Bugs in Quantum Computing Platforms and How to Detect Them Automatically

Michael Pradel
Software Lab – University of Stuttgart

Joint work with Matteo Paltenghi
Quantum Computing

- Classical computers are reaching their limits, but demand for computation keeps increasing
- Quantum computers: New kind of hardware
  - Builds on results in quantum physics
  - Computing power scales exponentially with number of qubits
Quantum Computing Platforms

Program runs on Platform runs on Hardware
Quantum Computing Platforms

- **Program**: Written in C, Java, Python
- **Platform**: gcc, Java VM
- **Hardware**: CPU, GPU
Quantum Computing Platforms

Program

Platform

Hardware

E.g., written in Python with Qiskit
E.g., compiler, execution environment, simulator
Quantum computer
## Typical Components of Platforms

<table>
<thead>
<tr>
<th>Quantum programming language</th>
<th>Compilation</th>
<th>Execution environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantum abstractions</td>
<td>Intermediate representations</td>
<td>Interface to quantum computer</td>
</tr>
<tr>
<td>Classical abstractions</td>
<td>Optimizations</td>
<td>Simulator</td>
</tr>
<tr>
<td>Domain-specific abstractions</td>
<td>Machine code generation</td>
<td>Quantum state evaluation</td>
</tr>
</tbody>
</table>

### Auxiliary components

- Testing
- Visualization and plotting
- Infrastructure scripts and glue code
Bugs in Platforms

Quantum computing platforms:
Complex software, likely to contain bugs

- Annoying for users
- Cause misleading results
- Hinder progress in quantum computing applications

50 years of research on testing traditional compilers:
A Survey of Compiler Testing (ACM CSUR’19)
Example

Program written in Qiskit:

```python
# initialize quantum registers and
# classical registers
q_registers = QuantumRegister(3)
c_registers = ClassicalRegister(3)

# create a quantum circuit
circ = QuantumCircuit(q_registers, c_registers)
circ.x(0)
circ.h(1)
circ.cx(1, 2)

# measure qubits and write results
# to classical registers
circ.measure(q_registers[0], c_registers[1])
circ.measure(q_registers[1], c_registers[0])
circ.measure(q_registers[2], c_registers[2])

https://github.com/Qiskit/qiskit-terra/issues/2704
```
Example

**Program written in Qiskit:**

```python
# initialize quantum registers and classical registers
q_registers = QuantumRegister(3)
c_registers = ClassicalRegister(3)

circ = QuantumCircuit(q_registers, c_registers)
circ.x(0)
circ.h(1)
circ.cx(1, 2)

circ.measure(q_registers[0], c_registers[1])
circ.measure(q_registers[1], c_registers[0])
circ.measure(q_registers[2], c_registers[2])
```

Qubits: Superpositions of 0 and 1

https://github.com/Qiskit/qiskit-terra/issues/2704
Example

Program written in Qiskit:

```python
# initialize quantum registers and
# classical registers
q_registers = QuantumRegister(3)
c_registers = ClassicalRegister(3)

# create a quantum circuit
circ = QuantumCircuit(q_registers, c_registers)
circ.x(0)
circ.h(1)
circ.cx(1, 2)

# measure qubits and write results
# to classical registers
circ.measure(q_registers[0], c_registers[1])
circ.measure(q_registers[1], c_registers[0])
circ.measure(q_registers[2], c_registers[2])
```

Circuit built from gates that operate on qubits

https://github.com/Qiskit/qiskit-terra/issues/2704
Example

Program written in Qiskit:

```python
# initialize quantum registers and
classical registers
q_registers = QuantumRegister(3)
c_registers = ClassicalRegister(3)

# create a quantum circuit
circ = QuantumCircuit(q_registers, c_registers)
circ.x(0)
circ.h(1)
circ.cx(1, 2)

# measure qubits and write results
to classical registers
circ.measure(q_registers[0], c_registers[1])
circ.measure(q_registers[1], c_registers[0])
circ.measure(q_registers[2], c_registers[2])
```

