    controlled(0))
qvm = PyQVM(n_qubits=2)
qvm.execute(p)
```

(b) Controlled X gate raises an error when 2 qubits are allocated for simulation.

Fig. 6: Bugs in Pyquil found by QDiff.

Take QAOA as an example, when running on IBM quantum hardware, the experiment would take around 7 minutes to wait in line on average (as launching a quantum job uses a shared web service for a few IBM computers in the world) and 10 seconds to execute on hardware. If users run all 3546 circuits, the total clock time would be 17 days. With filtering, QDiff removes 68% circuits that are not worthwhile to run and finishes the execution in 7 days.

C. RQ3: What has QDiff *found?*

From the 730 sets of semantically equivalent circuits, QDiff found 33 differing outcomes out of 730. 4 out of 33 are crashes in simulators. The remaining 29 cases are divergence beyond expected noise on IBM hardware. By inspecting all 33 cases carefully, we determined total 6 sources of instabilities: 4 simulator crashes and 2 potential root causes that may explain 25 out of 29 cases of divergence on IBM hardware. For the remaining 4 divergence cases, we could not easily determine their underlying root causes.

1) Crash bugs in simulators: QDiff reports four crashes during differential testing with both noiseless and noisy simulators. All divergences involve clear failure signals. All are due to bugs in compiler or simulator implementations, summarized in Table VIII. 2 out of 4 crashes were already reported by developers [43], [44] and 2 out of 4 crashes were confirmed by developers, when we filed crash reports [45], [46].

Compiler option error: Given an arbitrary program in Cirq, when the compiler option `clear_span` is set to a negative number or `clear_qubits` is set to an unregistered qubit, the execution does not terminate. Cirq does not check the boundary values of compiler options and attempts to reset non-existing qubits. This bug was detected during explorations of compiler settings. When these compiler options are set to the aforementioned values, QDiff notices that the execution does not finish in a reasonable time, indicating a potential infinite loop. QDiff found this bug on the 74^{th} iteration with P3.

Backend registers a wrong number of qubits: QDiff found that Pyquil's measurement crashed on `Aspen-0-xQ-A-qvm`.

TABLE VII: Seed subject programs

ID	Program	# of Qubits	Moments	2-qubit gate	Description	Iteration Number	Measurement trial with $t = 0.1$
P1	X gate	1	1	0	one-qubit X gate	46	2000∼2828
P2	Deutsch-Jozsa	4	39	33	check if a function is balanced	95	5293∼7998
P3	Bernstein-Vazira	4	41	32	find a and b for f(x) = ax+b	121	5293∼7998
P4	Grover	5	84	53	find a unique input in a database	129	8000∼11312
P5	VQE	4	36	28	approximate the lowest energy level	171	5293∼7998
P6	QAOA	5	29	19	QAOA algorithm	168	8000∼11312

TABLE VIII: Bugs found by QDIFF when executing generated programs with simulation only

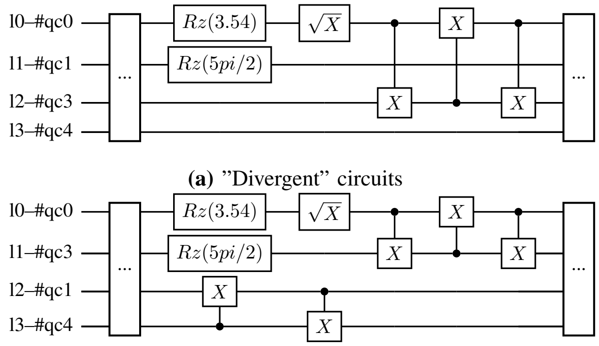
Platform	Bugs Description	Source of Bugs
Cirq	Program runs endlessly with some compiler settings	Compiler Setting
pyquil	Simulator register wrong number of qubits	Simulator Backends
	Crashes on control gates on certain backends	Gate implementation
	Simulator stuck into a bad state	Simulator Backends

In Pyquil, users can specify the quantum simulator to have the same topology as a real 16 qubit device `Aspen-0-16Q-A` by setting the backend to be `Aspen-0-16Q-A-qvm` (`16Q` refers to using 16 qubits in simulation). However, QDIFF found that when a user allocates to use 3 qubits in simulation (line 1 in Figure 6a) but attempts to conduct measurements at line 2 using `run_and_measure`, Pyquil simulator crashes due to a wrong number of allocated qubits. This bug was found with QDIFF's backend exploration. Another user reported the same issue on Pyquil's Github [43].

