Analyzing Quantum Programs with LintQ: A Static Analysis Framework for Qiskit

MATTEO PALTENGHI, University of Stuttgart, Germany

MICHAEL PRADEL, University of Stuttgart, Germany

As quantum computing is rising in popularity, the amount of quantum programs and the number of developers writing them are increasing rapidly. Unfortunately, writing correct quantum programs is challenging due to various subtle rules developers need to be aware of. Empirical studies show that 40–82% of all bugs in quantum software are specific to the quantum domain. Yet, existing static bug detection frameworks are mostly unaware of quantum-specific concepts, such as circuits, gates, and qubits, and hence miss many bugs. This paper presents LintQ, a comprehensive static analysis framework for detecting bugs in quantum programs. Our approach is enabled by a set of abstractions designed to reason about common concepts in quantum computing without referring to the details of the underlying quantum computing platform. Built on top of these abstractions, LintQ offers an extensible set of ten analyses that detect likely bugs, such as operating on corrupted quantum states, redundant measurements, and incorrect compositions of sub-circuits. We apply the approach to a newly collected dataset of 7,568 real-world Qiskit-based quantum programs, showing that LintQ effectively identifies various programming problems, with a precision of 91.0% in its default configuration with the six best performing analyses. Comparing to a general-purpose linter and two existing quantum-aware techniques shows that almost all problems (92.1%) found by LintQ during our evaluation are missed by prior work. LintQ hence takes an important step toward reliable software in the growing field of quantum computing.

CCS Concepts: • **Theory of computation → Program analysis**; • **Computer systems organization → Quantum computing**.

Additional Key Words and Phrases: program analysis, quantum computing, static analysis, bug detection

1 INTRODUCTION

Given the rising interest in quantum computing, ensuring the correctness of quantum software is increasingly important. Studies on bugs in quantum computing platforms [39] and quantum programs [36] show that many bugs in such software are problems specific to the quantum computing domain. For example, Paltenghi and Pradel [39] and Luo et al. [36] report that 40% and 82%, respectively, of the bugs found in quantum software are due to quantum-specific bug patterns. Detecting bugs in quantum programs is especially important because many bugs silently lead to unexpected results, which may be hard to spot due to the probabilistic results of quantum computations.

Authors' Contact Information: Matteo Paltenghi, University of Stuttgart, Stuttgart, Germany, mattepalte@live.it; Michael Pradel, University of Stuttgart, Stuttgart, Germany, michael@binaervarianz.de.

```
1  # Create a quantum registers and a classical register
2  qreg = QuantumRegister(4)
3  creg = ClassicalRegister(3)
4  # Create a quantum circuit
5  circ = QuantumCircuit(qreg, creg) # Bug 1: Oversized circuit
6  # Add gates and measurements to the circuit
7  for i in range(3):
8      circ.h(i)
9  circ.measure(qreg[0], creg[0])
10 circ.ry(0.9, qreg[0]) # Bug 2: Operation after measurement
11 circ.measure([0, 1, 2], creg)
12 # Execute the circuit on a simulator
13 backend_sim = Aer.get_backend("qasm_simulator")
14 results = backend_sim.run(transpile(circ, backend_sim), shots=1024).result()
```
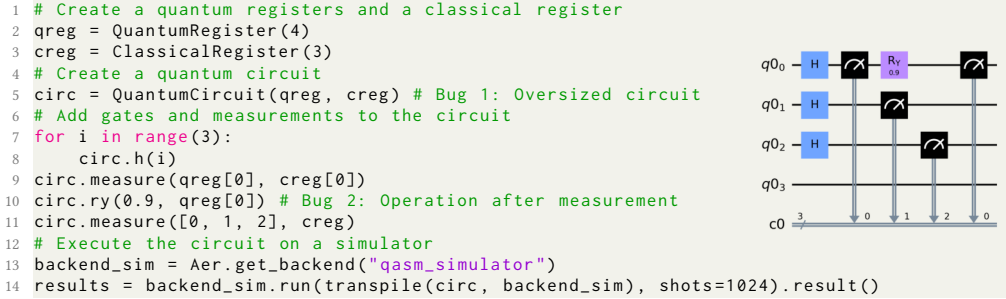
Fig. 1. Example of a quantum program with two bugs.

Unfortunately, popular bug detection tools, such as CodeQL [8], Pylint [4], Flake8 [1], Infer [13], and ErrorProne [24], are unaware of quantum computing. These tools consist of two parts: First, a framework that provides a set of abstractions to reason about general properties of programs, such as data flow and control flow. Second, a set of analyses built on top of the framework, each of which detects a particular kind of bug. This design has proven effective for general-purpose bug detection, as it allows to reuse the framework for different analyses. Yet, the abstractions provided by these frameworks are not sufficient to reason about quantum-specific concepts, such as quantum gates, quantum circuits, and quantum registers.

As a motivating example, Figure 1 shows a buggy quantum program. The program is based on Qiskit, a popular quantum computing platform, where a quantum program is Python code that uses a specific library. The code creates a circuit with a quantum register and a classical register, and then applies a sequence of gates and measurements to the circuit. The image on the right of the figure shows a graphical representation of the circuit. Despite being a simple program, the code contains two bugs. First, the code creates a circuit with four qubits, but then uses only three of them. Oversizing a circuit is strongly discouraged in quantum computing, because it wastes resources and because current hardware offers only a limited amount of qubits. Second, the code measures the state of qubit 0 at line 9, and afterwards applies a gate to the same qubit at line 10. Unfortunately, due to the properties of quantum mechanics, the measurement destroys the quantum state, and hence, the program feeds a collapsed state to the ry operation. Finding such bugs in an automated bug detector requires the ability to reason about quantum-specific concepts.

Applying a general-purpose static bug detector, e.g., Pylint [4], to the code in Figure 1 does not reveal the quantum-specific bugs. Recently, first techniques aimed at quantum programs have been proposed. One of them, QSmell [14] relies on dynamic analysis for most of its checks, and hence, is inherently harder to apply to real-world programs than a static analysis. Another one, QChecker [54], operates directly on the AST representation of quantum programs, but does not provide a general framework that abstracts over the details of the underlying quantum computing platform. Besides their conceptual limitations, neither QSmell[1] nor QChecker detects the bugs in Figure 1, showing that there is a need for a comprehensive static bug detection framework for quantum programs.

This paper introduces LintQ, a static analysis framework for detecting bugs in quantum programs. The approach is enabled by two key contributions. First, LintQ offers a set of abstractions of common concepts in quantum computing, such as circuits, gates, and qubits. These abstractions lift code written based on a specific quantum computing API, such as Qiskit, onto a higher level of abstraction. Second, we implement on top these abstractions an extensible set of ten analyses, each of which

---

[1]Here and also in our empirical evaluation, we refer to the static subset of QSmell's checks because getting arbitrary quantum programs to execute is non-trivial, e.g., due to unresolved dependencies and user input expected by a program.

queries the code for a particular kind of quantum programming problem. To benefit from prior work on general-purpose program analysis, the approach builds on an existing analysis framework, CodeQL [8].

Applying LintQ to the code in Figure 1 leads to warnings about the two bugs. To reach this conclusion, the framework first represents the different elements of the program using our quantum-specific abstractions. For example, this representation expresses the fact that the circuit created at line 5 has four qubits. An analysis checking for oversized circuits then uses this information to determine that the circuit only uses the first three of the four qubits. To find the second bug, an analysis aimed at warning about operations to a qubit applied after measuring the qubit builds upon the fact that our framework indicates that lines 9 and 10 operate on the same qubit. Importantly, none of the analyses need to reason about specific API calls in the Python code, but instead reasons about the quantum program at the level of LintQ's abstractions, which greatly simplifies the implementation of analyses.

We evaluate our approach by applying it to a novel dataset of 7,568 real-world, Qiskit-based quantum programs. The analyses built on top of our framework identify various problems in these programs. Manually inspecting a sample of 345 warnings from ten analyses shows that LintQ identifies 216 legitimate programming problems. Moreover, when using the default and recommended configuration of LintQ with six analyses, it achieves a precision of 91.0% (121 true positives out of 133 warnings). We reported 70 problems, seven of which have already been confirmed or even fixed. Our evaluation also shows that implementing an analysis on top of our abstractions is relatively simple, with an average of only 10 LoC per analysis, and that the analysis time is reasonable, with an average of 1.3 seconds per program.

In summary, this work makes the following contributions:

- A comprehensive framework for quantum program analysis, which provides reusable abstractions to reason about quantum software.
- Ten analyses implemented on top of these abstractions, which focus on programming problems reported in existing work on quantum-specific bugs [30, 55], or mentioned in GitHub issues and on StackExchange [3, 19].
- A novel dataset of 7,568 real-world, Qiskit-based quantum programs, which is the largest such dataset and hence, may serve as a basis for future work.
- A thorough evaluation of the effectiveness of the analyses, showing that the approach in its default configuration finds real-world issues with a precision of 91.0% while taking only 1.3 seconds to analyze a program.