After computing, measure results into classical bits

https://github.com/Qiskit/qiskit-terra/issues/2704
Example

Visual representation of the program:

\[
\begin{align*}
q_{00} & : X \\
q_{01} & : H \\
q_{02} & : \\
c_{0} & : /_3
\end{align*}
\]
Example

Output distribution:

Expected

Actual

https://github.com/Qiskit/qiskit-terra/issues/2704
This Talk

Part 1: Empirical study of bugs in 18 popular quantum computing platforms

Bugs in Quantum Computing Platforms: An Empirical Study (OOPSLA’22)

Part 2: Metamorphic testing to automatically detect bugs

MorphQ: Metamorphic Testing of Quantum Computing Platforms (arXiv)
Empirical Study of Bugs

RQ1: How many bugs are quantum-specific?

RQ2: Where do the bugs occur?

RQ3: How do the bugs manifest?

RQ4: Any recurrent bug patterns?

RQ5: How complex are the bug fixes?
Methodology

18 platforms with
24,857 commits

2,140 bug candidates

223 bugs
Methodology

18 platforms with 24,857 commits
2,140 bug candidates
223 bugs
Methodology

18 platforms with 24,857 commits
2,140 bug candidates
223 bugs

Commits message
- with “fix” in first line
- refer to issues or pull requests
- no mention of “refactor”, “typo”, etc.
Methodology

18 platforms with 24,857 commits

2,140 bug candidates

223 bugs

- Manual inspection of 20 candidates per project
- Keep actual bug fixes
- Disentangle commits into individual bug fixes
RQ1: Quantum-Specific Bugs

How many of the bugs are quantum-specific?

- Mistake is in handling quantum-specific concept
- Fixing requires knowledge of quantum computing
Examples

Classical bugs, i.e., not quantum-specific:

```python
for field in fields:
-    if not hasattr(field, '__options__):
+    if not hasattr(self.__options, field):
        raise AttributeError(
            "Options field %s is not valid for this "
            "backend" % field)
```

Misuse of Python API

```c++
case Operations::OpType::diagonal_matrix:
-    BaseState::qreg_.apply_diagonal_matrix(op.qubits, op.params);
+    BaseState::qreg_.apply_diagonal_unitary_matrix(op.qubits, op.params);
```
Examples

Quantum-specific bugs:

```python
def is_identity(term):
    return len(term) == 0
+    if isinstance(term, PauliTerm):
+        return len(term) == 0 and not np.isclose(term.coefficient, 0)
+    elif isinstance(term, PauliSum):
+        return len(term.terms) == 1 and (len(term.terms[0]) == 0) and 
+        not np.isclose(term.terms[0].coefficient, 0)
+    else:
+        raise TypeError("is_identity only checks PauliTerms and PauliSum objects!")
```

Overlooked corner case of Pauli terms

```python
def fold_global(circuit: QPROGRAM, stretch: float, **kwargs) -> QPROGRAM:
    
    
    
+    num_to_fold = int(round(fractional_stretch * len(ops) / 2))
    
    
    if num_to_fold > 0:
        folded += Circuit([inverse(ops[-num_to_fold:]), [ops[-num_to_fold:]]])
```

Incorrect numerical computation
RQ1: Quantum-Specific Bugs (2)

\[ \frac{89}{223} = 39.9\% \text{ of bugs are quantum-specific} \]
RQ1: Quantum-Specific Bugs (2)

89/223 = 39.9% of bugs are quantum-specific

- Existing bug detectors: Still helpful
- But: Quantum-specific techniques are also needed
RQ2: Where Do Bugs Occur?

Distribution across components:
RQ2: Where Do Bugs Occur?

Distribution across components:

Classical bugs affect infrastructural scripts and testing code.
RQ2: Where Do Bugs Occur?

Bug detection opportunities in compilation and optimization
RQ3: How Do Bugs Manifest?