Simulator is stuck into a bad state: Pyquil's simulator raises an exception when taking an empty circuit as input. Afterward, all subsequent invocations to the simulator crash even when the input circuit is valid and not empty. QDIFF detects this bug by mutating an input program to an empty program and executing it on both `pyqvm` simulator and state-vector simulator. While the state-vector simulator runs this empty circuit normally, `pyqvm` throws an exception. Then, QDIFF generates an arbitrary non-empty program. In the next few iterations, QDIFF finds this bug because `pyqvm` crashes, while the state-vector simulator does not.

Wrong type of the controlled gate: Figure 6b shows a scenario where Pyquil crashed when a control X gate was used on `pyqvm` with 2 qubit allocation. The exception message shows *"ValueError: cannot reshape array of size 4 into shape (2,2,2,2)"*. A control X gate should be a 2-qubit gate (which is 4×4 matrix), but the gate was represented as 1×4 by `pyqvm`, which is a bug. This bug was found in the 15^{th} iteration with P1 as a seed.

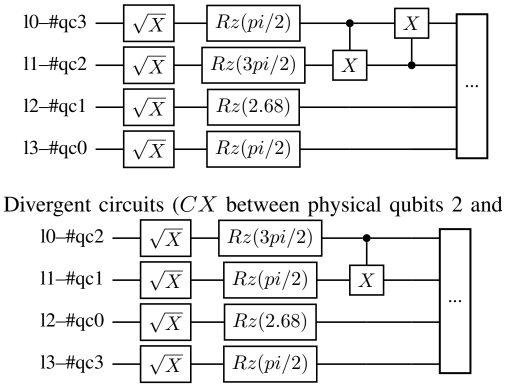
2) Divergences on real hardware: Figure 5 reports the total numbers of circuits executed on quantum hardware and those that exhibit behavioral divergence. The rate of latter is relatively low (0.3%-0.8%), which demonstrates the robustness of the quantum software and hardware we tested—this is not surprising since such software and hardware has been widely used and continuously improved for a number of years. On the other hand, these results also highlight the need of a systematic testing framework such as QDIFF for quantum developers—since quantum bugs are rare and hard-to-detect, developers should test their programs/tools exhaustively with a QDIFF-like approach before releasing them.



(**a**) "Divergent" circuits

(**b**) "Good" circuits

Fig. 7: Divergence on IBM `ibmq_yorktown`: no operation on qc4 for a long time.



(**a**) Divergent circuits (CX between physical qubits 2 and 3)

(**b**) "Good" circuits (CX between physical qubits 1 and 2)

Fig. 8: Divergence detected on IBM hardware: bad connection between qc_2 and qc_3.

For 29 divergence cases on IBM hardware, we manually inspected the corresponding circuits. We then determined 2 root causes that may explain 25 out of 29 cases. For the remaining 4 cases, we could not easily determine underlying root causes. We discuss each root cause with examples in this subsection.

Divergence due to 2 qubit gate errors We concluded that 1, 2, 6, and 7 divergences in P1, P3, P5, and P6 respectively—55% of all divergences detected on hardware in total—can be explained by placing many 2 qubit gates between so called *couplers* qubits. IBM released the reliability of connection between each qubit pair on their hardware [3]. For example, in hardware `santiago`, mapping CX between physical qubits $\{2, 3\}$ is less reliable than mapping CX between qubits $\{1, 2\}$ [3]. We found 16 out of 29 divergences could have the same underlying cause of using CX between known, unreliable qubit connections. These errors could be reduced through improved mapping to 2-qubit gates or using other strategies such as randomized compilation [47], [48].