2 BACKGROUND

2.1 Quantum Programming

Several quantum programming languages have been proposed, such as Qiskit [18], Cirq [17], Q# [44], and Silq [12]. Quantum programs, also called quantum circuits, are expressed as a sequence of operations, called quantum gates, applied to individual qubits. In Figure 1, we show a Qiskit program with four qubits and one classical bit, represented as horizontal lines, whereas the gates are shown as boxes or colored vertical lines. A special type of gate, called measurement gate, is used to measure the state of a qubit and store the result in a classical bit. The measurement gate, represented in black in the figure, produces a certain bit, either 0 or 1, with probabilities determined by the qubit state. Once the circuit has been defined, it is sent to a backend that executes it, typically a simulator or a real quantum computer. Note that a measurement has the important side-effect of destroying the quantum state. To sufficiently characterize its output, the circuit is executed multiple times, called *shots*, each time measuring the qubit.

```
1  import python
2  from If ifstmt, Stmt pass
3  where pass = ifstmt.getStmt(0) and
4    pass instanceof Pass
5  select ifstmt, "This 'if' statement is
       redundant."
```

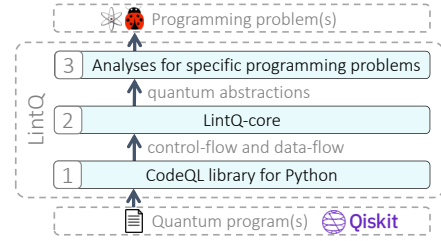Fig. 2. Example CodeQL query to find code redundant if statements.

Fig. 3. Overview of LintQ.

To define and run quantum programs, developers rely on a quantum computing platform. A popular approach is to implement a platform on top of Python, as done by Qiskit [18], Cirq [17], Tket [43], and Pennylane [10], so that a quantum program is essentially Python code that uses a specific library. Most platforms, with some noteworthy exceptions [12, 41, 48] describe programs on the level of circuits, and LintQ focuses on the analysis of quantum circuits. In particular, LintQ focus on code written using the Qiskit quantum computing platform due to its popularity in both practice [15] and research [21, 22, 36, 39, 40, 47, 56].

2.2   Static Analysis with CodeQL

CodeQL [8][2] is a popular engine for static analysis. It extracts facts from a program, such as its syntactic structure, data flow, and control flow, and then stores them in a relational database, which can be queried with the QL logic language. A query refers to classes, which we call *abstractions*, and their *predicates*, which represent relations between the abstractions. As a simple example, Figure 2 shows a CodeQL query that finds redundant if statements in Python. In the from section of the query, we define which program elements we consider, namely all if statements (*If*) and all statements (*Stmt*). Then, the where section restricts the query and focuses on those if statements that contain the pass keyword in their body. Finally, the select section specifies the warning and its message. At line 3, we see how the *If* abstraction offers a *getStmt* predicate, which connects the if statement with its first statement in the "if true" block. Although Python is the language of choice of many quantum computing platforms [5, 17, 43], the built-in QL library for Python extracts no information regarding quantum computing concepts, such as the quantum registers, the qubit position in a register, the quantum gates, or the difference between a state manipulation and a measurement, further motivating the need for LintQ.

3   OVERVIEW OF LINTQ

Figure 3 shows an overview of LintQ, which takes a quantum program written in Qiskit as its input, and then outputs warnings about quantum-specific programming problems in this program. LintQ is a static analysis framework realized in three stages:

(1) The existing static analysis engine CodeQL (Section 2) extracts general information about Python code, such as control flow paths, data flow facts, and how to resolve imports.
(2) The core of our approach, called LintQ-core, represents the behavior of the quantum program using a set of reusable quantum programming abstractions, such as qubits, gates, and circuits. Section 4 describes these abstractions in detail. The key benefit provided by this stage is to lift the program representation from a large and diverse set of Python constructs and Qiskit APIs into a smaller set of reusable abstractions.
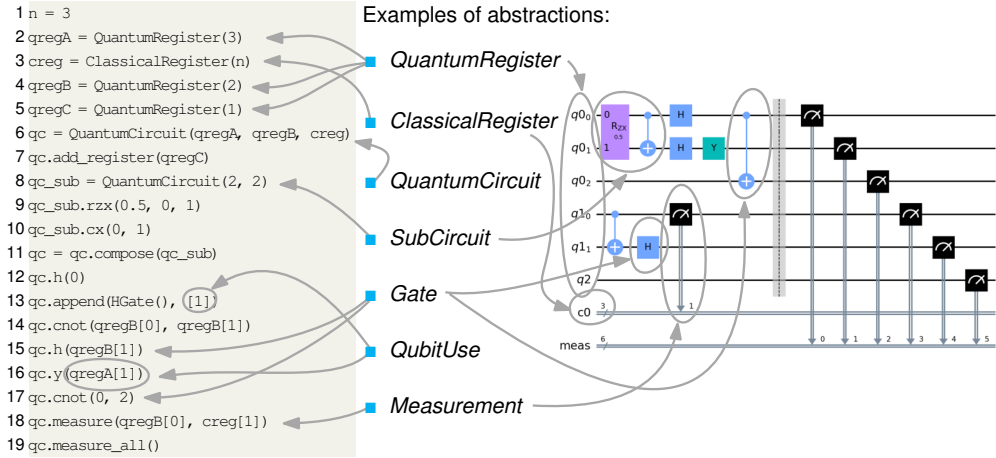
---

[2]https://codeql.github.com/

```
1  n = 3
2  qregA = QuantumRegister(3)
3  creg = ClassicalRegister(n)
4  qregB = QuantumRegister(2)
5  qregC = QuantumRegister(1)
6  qc = QuantumCircuit(qregA, qregB, creg)
7  qc.add_register(qregC)
8  qc_sub = QuantumCircuit(2, 2)
9  qc_sub.rzx(0.5, 0, 1)
10 qc_sub.cx(0, 1)
11 qc = qc.compose(qc_sub)
12 qc.h(0)
13 qc.append(HGate(), [1])
14 qc.cnot(qregB[0], qregB[1])
15 qc.h(qregB[1])
16 qc.y(qregA[1])
17 qc.cnot(0, 2)
18 qc.measure(qregB[0], creg[1])
19 qc.measure_all()
```

Examples of abstractions:

- QuantumRegister
- ClassicalRegister
- QuantumCircuit
- SubCircuit
- Gate
- QubitUse
- Measurement

Fig. 4. Examples of LintQ's abstractions and how they are represented in the program and the circuit.

(3) An extensible set of analyses builds on the abstractions to identify programming problems. Each analysis is formulated as a query over facts provided by CodeQL and LintQ-core, which allows for writing concise yet precise analyses. Section 5 describes ten analyses in detail.

Stages 2 and 3 of LintQ are the main technical contributions of our work. As Qiskit is a Python library, building on top of CodeQL in Stage 1 allows us to reuse its abilities at reasoning about Python programs. At the same time, CodeQL does not have any knowledge of quantum programming, which is why we introduce LintQ-core in Stage 2.

4   QUANTUM ABSTRACTIONS

LintQ-core provides a set of abstractions that represent concepts commonly found in quantum programs. The motivation for introducing these abstractions is that quantum programming platforms, such as the Qiskit, typically offer a wide range of APIs to express quantum computations. An alternative to LintQ-core would be to define analyses directly w.r.t. these APIs, which would require each analysis to consider the diversity of the Python language and the Qiskit APIs. As an example to illustrate this diversity, consider how a program may refer to qubits. First, there are multiple Python constructs for this purpose, including a single integer literal, e.g., 1, a single integer variable, e.g., qubit_idx, a sequence of integer literals or integer variables, e.g., [0, 1, 2], and expressions that retrieve a value from a variable that holds a qubit register, e.g., qreg[2]. Second, all the above references to qubits may occur in various code locations. For example, Qiskit offers over 50 functions to add different kinds of gates to a circuit. Each of these functions expects references to qubits at one or more argument positions. Instead of considering the full diversity of Python and Qiskit in each analysis, LintQ-core lifts quantum programs into more general abstractions. These abstractions enable us to write concise analyses that reason about quantum computing concepts instead of a low-level API that implements these concepts.

Figure 4 gives an overview of the abstractions offered by LintQ-core and illustrates through an example how they relate to the Qiskit API. Each abstraction corresponds to a CodeQL class that offers a number of predicates. These predicates express relationships between abstractions, e.g., to reason about the quantum register that stores the qubit a gate operates on, or properties of the specific abstraction, e.g. the size of a register. Although some of our abstraction borrow names used in the quantum circuit model [16, 38] and in Qiskit, the abstractions represent more general concepts than simple types or API calls. For example, as illustrated above, a single qubit or gate

can be referred to in various ways, but they all share the same abstractions. To clearly distinguish between the abstractions and the Qiskit API, we show abstractions in italics, e.g., *QuantumCircuit* and Python/Qiskit code in monospace, e.g., QuantumCircuit(qreg, creg).

4.1 Registers

The central storage facility of quantum programs are registers. Lines 2 to 7 of Figure 4 illustrate different ways of creating registers and associating them with circuits. For example, registers can be created explicitly by calling the respective constructors (lines 2 to 5), but also implicitly by passing the size of a register to a newly created circuit (line 8). Reasoning about the size of a register requires us to identify the integer constant that determines the register size, as illustrated in line 1. LintQ models registers and their relationships with the remaining program using two abstractions: *QuantumRegister* and *ClassicalRegister* for storing qubits and bits, respectively. The register abstraction is defined by a data flow node corresponding to the register's allocation site in the source code, e.g., a call to a constructor QuantumRegister. The abstraction offers predicates to retrieve the size of the register and the circuit it is associated with. For example, the *getSize* predicate of the *ClassicalRegister* created at line 3 returns 3 because LintQ tracks the value of the variable n (line 1) and its relationship with the register.