Bug symptoms

Functional

Application-specific
- Failing test: 16
  - Incorrect visualization: 2
  - Incorrect final measurement: 15
- Incorrect output: 77

Generic
- Compilation error: 2
- Crash: 92
  - OS/PL level: 67
  - Application level: 25

Non-functional
- Inefficiency: 8
- Other: 6
RQ3: How Do Bugs Manifest?

Most common: Crashes and incorrect output
RQ3: How Do Bugs Manifest? (2)

Symptoms vs. kind of bug:
RQ3: How Do Bugs Manifest? (2)

Symptoms vs. kind of bug:

Many quantum-specific bugs: Difficult to detect
RQ4: Recurring Bug Patterns

- Manually annotated based on code change, commit message, issue and/or pull request
- One bug may fit multiple patterns
RQ4: Recurring Bug Patterns (2)

![Bug Pattern Diagram]

- **API-related**
  - API misuse: 13
  - Outdated API client: 13

- **Incorrect application logic**
  - Intermediate representation
    - Missing information: 8
    - Wrong information: 21
  - Overlooked corner case: 40
  - Refer to wrong program element
    - Wrong concept: 9
    - Wrong identifier: 11
  - Qubit-related
    - Incorrect qubit order: 5
    - Incorrect qubit count: 3
  - Incorrect scheduling: 5
  - Incorrect numerical computation: 11
  - Incorrect randomness handling: 5

- **Math-related**

- **Qubit-related**
  - Incorrect qubit order: 5
  - Incorrect qubit count: 3

- **Intermediate representation**
  - Missing information: 8
  - Wrong information: 21

- **Overlooked corner case**
  - Wrong concept: 9
  - Wrong identifier: 11

- **API misuse**
  - Outdated API client: 13

- **Refer to wrong program element**
  - Wrong concept: 9
  - Wrong identifier: 11
RQ4: Recurring Bug Patterns (2)

- **Quantum-specific patterns**
  - **Math-related**
    - Incorrect randomness handling
    - Incorrect numerical computation
  - **Qubit-related**
    - Incorrect qubit order
    - Incorrect qubit count
    - MSB-LSB convention mismatch
    - Incorrect scheduling
  - **API-related**
    - API misuse
    - Outdated API client
    - Missing information
    - Wrong information
    - Wrong concept
    - Wrong identifier
    - Wrong in-formation
    - Overlooked corner case
    - Refer to wrong program element
  - **Intermediate representation**
    - Overlooked information
    - Missing information
  - **API-related**
    - Outdated API client
    - API misuse
  - **Math-related**
    - Incorrect randomness handling
    - Incorrect numerical computation
    - Incorrect randomness handling
    - API misuse
    - Outdated API client
    - Missing information
    - Wrong information
    - Correct concept
    - Wrong identifier
    - Overlooked corner case
    - Refer to wrong program element
    - MSB-LSB convention mismatch
    - Incorrect qubit order
    - Incorrect qubit count
    - Overlooked information
    - Missing information
    - API misuse
    - Outdated API client

RQ5: Complexity of Bug Fixes

How complex are the code changes that fix bugs?

![Graph showing bug-fix complexity distribution](image-url)
RQ5: Complexity of Bug Fixes

How complex are the code changes that fix bugs?

Possibly in reach for today’s repair techniques
### Implications

<table>
<thead>
<tr>
<th>39.9% quantum-specific</th>
<th>Need new bug detection techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many bugs in “inner” components</td>
<td>Can find bugs on simulators (no need for quantum hardware)</td>
</tr>
<tr>
<td>Crash is most common symptom</td>
<td>“Easy” oracle for testing</td>
</tr>
<tr>
<td>New, quantum-specific bug patterns</td>
<td>Opportunities for bug detectors</td>
</tr>
<tr>
<td>Many few-line fixes</td>
<td>Opportunities for program repair</td>
</tr>
</tbody>
</table>
This Talk

Part 1: Empirical study of bugs in 18 popular quantum computing platforms

Bugs in Quantum Computing Platforms: An Empirical Study (OOPSLA’22)

Part 2: Metamorphic testing to automatically detect bugs

MorphQ: Metamorphic Testing of Quantum Computing Platforms (arXiv)
Key Challenges