Consider the two equivalent circuits shown in Figure 8, generated from P6 QAOA. These two circuits generated divergent measurements on `santiago`, although both circuits have nearly the same moments and the same number of 2-qubit gates. This is because the CX error rate of physical qubits 2 and 3 is much higher than qubits 1 and 2 (77% higher according to published information from IBM [3]). It appears that Qiskit's logical to physical qubit mapping procedure does not always avoid the use of CX on qubits 2 and 3 in their compilation and qubit allocation steps. Thus, Figure 8a produces divergence from Figure 8b.

Divergence due to qubit dephasing & decoherence: 9 of 29 divergences—2, 3, 2, and 2 divergences in P2, P4, P5, and P6—could have the same underlying cause of qubit dephasing & decoherence. Qubits that remain idle for long periods tend to dephase and decohere [49]. Figure 7 shows a pair of circuits with a similar depth and a similar number of 2-qubit gates. However, when run on hardware `ibm_belem`, the pair produces divergences beyond expected noise. We speculate that, when no operation is applied to physical qubit 4 for 18 moments, it may increase dephasing and decoherence possibilities. This can by fixed by adding two successive Pauli Y gates [30] on idle qubits during the compilation phase [49].

Others: For other 4 divergence, we could not easily determine underlying root causes. Because the circuit moments and the number of CX gate are roughly the same with their equivalent groups, stochastic errors in hardware can mostly be ruled out. The problem could be low-level quantum control software bugs that emerge from different combinations of gates, resulting in different control / coherent errors introduced at the pulse level.

D. Threats to Validity

Lack of Error Correction: The number of divergences on quantum hardware found by QDIFF would depend on the reliability of hardware and its error correction capability. If it were to run on an error-corrected quantum hardware, which does not exist yet, it may report fewer divergences and it would be easier to disambiguate whether divergences are caused by software-level defects as opposed to hardware-level defects.

Similarly, 2-qubit gate errors depend on which qubit connections that the gates are applied to, Therefore, it may be necessary to adjust the divergence threshold t based on the empirical 2 qubit error rates for each connection and how many times the 2 qubit gates are used on that connection. Such impact of 2 qubit error rates must be investigated further.

Time Out: In fuzz testing, longer experimentation periods tend to expose more errors or new program execution paths [50]. The total time taken for all our experiments was limited to seven days.

Number of Qubits: The maximum number of qubits that we used for our experiment was 5 qubits, because the only publicly available hardware limits public access up to 16 qubits and the waiting time tends to increase significantly, as you request more qubits. Running experiments on Google's sycamore processor with 53 qubits and 1000+ 2q gates may produce different results [51].

VI. RELATED WORK

Compiler and Framework Testing. Random differential testing (RDT) [52], [53] is a widely-used technique that compiles the same input program with two or more compilers that implement the same specification. Equivalence modulo inputs (EMI) [10] is such an example that tests compilers by generating equivalent variants. Many random program generators are used for compiler testing [54]. Csmith [55] randomly generates C programs and checks for inconsistent behaviors via differential testing. Quest [56] focuses on argument passing and value returning, while testing with randomly generated programs. Different from Csimth-like tools, refactoring-based testing systematically modifies input programs with refactorings, as opposed to random program generation. Orion [10] adapts EMI to test GCC and LLVM compilers. Christopher et al. [33] combine random differential testing and EMI-based testing to test OpenCL compilers. Orison [57] uses a guided mutation strategy for the same purpose.

Such classical compiler testing is not directly applicable to quantum software stacks due to the three challenges: (1) how to generate variants, (2) how to test simulators and hardware together with compilers, and (3) how to interpret quantum measurements for differential testing.