4.2 Quantum Circuits

A quantum circuit describes a sequence of instructions that operate on data stored in registers. Qiskit provides several APIs for creating circuits, composing larger circuits from smaller ones, and for associating circuits with other program elements, e.g., registers and gates. For example, lines 6 and 8 of Figure 4 create two circuits and then add one as a sub-circuit into the other.

LintQ models circuits using the *QuantumCircuit* abstraction. The predicates offered by this abstraction include *isSubcircuitOf*, which allows for checking whether a circuit is a sub-circuit of another one. The *getNumberOfQubits* and *getNumberOfClassicalBits* predicates track the size of each circuit, even when it uses multiple registers, such as in line 6 or when the registers are added later in the code, such as in line 7. Finally, *getAGate* and *getAQuantumRegister* express the relationship between a circuit and any of its gates or quantum registers.

The *QuantumCircuit* abstraction is lifted from different source-code constructs: (1) calls to the QuantumCircuit constructor, (2) any call to a user-defined function returning a quantum circuit, (3) any built-in constructor of parametrized circuits, such as EfficientSU2, (4) any unknown object created via an external function call that uses methods specific to quantum circuits, such as to_gate, to_instruction, assign_parameters, (5) any copy of an existent circuit created by calling copy on another circuit, (6) a call to transpile, which returns a new version if the circuit that is compatible with the instruction set and connectivity of the target quantum computer. All of these cases are modeled as subclasses of the *QuantumCircuit* abstraction, giving analyses the option to refer to specific kinds of circuits. For example, *TranspiledCircuit* abstraction can be used to enforce rules about circuits once they have been transpiled.

4.3 Subcircuits and Composition

To create larger circuits from smaller ones, quantum programming platforms offer APIs to compose circuits. LintQ models how different circuits are composed with each other via the *isSubcircuitOf* predicate. The predicate *isSubCircuitOf* tracks via dataflow analysis all those quantum circuits that flow into the append and compose methods of a quantum circuit object, and it identifies the object on which the method is called as the *parent* circuit and the circuit passed as the argument as its *subcircuit*. In addition, LintQ-core models two other cases where a circuit is not yet explicitly composed into another one, but is likely to be a subcircuit: (a) when the circuit is returned by

a function and (b) when an entire circuit is converted into an atomic instruction or gate via the `to_instruction` or `to_gate` methods.

4.4 Quantum Operators: Reversible and Irreversible

Instructions in quantum programs are expressed via quantum operators being added to a circuit. Intuitively, a quantum operator is any function that manipulates the quantum state by acting on the values stored in one or more qubits. There are two main types of quantum operators: reversible and irreversible. Because irreversible operators, such as measurements, destroy the quantum state, they are typically placed at the end of a quantum program. LintQ-core represents quantum operators using the *QuantumOperator* abstraction, which is a superclass of the *Gate*, *Measurement*, and *Reset* abstractions.

The *Gate* abstraction represents reversible quantum operators, such as the Hadamard gates used in lines 12 and 13 of Figure 4. Overall, there are several dozens of different APIs for creating gates and many more for connecting gates with other parts of a quantum program, e.g., the qubits a gate operates on. To enable analyses to reason about gates without repeatedly listing all gate-related APIs, LintQ-core offers the *Gate* abstraction, which captures all gates and their properties. The abstraction provides predicates to reason about a gate's relations to other program elements. For example, the *getQuantumCircuit* predicate relates a gate to the circuit it is added to, and the *getATargetQubit* predicate allows for reasoning about the qubit a gate operates on. For illustration, consider the control-not gate created at line 14 of Figure 4. The *getATargetQubit* predicate returns the fact that this gate operates on the qubits stored at indices 0 and 1 of the quantum register created at line 4.

To represent irreversible quantum operators, LintQ-core offers multiple abstractions: *Measurement* and *MeasurementAll* to represent measurements of a single qubit and all qubits in a register, respectively; *Reset* for operations that reset a qubit to the $|0\rangle$ state; and *Initialize* for operations that initialize one or more qubits with a vector of complex numbers. In Figure 4, LintQ creates measurement abstractions for the code at lines 18 and 19.

4.5 Uses of Qubits and Classical Bits

Quantum information stored in qubits typically is used and manipulated by multiple quantum operators. At the level of the Qiskit API, uses of qubits come in various forms. For example, as illustrated above, a program may refer to a qubit via an integer passed as an argument to a gate operation or via an index into an array that represents a register. Reasoning about qubit uses is compounded by the fact that different gate operations use different parameter indices to refer to the qubits they operate on. For example, when adding a controlled unitary gate to a circuit via `qc.cu(1, 2, 3, 4, 5, 6)`, then only the last two arguments refer to qubits, whereas the others are parameters of the gate.

To help analyses in precisely reasoning about qubit uses, LintQ offers the *BitUse* abstraction, split in its quantum and classical subclasses: *QubitUse* and *ClbitUse*. The *QubitUse* abstraction uniquely identifies a used qubit based on the register a qubit is stored in and based on the integer index of a qubit in this register. The *BitUse* abstraction offers predicates to connect to other abstractions, e.g., for obtaining the gate where the (qu)bit is used, the register where the (qu)bit is stored, and the corresponding circuit. The *getAnIndex* and *getAnAbsoluteIndex* predicates return the position of the qubit in the register and the position of the qubit in the circuit, respectively. The latter predicate keeps track of all the registers added to the circuit before the current accessed register and shifts the index accordingly. For our running example in Figure 4, the *QubitUse* abstractions represents each of the many references to qubits, such as the use qubit 1 of `qregA` at line 13 or the use of qubit 0 of register `qregB` at line 14.

4.6   Quantum Data Flow

An important property of quantum computations, which does not have a direct correspondence in classical computing, is the order in which quantum operators are applied to qubits. However, the order of adding two quantum operators to a circuit does not necessarily imply that the operator added first is executed after another operator added to the circuit later. Instead, the order of applying operations depends on the qubits the operators act on. LintQ-core derives the ordering of two operators if and only if they act in the same qubit. To this end, the approach uses the *QubitUse* abstraction described above to check if the qubits that two operators op_1 and op_2 manipulate are the same, and if so, derives an ordering relation based on the order in which op_1 and op_2 are added into the circuit. Following this reasoning for all quantum operators yields a partial order between quantum operators, that we call *quantum data flow*, since it describes how data stored in qubits flows between quantum operators. LintQ exposes this partial order to analyses via the *mayFollow* predicate that relates two quantum operators if and only if the two are part of the same circuit and the first may follow the second according to derived quantum data flow.

For example, consider the control-not gate (line 14) and the measurement (line 18) in Figure 4. Since both operate on the same qubit belonging to the qregB register, LintQ exposes their ordering in the *mayFollow* predicate. In contrast, the analysis does not claim any order between the gates created at lines 15 and 16, as they act on different registers.

In addition to the *mayFollow* predicate, LintQ-core offers: (i) *mayFollowDirectly*, a variant where the two quantum operators are applied on the same qubit directly after one another, without any other operation in between, (i) *sortedInOrder*, which checks whether three quantum operators may appear in the given order according to the quantum data flow. Note that both are defined only for unambiguous quantum operator additions, i.e., when the approach knows for sure that the operators act on the same qubit.

4.7   Access to Low-Level Constructs

Although our abstractions help in writing concise analysis, the analysis developer is not limited to those, but can refer to lower-level constructs when necessary. For example, an analysis could restrict a *Gate* to refer to specific kind of gate or reason on any other specific parameter of an API, if needed. Ultimately, the main benefit of LintQ is to avoid repeating the same low-level details across different analyses by capturing commonly required abstractions.

4.8   Soundness and Precision: Unknown Quantum Operators

Following the philosophy of existing linters and many other static analyzers [34], LintQ aims neither at full soundness nor at full precision. Instead, the approach offers a pragmatic compromise between the two, with the goal of finding as many programming problems as possible without overwhelming developers with spurious warnings.

To prevent analyses from raising incorrect warnings due to general limitations of static analysis, LintQ-core explicitly model unknown information, which gives an analysis the option to (now) draw conclusions based on such information. Specifically, LintQ-core exposes a *QubitUse* only if the approach can unambiguously resolve both the register and the index. For example, if a program applies a Hadamard gate with qc.h(idx) where idx is a variable obtained from user input, then LintQ leaves the qubit access unresolved so that analyses do not draw inaccurate conclusions. To the same end, LintQ-core exposes a *UnknownQuantumOperator* abstraction when the analysis framework cannot resolve some of the qubits used in that operator, e.g., qc.cx(0, i), where the value of i is not statically known. LintQ also considers functions that may extend a circuit as an *UnknownQuantumOperator*. We identify those as either: (i) a call to an unknown function,

Table 1. Analyses for finding quantum programming problems.