Challenges for automatically finding bugs in quantum computing platforms:

1) Few quantum programs
2) Oracle problem
Automated, metamorphic testing of quantum computing platforms *

- Generator of quantum programs
- Ten metamorphic transformations

* Current target platform: Qiskit
Automated, metamorphic testing of quantum computing platforms *

- Generator of quantum programs
- Ten metamorphic transformations

Transform a source program into a follow-up program such that the behaviors of the two programs have a known relationship

* Current target platform: Qiskit
Overview of MorphQ

1. Program Generation
   - Source
     - q₀
     - q₁
   - Exec. Settings
     - A
     - B
     - D
     - E

2. Metamorphic Transformation
   - Follow-Up
     - q₀
     - q₁
     - A
     - B
     - D
     - C
     - X
     - X
     - E
   - Exec. Settings

3. Execution & Check Behavior
   - outₛ
     - no crash
     - crash
   - outₖ
     - no crash
     - crash

same? (R_OUTPUT)
Program Generation

- **Naive approach**: Randomly combine calls of Qiskit APIs
  - Problem: Mostly invalid programs (crashes, no deep testing)

- **Instead**: Combine *template-based* and *grammar-based* generation
  - Generates non-crashing programs by design
# Section: Prologue
<ALL_IMPORTS>

# Section: Circuit
qr = QuantumRegister(<N_QUBITS>, name='qr')
cr = ClassicalRegister(<N_QUBITS>, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
<GATE_OPS>

# Section: Measurement
qc.measure(qr, cr)

# Section: Transpilation/compilation
qc = transpile(qc,
    basis_gates=<TARGET_GATE_SET>,
    optimization_level=<OPT_LEVEL>,
    coupling_map=<COUPLING_MAP>)

# Section: Execution
simulator = Aer.get_backend(<BACKEND_NAME>)
counts = execute(qc, backend=simulator,
    shots=<N_SHOTS>).result().get_counts(qc)
Template of Quantum Programs

# Section: Prologue
<ALL_IMPORTS>

# Section: Circuit
qr = QuantumRegister(<N_QUBITS>, name='qr')
cr = ClassicalRegister(<N_QUBITS>, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
<GATE_OPS>

# Section: Measurement
qc.measure(qr, cr)

# Section: Transpilation/compilation
qc = transpile(qc,
    basis_gates=<TARGET_GATE_SET>,
    optimization_level=<OPT_LEVEL>,
    coupling_map=<COUPLING_MAP>)

# Section: Execution
simulator = Aer.get_backend(<BACKEND_NAME>)
counts = execute(qc, backend=simulator,
    shots=<N_SHOTS>).result().get_counts(qc)
Template of Quantum Programs

# Section: Prologue
<ALL_IMPORTS>

# Section: Circuit
qr = QuantumRegister(<N_QUBITS>, name='qr')
cr = ClassicalRegister(<N_QUBITS>, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
<GATE_OPS>

# Section: Measurement
qc.measure(qr, cr)

# Section: Transpilation/compilation
qc = transpile(qc,
    basis_gates=<TARGET_GATE_SET>,
    optimization_level=<OPT_LEVEL>,
    coupling_map=<COUPLING_MAP>)

# Section: Execution
simulator = Aer.get_backend(<BACKEND_NAME>)
counts = execute(qc, backend=simulator,
    shots=<N_SHOTS>).result().get_counts(qc)

Grammar to generate sequence of gates that operate on available qubits
## Metamorphic Transformations

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit transf.</td>
<td>Change of qubit order</td>
<td>Equivalence modulo mapping of measurements</td>
</tr>
<tr>
<td></td>
<td>Inject null-effect operation</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Add quantum register</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Inject parameters</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Partitioned execution</td>
<td>Cartesian product of output distributions</td>
</tr>
<tr>
<td>Repres. transf.</td>
<td>Roundtrip conversion via QASM</td>
<td>Equivalence</td>
</tr>
<tr>
<td>Execution transf.</td>
<td>Change of coupling map</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Change of gate set</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Change of optimiz. level</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Change of backend</td>
<td>Equivalence</td>
</tr>
</tbody>
</table>
## Metamorphic Transformations