Quantum Testing and Verification. Zhao [58] introduces a quantum software life cycle and lists the challenges and opportunities we face. Ying et al. [59] formally reason about quantum circuits by representing qubits and gates using matrix-valued Boolean expressions, and verify them using a combination of classical logical reasoning and complex matrix operations. Huang et al. [60] introduce quantum program assertions, allowing programmers to decide if a quantum state matches its expected value. They define a logic to provides ϵ-robustness to characterize the possible distance between an ideal program and an erroneous one. Proq [61] is a runtime assertion framework for testing and debugging quantum programs. It transforms hardware constraints to executable versions for measurement-restricted quantum computers. QPMC [62] applies classical model checking on quantum programs based on Quantum Markov Chain. Ali et al. [35] propose a new testing metric called quantum input output coverage, a test generation strategy, and two new test oracles for testing quantum programs. Two test oracles include *wrong output oracle* that checks whether a wrong output has been returned, and *output probability oracle* that checks whether the quantum program returns an expected output with its corresponding expected probability. However, their work targets at quantum program testing and the measurement they used might not be sufficient enough. While all these techniques find errors in *quantum programs*, QDIFF aims to find errors in quantum software stacks.

Verified quantum compilers guarantee gate transformation and circuit optimization is correct by construction. CertiQ [12] is a verified Qiskit compiler by introducing a calculus of quantum circuit equivalence to check the correctness of compiler transformation. VOQC [11] provides a verified optimizer for quantum circuits by adapting CompCert [63] to the quantum setting. Smith and Thornton [64] present a compiler with built-in translation validation via QMDD equivalence checking. As discussed in Section I, these tools ensure correctness in quantum gate optimizations. However, QDiff is a complimentary technique based on testing and its scope includes both quantum backends and frontends.

Differential Testing. Differential testing [52] has been used to test large software systems and to find bugs in various domains such as SSL/TLS [65], [66], machine learning applications [67], JVM [68], and clones [69], etc. Mucerts [70] applies differential testing to check the correctness of certificate validation in SSL/TLS. It uses a stochastic sampling algorithm to drive its input generation while tracking the program coverage. DLFuzz [67] does fuzz testing of Deep Learning systems to expose incorrect behaviors. Chen et al. [68] perform differential testing of JVM with input generated from Markov Chain Monte Carlo sampling with domain-specific mutations with the knowledge of Java class file formats.

Mutation-Based Fuzz Testing. Fuzz testing mutates the seed inputs through a *fuzzer* to maximize a specific guidance metric, such as branch coverage. It has been shown to be highly effective in revealing a diverse set of bugs, including correctness bugs [71]–[73], security vulnerabilities [74]–[76], and performance bugs [77], [78]. For example, AFL [32] mutates a seed input to discover previously unseen coverage profiles. MemLock [77] employs both coverage and memory consumption metrics to find abnormal memory behavior.

Instead of flipping several bits or bytes in each mutation, several techniques support domain-specific mutations. SDF [79] uses seed properties to guide mutation in web-browser fuzz testing. BigFuzz [80] designs mutations for dataflow-based big data applications. QDiff is similar to this line of work by adapting mutation testing to a new domain. However, different from traditional fuzzing, QDiff re-invents the notion of a test oracle by re-designing a quantum measurement comparison method in a noisy, probabilistic domain.

VII. Conclusion

Quantum computing has emerged to be a promising computing paradigm with remarkable advantages over classical computing. QDiff is the first to reinvent differential testing for quantum software stacks. It adapts the notion of equivalence checking to the quantum domain, redesigns underlying program generation and mutation methods, and optimizes differential testing to reduce compute-intensive simulation or expensive hardware invocation. It is effective in generating variants, reduce 66% unnecessary quantum hardware or noisy simulator invocations, and uses divergence to isolate errors in both the higher and lower levels of the quantum software stack.

VIII. Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback. We thank Alan Ho for his kind help in selective invocation and distribution comparison, and for proofreading the draft. This work is supported in part by National Science Foundations via grants CNS-1703598, CNS-1763172, CNS-1907352, CNS-2007737, CNS-2006437, CNS-2106838, CNS-2128653, CHS-1956322 CCF-1764077, CCF-1723773, ONR grant N00014-18-1-2037, and Intel CAPA grant.