Analysis name	Description	Origin
Measurement-related and gate-related problems:		
DoubleMeas	Two measurements measure the same qubit state one after the other.	[55]
OpAfterMeas	A gate operates on a qubit after it has been measured.	[55]
MeasAllAbuse	Measurement results are stored in an implicitly created new register, even though another classical register already exists.	[55]
CondWoMeas	Conditional gate without measurement of the associated register.	[30]
ConstClasBit	A qubit is measured but has not been transformed.	[30]
Resource allocation problems:		
InsuffClasReg	Classical bits do not suffice to measure all qubits.	[55]
OversizedCircuit	The quantum register contains unused qubits.	[45]
Implicit API constraints:		
GhostCompose	Composing two circuits without using the resulting composed circuit.	[19]
OpAfterOpt	A gate is added after transpilation.	[3]
OldIdenGate	Using a now-removed API to create an identity gate.	[54]

where a *QuantumCircuit* flows in as an argument, (ii) a call to a function that directly modifies the *QuantumCircuit* by referring to it via a global variable.

As a preprocessing step to reduce the number of *UnknownQuantumOperators*, when reasoning about programs with loops, the framework unrolls loops that have a statically known number of iterations. Such loops are relatively frequent in quantum programs, e.g., when the programmer applies the same gates multiple times and specifies the loop bound with range(1, 3) or range(4). We limit unrolling to loops with at most ten iterations as a tradeoff between better modeling of the program and the risk of introducing too many new program elements, thus affecting the scalability of the approach.

5 ANALYSES FOR FINDING QUANTUM PROGRAMMING PROBLEMS

To illustrate the usefulness of LintQ-core, the following present an extensible set of ten analyses built on top of the abstractions provided by the framework.

5.1 Methodology: Collecting a Catalogue of Bug Patterns

To identify a set of quantum programming problems that can be detected by static analysis, we search through existing literature and developer discussions. We collect literature that studies programming issues in quantum programs by querying both the ACM Digital Library and IEEE Xplore, looking for any work that contains both keywords "quantum" and "bug" in its metadata. We apply a cutoff on the search results and inspect the top 50 results of each database, sorted by relevance, which yields 100 candidate papers. Next, we exclude papers that match one of the following: they are duplicate papers (2); they focus on hardware faults (13); they discuss only quantum computing concepts (12) or give an overview of the field (8); they are theory papers (2); they only use quantum computing methods and are not focused on bug detection (9); they focus quantum-related software, such as quantum computing platforms, but not on quantum programs (16); they do not provide a list of bugs or issues (16); or they require a specification of each quantum program, which typically is not available to a linter (16). We also exclude the dataset paper of Bugs4Q [53], because we use it as a benchmark in our evaluation. After this filtering, we are left with five papers that contain a list of programming issues or bug patterns in quantum programs. To complement the list of bug patterns from the literature, we collect three additional patterns that

```
1 circuit = QuantumCircuit(3, 3)
2 circuit.ccx(0, 1, 2)
3 circuit.measure(0, 0)
4 circuit.measure(2, 2)
5 # Problem: Qubit 0 already measured
6 circuit.measure(0, 1)
```

```
1 from Measurement m1, Measurement m2, int q
2 where mayFollowDirectly(m1, m2, q)
3 select m2, "Redundant measurement on the same
        qubit"
```

Fig. 5. Redundant measurement example (left) and its analysis (right).

we identified in developer discussions on StackOverflow and in GitHub issues using an approach similar to prior work [36].

Inspecting the selected papers and discussions, we find a total of 39 unique bug patterns for which we have clear examples of the problem to be identified. We exclude patterns for the following reasons: implementing an accurate analysis requires knowledge of the exact hardware the program will run on (5); they require a specification for each quantum program, e.g., describing the "correct" gate, which is generally not available to a linter (5); they require runtime information (1); they require abstractions not available in LintQ (7); the described pattern is not an issue anymore in the current Qiskit release (3); or they look for rare combinations of APIs that hardly appear in our large evaluation dataset (9). After this filtering, we obtain a list of ten analyses to implement in LintQ, which are listed in Table 1. Note that, although the ultimate goal of LintQ is to find bugs, we acknowledge that some of the patterns could also be considered code smells or anti-patterns. The following describes each analysis in detail by introducing the problem and by then describing how to query for instances of the problem using the LintQ abstractions.

5.2 Measurement-Related and Gate-Related Problems

Double measurement. Any two subsequent measurements on the same qubit produce the same classical result, making the second measurement not only redundant but also a possible sign of unintended behavior or a misunderstanding of the properties of quantum information. Figure 5 (left) shows an example of the problem. *Analysis*: The query to spot this problem is shown in Figure 5 (right). It searches for two consecutive measurements of the same qubit by checking whether the two operations are directly adjacent w.r.t. the order derived from quantum data flow (Section 4.6). Note that simply relying on the integer to spot two gates operating on the same qubit is ineffective since they might refer to the same position but in two different registers. To avoid this problem, the analysis relies on the *mayFollowDirectly* predicate provided by LintQ-core, which leads to simple and concise analysis.

Operation after measurement. When a qubit is measured, its quantum state collapses to a classical value, either 0 or 1. The measurement operation thus effectively destroys the quantum state. Any subsequent operation after the measurement acts on a destroyed quantum state, which is unlikely to be the intended behavior. Figure 6 (left) shows an example of the problem, where qubit 0 is measured and then a Pauli-Z gate is applied on it. Note how this case differs from a redundant measurement, where the clash is between two measurements of the same qubit, whereas here it is between a measurement and a gate. *Analysis*: The query searches for a quantum gate that is applied on a qubit, which has just been measured. Note that a trivial check of the usage of API calls that syntactically happen one after the other is insufficient, e.g., because the control flow may be more complex and the two calls could refer to two different qubits. Instead, our query ensures that the two operations are applied both to the same qubit belonging to the same register thanks to the mayFollowDirectly predicate, as shown in Figure 6 (right). The query excludes cases where applying the gate depends on a classical bit that resulted from a measurement, since this is a common pattern in quantum programs and the measurement preceding the gate could be required.

```
1 qc = QuantumCircuit(2, 2)
2 qc.h(1)
3 qc.cx(1, 0)
4 qc.measure(0, 0)
5 qc.measure(1, 1)
6 qc.z(0) # Problem: Qubit 0 has collapsed
7 qc.measure(0, 0)
```

```
1 from Measurement m, Gate g, int q
2 where
3   mayFollowDirectly(m, g, q)
4   and not g.isConditional()
5 select gate, "Gate after measurement
      on qubit " + q
```

Fig. 6. Operation after measurement example (left) and its analysis (right).

```
1 qc = QuantumCircuit(2, 2)
2 qc.h(q[0])
3 qc.cx(q[0], q[1])
4 # Problem: Implicitly creates a new
      classical register
5 qc.measure_all()
6 job = execute(qc,backend,shots=1000)
7 result = job.result().get_counts(qc)
8 # output: {'00 00': 487, '11 00': 513}
```

```
1 from
2   QuantumCircuit c, MeasurementAll m
3 where c = m.getQuantumCircuit() and
4   c.getNumberOfClassicalBits() > 0
5   and m.createsNewRegister()
6 select m, "measure_all() with classical
      register"
```

Fig. 7. Measure all abuse example (left) and its analysis (right).

Measure all abuse. In Qiskit, the API call measure_all with default arguments is used to measure all the qubits of a program and store the result in a classical register that is generated on the fly. Calling measure_all on a circuit that already contains a classical register may cause a silent problem since the output string would include additional output registers, while the original classical register would likely end up being empty, initialized with all zeros. Figure 7 (left) shows an example, where the output bitstring has four bits instead of two, because of the newly added register. Note that although the developer might still be able to correctly interpret the longer string, it is a waste of register space and might lead to unexpected results. *Analysis:* The query searches for a use of measure_all on a QuantumCircuit that has a classical register. This chain of relationships is handled by LintQ abstractions, as shown in Figure 7 (right). Note that, thanks to our abstractions, LintQ knows that the circuit has classical bits even if a ClassicalRegister object is not explicitly instantiated, but it gets this information indirectly when creating the circuit, i.e., QuantumCircuit(2, 2). Ultimately, we use an auxiliary predicate to ensure that the measure_all gate indeed creates a new register, which happens when it gets called with default arguments, i.e. whenever the argument is not add_bits=False.

Conditional gate without measurement. In quantum programming, conditional gates play a crucial role in introducing conditional behavior into quantum circuits. A conditional gate is applied to a target qubit only when a condition expressed through a classical bit is satisfied. For example, qc.h(0).c_if(creg, 0) applies the h gate only if the classical register creg contains the value 0. However, applying a conditional gate without any preceding measurement that stores a value into the classical bit(s) used in the condition essentially means a constant condition, which usually is not the programmer's intention. *Analysis:* The query searches for a conditional gate in a circuit in which all measurements are applied after the conditional gate, i.e., no preceding measurement exists. For better precision, the query excludes warning raised on circuits involved in circuit compositions, because those circuits might be composed with others that have a preceding measurement.