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit transf.</td>
<td>Change of qubit order</td>
<td>Equivalence modulo mapping of measurements</td>
</tr>
<tr>
<td></td>
<td>Inject null-effect operation</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Add quantum register</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Inject parameters</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Partitioned execution</td>
<td>Cartesian product of output distributions</td>
</tr>
<tr>
<td>Repres. transf.</td>
<td>Roundtrip conversion via QASM</td>
<td>Equivalence</td>
</tr>
<tr>
<td>Execution transf.</td>
<td>Change of coupling map</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Change of gate set</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Change of optimiz. level</td>
<td>Equivalence</td>
</tr>
<tr>
<td></td>
<td>Change of backend</td>
<td>Equivalence</td>
</tr>
</tbody>
</table>
Change of Qubit Order

Reorder qubits, and then adjust gates and measurements
Inject Null-Effect Operations

Exploit reversibility of quantum computations

```
subcirc = QuantumCircuit(qr, cr, name='subcirc')
subcirc.append(RXGate(6.12), qargs=[qr[0]], cargs=[])  # ... sequence of additional gates
qc.append(subcirc, qargs=qr, cargs=cr)
qc.append(subcirc.inverse(), qargs=qr, cargs=cr)
```
Inject Null-Effect Operations

Exploit reversibility of quantum computations

Randomly generated new subcircuit

```python
subcirc = QuantumCircuit(qr, cr, name='subcirc')
subcirc.append(RXGate(6.12), qargs=[qr[0]], cargs=[])  # ... sequence of additional gates
qc.append(subcirc, qargs=qr, cargs=cr)
qc.append(subcirc.inverse(), qargs=qr, cargs=cr)
```
Inject Null-Effect Operations

Exploit reversibility of quantum computations

Randomly generated new subcircuit

```python
subcirc = QuantumCircuit(qr, cr, name='subcirc')
subcirc.append(RXGate(6.12), qargs=[qr[0]], cargs=[])  # ... sequence of additional gates
qc.append(subcirc, qargs=qr, cargs=cr)
qc.append(subcirc.inverse(), qargs=qr, cargs=cr)
```

Insert both subcircuit and its inverse
Partitioned Execution

Find subsets of gate operations on disjoint sets of qubits, and move them into separate subcircuits.
Partitioned Execution

Find subsets of gate operations on disjoint sets of qubits, and move them into separate subcircuits.

Combine output distributions via Cartesian product.
Roundtrip via QASM

- OpenQASM: Assembly language shared across different platforms
- Transformation: **Export** to QASM and then **import** again

\[
qc = qc.from qasm \text{str}(qc.qasm())
\]
Change of Coupling Map

- **Coupling map**: Physical connections between qubits on the target hardware
  - Any map that correspond to a *connected graph* of qubits is legal

- **Transformation**: Replace coupling map with another *randomly generated* coupling map
Comparing Execution Behavior

- **Crash differences**: Source program terminates normally, but follow-up program crashes
- **Distribution differences**: Compare output distribution with Kolmogorov-Smirnov test
Evaluation: Warnings

How many warnings does MorphQ produce? (48 hours of automated testing)

<table>
<thead>
<tr>
<th>Warnings</th>
<th>No.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tested program pairs</td>
<td>8,360</td>
<td>100.0</td>
</tr>
<tr>
<td>→ Crashes in source program</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>→ Crashes in follow-up program</td>
<td>1,943</td>
<td>23.2</td>
</tr>
<tr>
<td>→ Successful executions</td>
<td>6,417</td>
<td>76.8</td>
</tr>
<tr>
<td>→ Distribution differences</td>
<td>56</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Evaluation: Real-World Bugs