References

[1] Q. A. team and collaborators, “Cirq,” Oct. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4062499

[2] R. S. Smith, M. J. Curtis, and W. J. Zeng, “A practical quantum instruction set architecture,” *arXiv preprint arXiv:1608.03355*, 2016.

[3] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C. Chen *et al.*, “Qiskit: An open-source framework for quantum computing,” *Accessed on: Mar*, vol. 16, 2019.

[4] ——, “Qiskit: An open-source framework for quantum computing,” *Accessed on: Mar*, vol. 16, 2019.

[5] https://github.com/rigetti/pyquil/issues, 2020.

[6] https://github.com/quantumlib/Cirq/issues, 2020.

[7] https://stackoverflow.com/search?q=quantum++error, April,20 2021.

[8] M. Amy, D. Maslov, and M. Mosca, “Polynomial-time t-depth optimization of clifford+ t circuits via matroid partitioning,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1476–1489, 2014.

[9] Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, “Automated optimization of large quantum circuits with continuous parameters,” *npj Quantum Information*, vol. 4, no. 1, pp. 1–12, 2018.

[10] V. Le, M. Afshari, and Z. Su, “Compiler validation via equivalence modulo inputs,” *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[11] K. Hietala, R. Rand, S.-H. Hung, X. Wu, and M. Hicks, “A verified optimizer for quantum circuits,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–29, 2021.

[12] Y. Shi, X. Li, R. Tao, A. Javadi-Abhari, A. W. Cross, F. T. Chong, and R. Gu, “Contract-based verification of a realistic quantum compiler,” *arXiv preprint arXiv:1908.08963*, 2019.

[13] https://github.com/rigetti/pyquil/issues/1034, 2020.

[14] https://github.com/rigetti/pyquil/issues/1002, 2020.

[15] https://github.com/quantumlib/Cirq/issues/2553, 2020.

[16] https://www.cio.com/article/3161031/d-waves-quantum-computer-runs-a-staggering-2000-qubits.html, 2017.

[17] S.-O. Chan, I. Diakonikolas, P. Valiant, and G. Valiant, “Optimal algorithms for testing closeness of discrete distributions,” in *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 2014, pp. 1193–1203.

[18] A. Kolmogorov, “Sulla determinazione empirica di una lgge di distribuzione,” *Inst. Ital. Attuari, Giorn.*, vol. 4, pp. 83–91, 1933.

[19] N. V. Smirnov, “On the estimation of the discrepancy between empirical curves of distribution for two independent samples,” *Bull. Math. Univ. Moscou*, vol. 2, no. 2, pp. 3–14, 1939.

[20] I. Good, “Some terminology and notation in information theory,” *Proceedings of the IEE-Part C: Monographs*, vol. 103, no. 3, pp. 200–204, 1956.

[21] J. Shore and R. Johnson, “Axiomatic derivation of the principle of maximum entropy and the principle of minimum cross-entropy,” *IEEE Transactions on information theory*, vol. 26, no. 1, pp. 26–37, 1980.

[22] https://github.com/quantumlib/Cirq/issues/673, 2020.

[23] https://github.com/quantumlib/Cirq/issues/2240, 2019.

[24] https://github.com/quantumlib/Cirq/issues/1713, 2019.

[25] L. K. Grover, “A fast quantum mechanical algorithm for database search,” in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[26] P. W. Shor, “Algorithms for quantum computation: discrete logarithms and factoring,” in *Proceedings 35th annual symposium on foundations of computer science*. Ieee, 1994, pp. 124–134.

[27] R. LaRose, “Overview and comparison of gate level quantum software platforms,” *Quantum*, vol. 3, p. 130, 2019.

[28] (2019) Cirq: A python framework for creating, editing, and invoking noisy intermediate scale quantum (nisq) circuits. [Online]. Available: https:/github.com/quantumlib/Cirq

[29] R. Computing, “Pyquil documentation,” *URL http://pyquil. readthedocs. io/en/latest*, 2019.