Constant classical bit. Whenever a qubit is manipulated, it is impossible to know its value without measuring it. In contrast, if a qubit is never modified, its state remains in the initial default state, i.e., 0, and thus any measurements of it will certainly return a constant value. *Analysis:* The query searches for a measurement on a qubit for which there is no preceding gate applied on it. To reduce false positives, it also excludes cases where the circuit has *Subcircuits* or an *UnknownQuantumOperator*.

```
1  from QuantumCircuitConstructor circ, int numQubits
2  where
3    // the circuit has a number of qubits
4    numQubits = circ.getNumberOfQubits() and numQubits > 0 and
5    // there is one qubit position not accessed by any gate
6    not exists(QubitUse bu, int i | i in [0 .. numQubits - 1] |
7      bu.getAnAbsoluteIndex() = i and
8      bu.getAGate().getQuantumCircuit() = circ
9    ) and
10   // the circuit has no (unknown) sub-circuits
11   not exists(SubCircuit sub | sub.getAParentCircuit() = circ) and
12   // there is no initialize op, because it can potentially touch all qubits
13   not exists(Initialize init | init.getQuantumCircuit() = circ) and
14   // all its registers have well-known size
15   not exists(QuantumRegisterV2 reg | reg = circ.getAQuantumRegister() and not reg.
        hasKnownSize()) and
16   // there are no unknown quantum operators
17   not exists(UnknownQuantumOperator unkOp | unkOp.getQuantumCircuit() = circ)
18 select circ, "Circuit has unused qubits"
```

Fig. 8. Analysis to detect oversized circuits (see Figure 1 for an example).

5.3 Resource Allocation Problems

The current generation of quantum computers are still limited in terms of qubits and gates, thus the use of resources must be carefully managed to avoid wasting them. At the same time, enough resources must be allocated for the quantum state to evolve and being measured correctly.

Insufficient classical register. This problem happens when we define a quantum program that uses more qubits than those that can be measured in the classical register allocated in the beginning. For example, the problem arises when the developer allocates a classical register with only two bits and then works on three qubits, i.e., QuantumCircuit(3, 2). *Analysis*: The query searches for circuits with the number of qubits greater than the number of classical bits. The *QuantumCircuit* abstraction is used to reason about the number of classical and quantum bits, but also to check with the predicate *isSubCircuit* that the circuit is not used as a sub-circuit. This is necessary to reduce false positives, since sub-circuits are often used legitimately without a classical register.

Oversized circuit. This problem happens when a program allocates a quantum register that is larger than the number of qubits actually used. Given the high cost of implementing a single qubit in hardware, when this issue happens it implies a waste of resources. The motivating example in Figure 1 shows an instance of this problem, where the program allocates a quantum register of size four but uses only three of the qubits. To the best of our knowledge, this work is the first to describe this programming problem. *Analysis:* As shown in Figure 8 (bottom), the query scans all gates used in a quantum circuit and raises a warning if any of the slots in the quantum register is not used (line 6). To ensure precision, the query checks several situations where no warning should be reported, e.g., circuits with an unknown register size or an unknown gate, and circuits that have sub-circuits.

5.4 Implicit API Constraints

The Qiskit API imposes several implicit constraints that quantum program should respect.

Ghost composition. When two circuits are composed with a compose call, the final merged circuit is the return value of the method call. That is, failing to use this return value leads to what we call a "ghost" composition, where the sub-circuit is not added at all. The problem is difficult to detect since it does not lead to any crash, but likely leads to unintended output. *Analysis*: The query searches for a call of compose on a quantum circuit, where the return value is not assigned to a variable or used otherwise. The query does not raise a warning in case the compose call has a inplace=True parameter, which would make the composition correct.

Operation after transpilation. Transpilation is a procedure to adapt the quantum program to the specific architecture of the quantum computer, and it involves optimization procedures. One particularly insidious optimization pass, which is active with optimization level 3, is "Optimize Swap Before Measure", because it removes any swap gate at the end of a circuit if the qubits it acts on are not measured. Thus, if the transpilation is done before adding the final measurements, this leads to a silent bug with unexpected results, because the swap gates get removed. *Analysis:* The query searches for a gate added to a circuit that was transpiled with optimization level 3. Thanks to the *TranspiledCircuit* abstraction, the reasoning on all transpiled circuits can be done with a single abstraction, which also offers a predicate to check the optimization level.

Old identity gate. The identity gate is a special gate that does not change the quantum state. Since Qiskit v0.23.0, the identity gate API call `qc.iden(index)` has been superseded by two new APIs: `qc.i(index)` and `qc.id(index)`. The use of deprecated APIs is a potential source of future problems, and it is thus important to detect it. *Analysis:* The query searches for the deprecated `iden` call in a quantum circuit. This analysis uses the existing *QuantumCircuit* abstraction, and could be easily extended to cover other deprecated API calls.

6 EVALUATION

We seek to answer the following research questions:

- **RQ1:** How effective are the LintQ-core abstractions at supporting the analyses?
- **RQ2:** Which bugs are detected by LintQ in real-world quantum programs?
- **RQ3:** What precision and recall does LintQ offer, and what causes false positives?
- **RQ4:** How does LintQ compare to existing static analyses?
- **RQ5:** How efficient is LintQ?

6.1 Dataset

The evaluation applies LintQ to a newly collected dataset of 7,568 real-world quantum programs gathered from GitHub. To gather Qiskit programs, which are the targets of LintQ, we query the GitHub search API for .py and .ipynb files that contain `import qiskit` or `from qiskit import` on Feb 14, 2023. Because some repositories contain generated and duplicate programs, we manually inspect the 30 repositories with most files returned, and remove repositories with generated or duplicate programs, coming mainly from companion repositories of research projects [39, 40, 47, 56]. This filter reduces the dataset from 75,304 to 41,739 programs. Next, we convert the notebooks to Python scripts using nbconvert [2]. Finally, we discard (i) files that cannot be parsed, (ii) duplicates based on the hash of the file content, (iii) files not including the search keywords in their code but, e.g., in a comment, (iv) code that is part of Qiskit itself. To the best of our knowledge, the resulting 7,568 programs are the largest dataset of real-world quantum programs to date. The resulting dataset includes 757,369 lines of code, excluding blank lines and comments. Figure 9 shows the frequency of the top-10 most popular gates in the dataset. The most popular gate are the Pauli-X gate (x), followed by the Hadamard (h), and the controlled-X gate (cx). On average, a program has 9.9 gate calls and 1.2 calls to the measure API.

6.2 RQ1: Effectiveness of Abstractions

The primary goal of LintQ's abstraction is to allow for writing concise analyses. To quantify to what extent we reach this goal, we compute the number of lines of code (LoC) needed to implement the various analyses. With an average of only 10 LoC, varying from 3 to 17 LoC, we confirm that our analyses are short. Note that we limit each line to 80 characters and that we also include the
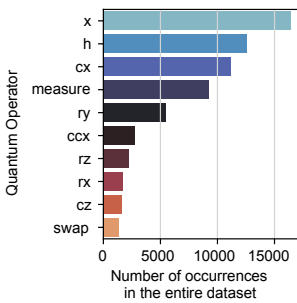
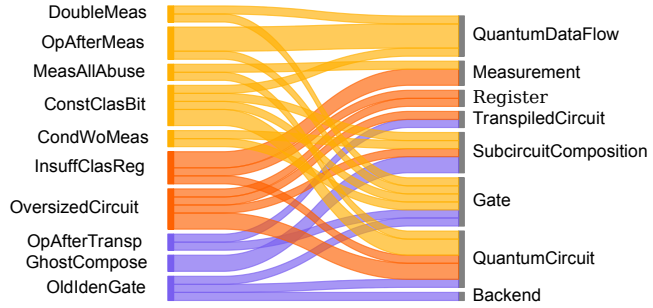Fig. 9. Frequency of most popular gates in our dataset.

Fig. 10. Mapping between analyses (left) and the abstraction families they use (right).

select clause, which provides an informative error message and therefore is longer than shown in the example analyses in Section 5.

To quantify the reusability of our abstractions, Figure 10 illustrates what analyses (left) explicitly use what abstraction families (right). The width of the connection between an analysis and a family of abstractions is proportional to the number of abstractions of that family that are used by the analysis. For example, *OpAfterMeas* uses the *mayFollow* and *sortedInOrder* predicates, which belong to the *QuantumDataFlow* family of abstractions. Each abstraction is used, on average, by 5.5 queries, showing that the abstractions are effectively reused across multiple analyses, with the *QuantumCircuit* being the most reused one. Each analysis uses, on average, 3.1 abstractions, showing how few abstractions are needed to implement an analysis. This is possible because the predicates often allow for indirectly obtaining information about other abstractions without explicitly declaring them in the query. For example, the *getNumberOfQubits* predicate of the *QuantumCircuit* abstraction allows for indirect reasoning about the size of registers without using the *QuantumRegister* abstraction.

> **Answer to RQ1**: LintQ abstractions are general enough to be reused across multiple analyses, which enables analyses with only 10 LoC, on average.

6.3 RQ2: Bugs Detected by LintQ

Table 3 presents the number of warnings reported by the different analyses. In addition to the absolute number of warnings, the third column also shows how many of all analyzed files have at least one warning produced by an analysis. The results show that the number of warnings varies widely from analysis to analysis. We evaluate the precision of analyses, i.e., their ability to pinpoint relevant problems, as opposed to false positives, in RQ3.

Reporting Protocol. While inspecting a subset of the warnings that LintQ produces, we found 216 bugs that should be fixed in the quantum programs. Because adequately reporting bugs to developers is a time-consuming process, we report a subset of 70 of the true positives. The subset is selected by prioritizing warnings based on their likelihood of eliciting a response from the developers and based on the relevance of the project. Concretely, we consider the number of stars, contributors, issues, and the days since the last commit to identify projects where reporting issues is worthwhile. We then review all the remaining true positives and after careful consideration, we identify some cases in which, although the programming issue is present, it is not worth the time of the developers to fix it. The cases which we deemed not worth reporting issues are: the repository is archived (9) or not maintained (31), e.g., with more than two years since the last commit; the

Table 2. Examples of true positives. The ID refers to the supplementary material.