Semi-automated clustering of warnings

Manual inspection of a random sample

13 bugs in the latest version of Qiskit
(9 confirmed so far)
Example 1

ValueError: too many subscripts in einsum

```python
qr = QuantumRegister(11, name='qr')
cr = ClassicalRegister(11, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
subcircuit = QuantumCircuit(qr, cr, name='subcirc')
subcircuit.x(3)
qc.append(subcircuit, qargs=qr, cargs=cr)
qc.x(3)
qc = transpile(qc, optimization_level=2)
```
Example 1

ValueError: too many subscripts in einsum

```python
qr = QuantumRegister(11, name='qr')
cr = ClassicalRegister(11, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
subcircuit = QuantumCircuit(qr, cr, name='subcirc')
subcircuit.x(3)
qc.append(subcircuit, qargs=qr, cargs=cr)
qc.x(3)
qc = transpile(qc, optimization_level=2)
```

Found via “Inject null-effect operations” and “Change of optimization level”
Example 1

ValueError: too many subscripts in einsum

Optimization fails for circuits with $\geq 11$ qubits

```python
qr = QuantumRegister(11, name='qr')
cr = ClassicalRegister(11, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
subcircuit = QuantumCircuit(qr, cr, name='subcirc')
subcircuit.x(3)
qc.append(subcircuit, qargs=qr, cargs=cr)
qc.x(3)
qc = transpile(qc, optimization_level=2)
```
Example 2

QasmError: 'subcirc' uses 4 qubits but is declared for 2 qubits

qr = QuantumRegister(2, name='qr')
cr = ClassicalRegister(2, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
subcircuit = QuantumCircuit(qr, cr, name='subcirc')
subcircuit.x(qr[0])
qc.append(subcircuit, qargs=qr, cargs=cr)
qc = QuantumCircuit.from_qasm_str(qc.qasm())
Example 2

QasmError: 'subcirc' uses 4 qubits but is declared for 2 qubits

```
qr = QuantumRegister(2, name='qr')
cr = ClassicalRegister(2, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
subcircuit = QuantumCircuit(qr, cr, name='subcirc')
subcircuit.x(qr[0])
qc.append(subcircuit, qargs=qr, cargs=cr)
qc = QuantumCircuit.from_qasm_str(qc.qasm())
```

Found via “Inject null-effect operations” and “Roundtrip conversion via QASM”
Example 2

QasmError: 'subcirc' uses 4 qubits but is declared for 2 qubits

Importer fails because of wrongly exported QASM

qr = QuantumRegister(2, name='qr')
cr = ClassicalRegister(2, name='cr')
qc = QuantumCircuit(qr, cr, name='qc')
subcircuit = QuantumCircuit(qr, cr, name='subcirc')
subcircuit.x(qr[0])
qc.append(subcircuit, qargs=qr, cargs=cr)
qc = QuantumCircuit.from_qasm_str(qc.qasm())
Precision of Warnings

- **Crash differences**: All true positives (except one cluster of warnings due to timeouts in an optimization)

- **Distribution differences**: Tens of inspected pairs, but all false positives due to randomness
Precision of Warnings

- **Crash differences**: All true positives (except one cluster of warnings due to timeouts in an optimization)

- **Distribution differences**: Tens of inspected pairs, but all false positives due to randomness

Opportunity for future work: Reliably comparing output distributions
Comparison with QDiff *

- **Disjoint sets of detected bugs**
- Testing with QDiff’s semantics-preserving code transformations (48 hours):
  - No crash differences
  - MorphQ achieves higher coverage: 6.1% vs. 8.1%

* QDiff: Differential Testing of Quantum Software Stacks, ASE’21
Conclusions

- **Quantum computing platforms:** Important target for automated testing
  - Quantum-specific bugs and bug patterns

- **MorphQ: First metamorphic testing technique**
  - Program generator that creates non-crashing programs
  - 10 metamorphic transformations
  - 13 bugs in Qiskit
This Talk

Part 1: Empirical study of bugs in 18 popular quantum computing platforms

*Bugs in Quantum Computing Platforms: An Empirical Study* (OOPSLA’22)

Part 2: Metamorphic testing to automatically detect bugs