[30] M. A. Nielsen and I. Chuang, “Quantum computation and quantum information,” 2002.

[31] https://quantumcomputing.stackexchange.com/questions/8694/is-there-a-mistake-in-the-vqe-ansatz-in-cirqs-tutorial, 2019.

[32] “American fuzz loop,” http://lcamtuf.coredump.cx/afl/, 2020.

[33] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, “Many-core compiler fuzzing,” *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 65–76, 2015.

[34] E. Mendiluze, S. Ali, P. Arcaini, and T. Yue, “Muskit: A mutation analysis tool for quantum software testing.”

[35] 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)*. IEEE, 2021, pp. 13–23.

[36] S. S. Tannu and M. K. Qureshi, “Not all qubits are created equal: a case for variability-aware policies for nisq-era quantum computers,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 987–999.

[37] S. Aaronson and L. Chen, “Complexity-theoretic foundations of quantum supremacy experiments,” *arXiv preprint arXiv:1612.05903*, 2016.

[38] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, and J. M. Gambetta, “Validating quantum computers using randomized model circuits,” *Physical Review A*, vol. 100, no. 3, p. 032328, 2019.

[39] Y. Shi, M. Wang, W. Shi, J.-H. Lee, H. Kang, and H. Jiang, “Accurate and efficient estimation of small p-values with the cross-entropy method: applications in genomic data analysis,” *Bioinformatics*, vol. 35, no. 14, pp. 2441–2448, 2019.

[40] E. Bernstein and U. Vazirani, “Quantum complexity theory,” *SIAM Journal on computing*, vol. 26, no. 5, pp. 1411–1473, 1997.

[41] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik, “The theory of variational hybrid quantum-classical algorithms,” *New Journal of Physics*, vol. 18, no. 2, p. 023023, 2016.

[42] D. Deutsch and R. Jozsa, “Rapid solution of problems by quantum computation,” *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, vol. 439, no. 1907, pp. 553–558, 1992.

[43] https://github.com/rigetti/pyquil/issues/1050, 2019.

[44] https://github.com/rigetti/pyquil/issues/1034, Nov,20 2020.

[45] https://github.com/quantumlib/Cirq/issues/3907, Nov,20 2020.

[46] https://github.com/rigetti/pyquil/issues/1259, Nov,20 2020.

[47] J. J. Wallman and J. Emerson, “Noise tailoring for scalable quantum computation via randomized compiling,” *Physical Review A*, vol. 94, no. 5, p. 052325, 2016.

[48] A. Hashim, R. K. Naik, A. Morvan, J.-L. Ville, B. Mitchell, J. M. Kreikebaum, M. Davis, E. Smith, C. Iancu, K. P. O’Brien *et al.*, “Randomized compiling for scalable quantum computing on a noisy superconducting quantum processor,” *arXiv preprint arXiv:2010.00215*, 2020.

[49] https://quantumai.google/cirq/google/best_practiceskeep_qubits_busy, 2021.