ID	Analysis	Description	Manifestation	Status
1	DoubleMeas	Redundant measurement	Silent	Fixed
2	GhostCompose	Missing compose assignment in a test case	Silent	Fixed
3	GhostCompose	Missing compose assignment in a test case	Silent	Fixed
4	GhostCompose	Missing compose assignment in a test case	Silent	Fixed
5	MeasAllAbuse	Measurement creates overly long output	Silent	Fixed
6	MeasAllAbuse	Measurement creates overly long output	Silent	Confirmed
7	OpAfterMeas	Non idiomatic teleportation, lack of c_if	Silent	Confirmed
8	InsuffClasReg	Unused qubit	Silent	Reported
9	OpAfterTransp	Measure gate added after transpilation	Silent	Reported
10	OpAfterTransp	Measure gate added after transpilation (Grover's algor.)	Silent	Reported
11	MeasAllAbuse	Measurement creates overly long output (Bell pair)	LongOut	Reported
12	MeasAllAbuse	Measurement creates overly long output (GHZ algor.)	LongOut	Reported
13	MeasAllAbuse	Measurement creates overly long output	LongOut	Reported
14	GhostCompose	Missing insertion of the inverse QFT (Shor's algor.)	Silent	Reported
15	MeasAllAbuse	Measurement creates overly long output (CTF competition)	Silent	Not Fix

problem happens in a test case that runs despite the quantum circuit being buggy, because it tests a different feature (27); the problem happens in a personal project, which was obviously not created with the intention of maintaining the code (16); the problem happens in the interactive context of a notebook, where incomplete or buggy code is generally allowed for visualization purposes (16), or in teaching material, which is purposefully buggy or incomplete (47).

Developer Responses. Table 2 lists a selection of the reported bugs, along with a short description, the analysis involved, and how the problem manifests. If the "Manifestation" column reports "silent", then the bug does not lead to any obvious sign of misbehavior, such as an exception, but instead either silently produces an incorrect result or causes unnecessary computation. Finally, the last column shows which bugs we have already reported to the developers, and if available, their response. In total, seven of the bugs have already been confirmed or even fixed, and all but one of the remaining reports are still pending.

Examples. We describe some examples of bugs found by LintQ. Figure 11a shows code where a measurement is performed on a qubit that has just been measured. The second measurement hence is redundant and causes unnecessary computation. The developers confirmed and fixed this bug in response to our report. As a second example, Figure 11b shows some testing code where the states of two circuits are compared. However, the two circuits are identical because the compose call is not properly assigned to the second circuit (line 10), and thus, no circuit contains the Toffoli gate (line 9), which should instead have been the subject of the test. This bug was confirmed by the developers and fixed in response to our report. Notably, the only "not fix" is because the repository belonged to a competition submission, and the developers preferred to keep the version that they submitted for historical reasons. Interestingly, LintQ also finds a measure_all bug in the official Qiskit tutorials (Id 1 in Table 2), which was confirmed by the developers, showing that even highly maintained quantum code may suffer from problems LintQ can find.

> **Answer to RQ2**: LintQ identifies problems in various real-world quantum programs, some of which were already confirmed and fixed by the respective developers.

```
1  def inefficientNOT(inefficiencies: int,      1  def test_linear_toffoli2(self):
      inp: str):                                2    gate_x = np.array([[0, 1], [1, 0]])
2    qc = QuantumCircuit(1, 1)                   3    qc2 = QuantumCircuit(4)
3    qc.reset(0)                                 4    qc2.x(2)
4    if inp == '1':                              5    qc2.x(3)
5      qc.x(0)                                   6    qc2.x(0)
6    qc.barrier()                                7    state1 = qclib.util.get_state(qc2)
7    for i in range(inefficiencies):             8    circ = QuantumCircuit(4)
8      print(i+1, "x gates added")              9    mc_gate(gate_x, circ, [3, 2, 1], 0)
9      qc.x(0)                                  10    qc2.compose(circ, qc2.qubits) # Bug
10   qc.barrier()                               11    state2 = qclib.util.get_state(qc2)
11   qc.measure(0,0)                            12    self.assertTrue(np.allclose(state1,
12   trial = qc.measure(0,0)   # Bug                  state2))
```

(a) Redundant measurement problem. (ID: 2)   (b) Ghost composition problem. (ID: 3)

Fig. 11. Examples of bugs found by LintQ.

Table 3. Warnings and precision of the LintQ analyses (left) and the result of manual inspection (right).

Analysis Name	Tot. warnings	% Files	Precision	FP / NW / TP
DoubleMeas	39	0.36%	72.0%	4/3/18
OpAfterMeas	127	0.92%	100.0%	0/0/44
MeasAllAbuse	22	0.26%	94.1%	0/1/16
ConstClasBit	533	4.29%	48.3%	21/10/29
CondWoMeas	46	0.22%	100.0%	0/0/28
InsuffClasReg	3489	17.35%	34.8%	22/21/23
OversizedCircuit	378	3.01%	50.0%	16/13/29
OpAfterTransp	7	0.05%	100.0%	0/0/7
GhostCompose	12	0.09%	66.7%	0/4/8
OldIdenGate	46	0.37%	50.0%	11/3/14

6.4 RQ3: Precision and Recall

Precision. Because precision is crucial for practical adoption of static analyzers [11, 20, 28], we assess to what extent the analyses suffer from false positives. We manually inspect a random sample of ten warnings for each analysis, or all produced warnings if that number is lower than ten. Two of the authors, who are both experienced in static analysis and with quantum computing knowledge, independently inspect the warnings and then discuss them to reach a consensus. After the initial inspection, a 70.1% agreement was reached, and after the discussion, all disagreements were resolved. Based on the agreement, we compile an annotation protocol and a single author proceeds to annotate more warnings up to reaching a statistically relevant sample with a confidence level of 90% and a margin of error of 10% for each of the ten analyses, similar to related work [23].

We categorize each warning into one of three categories. A *true positive* is a warning that reveals clearly incorrect behavior in the program. Such incorrect behavior may result in a program crash, in incorrect output, or in an unnecessary performance degradation. A *noteworthy* warning is a potential problem where the analysis correctly detects an instance of the targeted programming problem, but we cannot certainly say whether the behavior is unexpected by the developer. Finally, a *false positive* is a warning reported despite the code being correct, which is typically caused by overly strong assumptions made by an analysis. Based on this classification, we compute *precision* as the percentage of true positives among all warnings. That is, our notion of precision underestimates the true precision, as it includes noteworthy warnings in the denominator, but not in the numerator.

Table 3 illustrates the results of our manual inspection. Each analysis identifies at least a few true positives. The median precision across all analyses is 69.3%. The overall precision across all inspected warnings is 62.6% (216 true positives out of 345 inspected warnings). In practice, we recommend to enable those analyses that produce sufficiently precise results for the usage

environment, and to inspect warnings by high-precision analyses first. For example, keeping only analyses with precision above 50%, yields an overall precision of 91.0% from the remaining six analyses, which we recommend as the default configuration for LintQ.

Root causes of false positives. To better understand the reasons for false positives, we discuss representative cases in the following. False positives of the *InsuffClasReg* happen when: (ii) the circuit has more qubits than classical bits because of the presence of ancilla qubits, i.e., auxiliary storage used during a computation that does not need to be measured; (ii) the circuit is used as a submodule of a bigger circuit, thus it is not responsible of instantiating the classical bits. Better distinguishing between ancilla qubits and missed classical bits remains as a challenge for future work. The *OversizedCircuit* analysis causes false positives when the circuit has sub-circuits generated via a function call (e.g., `qc.append(QFT(3), qargs=[0, 1, 2])`), which LintQ currently does not track, thus making the circuit appear underused.

Recall. Since we do not know the ground truth of all bugs in the 7,568 real-world quantum programs, we cannot compute the recall of LintQ on this large dataset. Instead, we use Bugs4Q [36, 56][3], an existing benchmark of 42 quantum bugs. We run LintQ on the 42 buggy files, manually inspect the warnings by using the same annotation procedure applied in the rest of this work, and then check how many of LintQ's true positives match a known bug in Bugs4Q. LintQ raises four true positive warnings (two by the *OpAfterMeas*, one by *MeasAllAbuse*, and one by *OldIdenGate*) that correspond to three known bugs in Bugs4Q. Thus, the recall of LintQ is 7.1% (3/42). While this number may seem low, it is actually higher than the recall of popular static bug detectors on Defect4J [29], which prior work has measured to be between 1% and 3%, depending on the bug detector [25]. Interestingly, LintQ also finds some problems in the benchmark code beyond the known bugs of the benchmark. For example, the *InsuffClasReg* analysis raises several true positive warnings because some circuits do not have any classical register, which is likely due to the fact that the examples are incomplete code snippets gathered from issues and forum questions.

> **Answer to RQ3**: Our analyses have an overall precision of 62.6%, or when turning off low performing analyses, 91.0% from the remaining six analyses which we recommend as the default configuration for LintQ. The recall is 7.1% on the Bugs4Q benchmark. Common root causes of false positives are ancilla qubits and insufficient modeling of complex circuit compositions.

6.5 RQ4: Comparison with Prior Work

Due to the young field of quantum software engineering, there are only few static analyzers aimed at quantum programs: (i) QSmell [14], which detects smells in quantum programs, (ii) QChecker [54], an AST-based static analysis tool for quantum programs, (iii) QCPG [30], a toolkit that extends Code Property Graphs [51] to analyze quantum code. Moreover, since the quantum programs we analyze are written in Python, we also compare with Pylint [4], a popular linter for Python designed for classical software. For QSmell we focus on their two static analysis-based detectors; for QChecker we use their eight AST-based checkers; for Pylint, we run the tool its default configuration. Unfortunately, we had to drop QCPG from the comparison because the tool is not available yet.[4]

Prior work applied to problems found by LintQ. We run all three competitors on the programs where LintQ detects one of the 216 true positives that we have manually confirmed (Section 6.4), and we check which of them the existing techniques detect. QSmell, QChecker, and Pylint raise 77, 200, and 8,627 warnings, respectively. We inspect each warning that is at the same line as one of the LintQ warnings, which corresponds to 0, 9, and 42 warnings, and we assess

---

[3]https://github.com/Z-928/Bugs4Q-Framework
[4]Although the authors plan to release the tool, the repository states the code is undergoing export checks, as confirmed by email.

whether they refer to the same problem or coincidentally flag the same line. For the QChecker warnings, we found that seven problems are found by both QChecker and LintQ, two of QChecker's warnings coincidentally flag the same line, and the remaining 207 are missed by QChecker and found only by LintQ. For the Pylint warnings, 15 Pylint warnings correspond to the same issues flagged by *OldIdenGate* (14) and *OversizedCircuit* (1), 32 warnings coincidentally flag the same line, including conventions and coding style (29), and uses of missing APIs and arguments (3), and the remaining 174 warnings are missed by Pylint and found only by LintQ. Overall, prior work can find only 7.9% (17/216) of the true positives found by LintQ, overlooking the remaining 92.1% (199/216).

LintQ applied to problems found by prior work. We also study the opposite direction: How many of the warnings raised by the competitors are also raised by LintQ? To this end, we run each quantum-specific competitor and inspect a sample of up to ten warnings produced by each of their detectors. For QSmell and QChecker we inspect, respectively, 20 warnings from two detectors and 59 warnings from six detectors.

For QSmell, we unfortunately found no true positives among the 20 inspected warnings. The first detector (*NC*) flags a file that has more `run` and `execute` calls, used to run a circuit on a simulator or real hardware, than `bind_parameters` calls, used to convert any parametric gate into its concrete version before execution. These warnings are false positives, for two reasons: (1) The warning is emitted also when there is a single `execute` call, which is normal for any circuit that uses only concrete gates. (2) The detector does not model the `assign_parameters` API, which is a legitimate alternative to calling `bind_parameters`. The second detector (*LPQ*) checks if there is a `transpile` API call without the argument `initial_layout` set, since passing that argument is a good practice when running on a real quantum computer. Again, all the inspected warnings are false positives, for two main reasons: (1) The missing `initial_layout` argument is present with a simulator backend, which in practice has no hardware constraint to respect. (2) The rule considers any `transpile` calls, even those not belonging to Qiskit, which has no `initial_layout` argument.

For QChecker, we found three true positives among the 59 inspected warnings. The true positives are raised by the *Deprecated Order* detector, which flags a deprecated usage of the `iden`, analogously to our *OldIdenGate* analysis. However, QChecker also reports many false positives, primarily because it warns about any function call that includes the substring `iden`, which also happens in functions unrelated to the quantum library. All three true positives found by QChecker are also detected by LintQ, because our *OldIdenGate* analysis targets the same kind of problem. The main difference is that LintQ raises fewer false positives, because it explicitly models gates, instead of relying on a text-based matching of the API name.

> **Answer to RQ4**: Overall, 92.1% of the 216 true positves issues found by LintQ are overlooked by the existing techniques, whereas the three true positives found by the existing quantum-specific techniques QSmell and QChecker are also found by LintQ. Pylint remains effective on mostly classical patterns, such as *OldIdenGate*, but it misses all other quantum-specific problems found by LintQ.

6.6 RQ5: Efficiency

We measure the time spent for analyzing all 7,568 quantum programs with all ten analyses. All experiments are run on an Ubuntu machine with an Intel Xeon Silver 4214 CPU with 48 cores and 252 GB of RAM. There are three main computational steps: (i) Using CodeQL to build the database of facts about the Python code, which takes 74 minutes for all 7,568 programs; (ii) Compiling the query plan of the analyses, which takes 97.0 seconds; and (iii) Running the analyses on the database, which takes 162 minutes. Inspecting the computational cost of individual analyses shows that the two most expensive analyses are those that reason about the gate execution order, *DoubleMeasurement*

and *OpAfterMeasurement*, which take 2,290 and 1,761 seconds to evaluate, respectively. Taking all three steps together, LintQ takes 1.3 seconds per analyzed program.

Answer to RQ5: With 1.3 seconds per analyzed program, LintQ is sufficiently fast for a practical analysis.

7 THREATS TO VALIDITY

Internal Validity. First, our analyses scan each program file individually, not considering the other files in the same repository. Applying LintQ at the repository level may produce different warnings. Second, we inspect only a subset of all warnings. To mitigate this thread, we sample the inspected warnings randomly and use a statistically relevant sample size. Third, our implementation may contains bugs. To mitigate this risk, we implement test cases for the abstractions and the analyses, and we make our implementation publicly available as open-source. Fourth, our literature review may have missed some relevant bug patterns. However, LintQ is designed to be extensible, i.e., additional bug patterns can be added to the framework in the future. Fifth, the manual inspection of warnings is inherently subjective. To mitigate this, two authors participated in the process, collaboratively developing and agreeing on an annotation protocol. This protocol has been documented and is made available in our artifact.

External Validity. First, while the abstractions of LintQ are designed to be also applicable to other quantum computing platforms, such as Cirq [17] and Tket [43], we cannot claim that our results generalize beyond Qiskit. Second, we cannot guarantee that our results generalize to other quantum programs. To mitigate this threat, we evaluate the approach on 7,568 real-world programs, which represents the largest such dataset to date.

8 RELATED WORK

Quantum Software Testing. Miranskyy et al. [37] highlight quantum-specific debugging issues when working with quantum programs and discuss how classical solutions could be adapted to the quantum domain. Regarding platform code, various approaches have been proposed, including differential testing [47], metamorphic testing [40], and fuzzing [49]. However, they all focus on platform code and require executing the code, whereas LintQ focuses on application code and is based on static analysis. Regarding application code, various techniques have been proposed. QuanFuzz[46] tests a single algorithm with different inputs with the goal of maximizing branch coverage via a genetic algorithm. Quito [6] relies on a program specification and statistical tests to evaluate the correctness of a single small program. Huang and Martonosi [26] propose statistical approaches to evaluate assertions in a quantum program. Li et al. [33] describe a projection-based runtime assertion scheme that allows for asserting in the middle of the circuit without affecting the tested state if the assertion is satisfied. All these approaches assume to have single circuit programs that can be easily executed multiple times, which may not be the case in practice. Moreover, they rely on executing the programs, whereas LintQ is based on static analysis.

Quantum Program Analysis. Few analyses for quantum programs have been proposed so far, including QSmell [14], ScaffCC [27], QChecker [54], and QCPG [30]. QSmell mostly relies on dynamic analysis, and it focuses on code smells only. ScaffCC is a compiler that performs a limited set of analyses using a new flavor of QASM, whereas we focus on Qiskit-based Python code. QChecker is a static analysis tool that relies only on AST information, but does not provide a general framework to build new analyses and does not model any control flow. QCPG [30] extends Code Property Graphs [51] to analyze quantum code in a single circuit, whereas LintQ is designed to analyze entire programs and models the composition of circuits, as well as unknown quantum operators. There

also exist quantum-specific program analyses, such as entanglement analysis [27, 42] and automatic uncomputation [12, 41, 50]. However, most of them address a single problem each, whereas LintQ offers a set of general abstractions.Quantum abstract interpretation [52] and runtime assertions [33] are two techniques to assert properties of quantum computations. They require manually crafted, algorithm-specific assertions, whereas LintQ does not require any prior knowledge of the program. In summary, LintQ goes beyond the purely syntactic level and single-circuit approaches by providing reusable abstractions to build a wide range of analyses in realistic settings with multiple circuits and unknown quantum operators.

Datasets of Quantum Programs. Paltenghi and Pradel [39] share a large dataset of bugs in quantum computing platforms. However, the focus of LintQ is on application code written in Qiskit, and not on platform code. Two application-level datasets are QASMBench [32], which includes 48 programs written in OpenQASM, and work by Long and Zhao [35], which proposes a dataset of 63 Q# programs.However, they are not suitable for our evaluation because they are not written in Python/Qiskit. Luo et al. [36] and Zhao et al. [56] study the bugs in quantum computing programs in Qiskit, mainly collected from GitHub issues of the official Qiskit repository and StackOverflow questions. In contrast, we present a much larger dataset of 7,568 real-world programs, including many programs that are not part of the Qiskit repositories.

Domain-Specific API Modeling. Previous work has modeled other specialized Python libraries, e.g., in machine learning [9, 31], to spot bugs with static analysis. Our work also relates to general static API misuse detectors [7], which mostly focuses on Java and traditional application domains. Instead, we focus on the quantum domain, which comes with its own concepts and APIs to model.