[50] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[51] X. Mi, P. Roushan, C. Quintana, S. Mandra, J. Marshall, C. Neill, F. Arute, K. Arya, J. Atalaya, R. Babbush, J. C. Bardin, R. Barends, A. Bengtsson, S. Boixo, A. Bourassa, M. Broughton, B. B. Buckley, D. A. Buell, B. Burkett, N. Bushnell, Z. Chen, B. Chiaro, R. Collins, W. Courtney, S. Demura, A. R. Derk, A. Dunsworth, D. Eppens, C. Erickson, E. Farhi, A. G. Fowler, B. Foxen, C. Gidney, M. Giustina, J. A. Gross, M. P. Harrigan, S. D. Harrington, J. Hilton, A. Ho, S. Hong, T. Huang, W. J. Huggins, L. B. Ioffe, S. V. Isakov, E. Jeffrey, Z. Jiang, C. Jones, D. Kafri, J. Kelly, S. Kim, A. Kitaev, P. V. Klimov, A. N. Korotkov, F. Kostritsa, D. Landhuis, P. Laptev, E. Lucero, O. Martin, J. R. McClean, T. McCourt, M. McEwen, A. Megrant, K. C. Miao, M. Mohseni, W. Mruczkiewicz, J. Mutus, O. Naaman, M. Neeley, M. Newman, M. Y. Niu, T. E. O’Brien, A. Opremcak, E. Ostby, B. Pato, A. Petukhov, N. Redd, N. C. Rubin, D. Sank, K. J. Satzinger, V. Shvarts, D. Strain, M. Szalay, M. D. Trevithick, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, I. Aleiner, K. Kechedzhi, V. Smelyanskiy, and Y. Chen, “Information scrambling in computationally complex quantum circuits,” 2021.

[52] W. M. McKeeman, “Differential testing for software,” *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[53] Y. Tang, Z. Ren, W. Kong, and H. Jiang, “Compiler testing: a systematic literature analysis,” *Frontiers of Computer Science*, vol. 14, no. 1, pp. 1–20, 2020.

[54] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, “An empirical comparison of compiler testing techniques,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180–190.

[55] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[56] C. Lindig, “Random testing of c calling conventions,” in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, 2005, pp. 3–12.

[57] V. Le, C. Sun, and Z. Su, “Finding deep compiler bugs via guided stochastic program mutation,” *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 386–399, 2015.

[58] J. Zhao, “Quantum software engineering: Landscapes and horizons,” *arXiv preprint arXiv:2007.07047*, 2020.

[59] M. Ying and Z. Ji, “Symbolic verification of quantum circuits,” *arXiv preprint arXiv:2010.03032*, 2020.

[60] Y. Huang and M. Martonosi, “Statistical assertions for validating patterns and finding bugs in quantum programs,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 541–553.

[61] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, “Proq: Projection-based runtime assertions for debugging on a quantum computer,” 2020.

[62] Y. Feng, E. M. Hahn, A. Turrini, and L. Zhang, “Qpmc: A model checker for quantum programs and protocols,” in *International Symposium on Formal Methods*. Springer, 2015, pp. 265–272.

[63] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, “Compcert-a formally verified optimizing compiler,” in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[64] K. N. Smith and M. A. Thornton, “A quantum computational compiler and design tool for technology-specific targets,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 579–588.

[65] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, “Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations,” in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 114–129.

[66] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, “Nezha: Efficient domain-independent differential testing,” in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 615–632.

[67] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, “Dlfuzz: Differential fuzzing testing of deep learning systems,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 739–743.

[68] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, “Coverage-directed differential testing of jvm implementations,” in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[69] T. Zhang and M. Kim, “Automated transplantation and differential testing for clones,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 665–676.

[70] C. Chen, C. Tian, Z. Duan, and L. Zhao, “Rfc-directed differential testing of certificate validation in ssl/tls implementations,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 859–870.

[71] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, “Fudge: fuzz driver generation at scale,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 975–985.

[72] J. Park, M. Kim, B. Ray, and D.-H. Bae, "An empirical study of supplementary bug fixes," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 40–49.

[73] Q. Zhang, J. Wang, and M. Kim, "Heterofuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, 2021, p. 242–254.

[74] J. De Ruiter and E. Poll, "Protocol state fuzzing of {TLS} implementations," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 193–206.

[75] T. Brennan, S. Saha, and T. Bultan, "Jvm fuzzing for jit-induced side-channel detection," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1011–1023.

[76] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[77] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in *42nd International Conference on Software Engineering*. ACM, 2020.

[78] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 254–265.

[79] Y.-D. Lin, F.-Z. Liao, S.-K. Huang, and Y.-C. Lai, "Browser fuzzing by scheduled mutation and generation of document object models," in *2015 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 2015, pp. 1–6.

[80] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, "Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 722–733.