9 CONCLUSION

We present LintQ, a framework for statically analyzing quantum programs and an extensible set of ten analyses. The approach introduces a set of abstractions that capture common concepts in quantum programs, such as circuits, gates, and qubits, as well as the relations between these concepts. Thanks to these abstractions, analyses aimed at finding specific kinds of programming problems can be easily implemented in a few lines of code (10 LoC). To evaluate LintQ, we apply the approach to a novel dataset of 7,568 quantum programs, and in its default configuration with six analyses, LintQ achieves a precision of 91.0% (121 true positives out of 133 warnings).

10 DATA AVAILABILITY

LintQ, our dataset, and all results are publicly available at https://github.com/sola-st/LintQ and archived at https://zenodo.org/records/11095456.

11 ACKNOWLEDGMENTS

REFERENCES

[1] [n. d.].   Flake8: Your Tool For Style Guide Enforcement — Flake8 6.0.0 Documentation. https://flake8.pycqa.org/en/latest/.
[2] [n. d.].   Nbconvert: Convert Notebooks to Other Formats — Nbconvert 7.2.9 Documentation. https://nbconvert.readthedocs.io/en/latest/index.html.
[3] [n. d.]. OptimizeSwapBeforeMeasure Pass Drops Swap Gate (Even If There Is NO Measure after It) · Issue #7642 · Qiskit/Qiskit. https://github.com/Qiskit/qiskit/issues/7642.
[4] [n. d.]. Pylint - Code Analysis for Python | Www.Pylint.Org. https://www.pylint.org/.
[5] 2021. Qiskit/Qiskit. https://github.com/Qiskit/qiskit.

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

[7] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* 45, 12 (Dec. 2019), 1170–1188. https://doi.org/10.1109/TSE.2018.2827384

[8] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[9] Wilson Baker, Michael O'Connor, Seyed Reza Shahamiri, and Valerio Terragni. 2022. Detect, Fix, and Verify TensorFlow API Misuses. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 925–929. https://doi.org/10.1109/SANER53432.2022.00110

[10] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, M. Sohaib Alam, Shahnawaz Ahmed, Juan Miguel Arrazola, Carsten Blank, Alain Delgado, Soran Jahangiri, Keri McKiernan, Johannes Jakob Meyer, Zeyue Niu, Antal Száva, and Nathan Killoran. 2020. PennyLane: Automatic Differentiation of Hybrid Quantum-Classical Computations. *arXiv:1811.04968 [physics, physics:quant-ph]* (Feb. 2020). arXiv:1811.04968 [physics, physics:quant-ph]

[11] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[12] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3385412.3386007

[13] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings 7*. Springer, 3–11.

[14] Qihong Chen, Rúben Câmara, José Campos, André Souto, and Iftekhar Ahmed. 2023. The Smelly Eight: An Empirical Study on the Prevalence of Code Smells in Quantum Computing - Artifact. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (Jan. 2023). https://doi.org/10.5281/ZENODO.7556360

[15] James Dargan. 2022. Top 5 Quantum Programming Languages.

[16] David Elieser Deutsch and Roger Penrose. 1997. Quantum Computational Networks. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 425, 1868 (Jan. 1997), 73–90. https://doi.org/10.1098/rspa.1989.0099

[17] Cirq Developers. 2021. Cirq. Zenodo. https://doi.org/10.5281/zenodo.5182845

[18] Qiskit Developers. 2021. Qiskit: An Open-Source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2573505

[19] Egretta.Thula. 2023. Answer to "Why Does Composing a Clifford Circuit to Another Circuit Not Work? (Qiskit)" - Quantum Computing Stack Exchange.

[20] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. *ACM SIGPLAN Notices* 37, 5 (May 2002), 234–245. https://doi.org/10.1145/543552.512558

[21] Daniel Fortunato, José Campos, and Rui Abreu. 2022. Mutation Testing of Quantum Programs Written in QISKit. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 358–359. https://doi.org/10.1145/3510454.3528649

[22] Daniel Fortunato, José Campos, and Rui Abreu. 2022. QMutPy: A Mutation Testing Tool for Quantum Algorithms and Applications in Qiskit. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 797–800. https://doi.org/10.1145/3533767.3543296

[23] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 945–956. https://doi.org/10.1109/ICSE48619.2023.00087

[24] Google. 2015. Error Prone: static analysis tool for Java. http://errorprone.info/.

[25] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 317–328. https://doi.org/10.1145/3238147.3238213

[26] Yipeng Huang and Margaret Martonosi. 2019. Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association

for Computing Machinery, New York, NY, USA, 541–553. https://doi.org/10.1145/3307650.3322213

[27] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2014. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2597917.2597939

[28] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, San Francisco, CA, USA, 672–681.

[29] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[30] Maximilian Kaul, Alexander Küchler, and Christian Banse. 2023. A Uniform Representation of Classical and Quantum Source Code for Static Code Analysis. https://doi.org/10.48550/arXiv.2308.06113 arXiv:2308.06113 [cs]

[31] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[32] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. QASMBench: A Low-Level Quantum Benchmark Suite for NISQ Evaluation and Simulation. *ACM Transactions on Quantum Computing* 4, 2 (Feb. 2023), 10:1–10:26. https://doi.org/10.1145/3550488

[33] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 150:1–150:29. https://doi.org/10.1145/3428218

[34] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[35] Peixun Long and Jianjun Zhao. 2023. Equivalence, Identity, and Unitarity Checking in Black-Box Testing of Quantum Programs. https://doi.org/10.48550/arXiv.2307.01481 arXiv:2307.01481 [quant-ph]

[36] Junjie Luo, Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. 2022. A Comprehensive Study of Bug Fixes in Quantum Programs. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1239–1246. https://doi.org/10.1109/SANER53432.2022.00147

[37] Andriy Miranskyy, Lei Zhang, and Javad Doliskani. 2020. Is Your Quantum Program Bug-Free? *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (June 2020), 29–32. https://doi.org/10.1145/3377816.3381731 arXiv:2001.10870

[38] Michael A Nielsen, Isaac Chuang, and Lov K Grover. 2002. Quantum Computation and Quantum Information. *Am. J. Phys.* 70, 5 (2002), 4.

[39] Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum Computing Platforms: An Empirical Study. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 86:1–86:27. https://doi.org/10.1145/3527330

[40] Matteo Paltenghi and Michael Pradel. 2023. MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 2413–2424. https://doi.org/10.1109/ICSE48619.2023.00202

[41] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. 2021. Unqomp: Synthesizing Uncomputation in Quantum Circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 222–236. https://doi.org/10.1145/3453483.3454040

[42] Simon Perdrix. 2008. Quantum Entanglement Analysis Based on Abstract Interpretation. In *Static Analysis (Lecture Notes in Computer Science)*, María Alpuente and Germán Vidal (Eds.). Springer, Berlin, Heidelberg, 270–282. https://doi.org/10.1007/978-3-540-69166-2_18

[43] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. T|ket⟩: A Retargetable Compiler for NISQ Devices. *Quantum Science and Technology* 6, 1 (Nov. 2020), 014003. https://doi.org/10.1088/2058-9565/ab8e92

[44] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3183895.3183901

[45] user19571. 2022. Question: "Remove Inactive Qubits from Qiskit Circuit" - Quantum Computing Stack Exchange.
[46] Jiyuan Wang, Fucheng Ma, and Yu Jiang. 2021. Poster: Fuzz Testing of Quantum Program. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 466–469. https://doi.org/10.1109/ICST49551.2021.00061
[47] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 692–704. https://doi.org/10.1109/ASE51524.2021.9678792
[48] Manuela Weigold, Johanna Barzen, Frank Leymann, and Marie Salm. 2021. Encoding Patterns for Quantum Algorithms. *IET Quantum Communication* 2, 4 (2021), 141–152. https://doi.org/10.1049/qtc2.12032
[49] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3597503.3639121
[50] Shangzhou Xia and Jianjun Zhao. 2023. Static Entanglement Analysis of Quantum Programs. In *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*. 42–49. https://doi.org/10.1109/Q-SE59154.2023.00013
[51] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44
[52] Nengkun Yu and Jens Palsberg. 2021. Quantum Abstract Interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 542–558. https://doi.org/10.1145/3453483.3454061
[53] Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. 2023. Bugs4Q: A Benchmark of Existing Bugs to Enable Controlled Testing and Debugging Studies for Quantum Programs. *Journal of Systems and Software* 205 (Nov. 2023), 111805. https://doi.org/10.1016/j.jss.2023.111805
[54] Pengzhan Zhao, Xiongfei Wu, Zhuo Li, and Jianjun Zhao. 2023. QChecker: Detecting Bugs in Quantum Programs via Static Analysis. https://doi.org/10.48550/arXiv.2304.04387 arXiv:2304.04387 [cs]
[55] Pengzhan Zhao, Jianjun Zhao, and Lei Ma. 2021. Identifying Bug Patterns in Quantum Programs. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. 16–21. https://doi.org/10.1109/Q-SE52541.2021.00011
[56] Pengzhan Zhao, Jianjun Zhao, Zhongtao Miao, and Shuhan Lan. 2021. Bugs4Q: A Benchmark of Real Bugs for Quantum Programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE51524.2021.9678908