Bugs in Quantum Computing Platforms and How to Detect Them Automatically

Michael Pradel

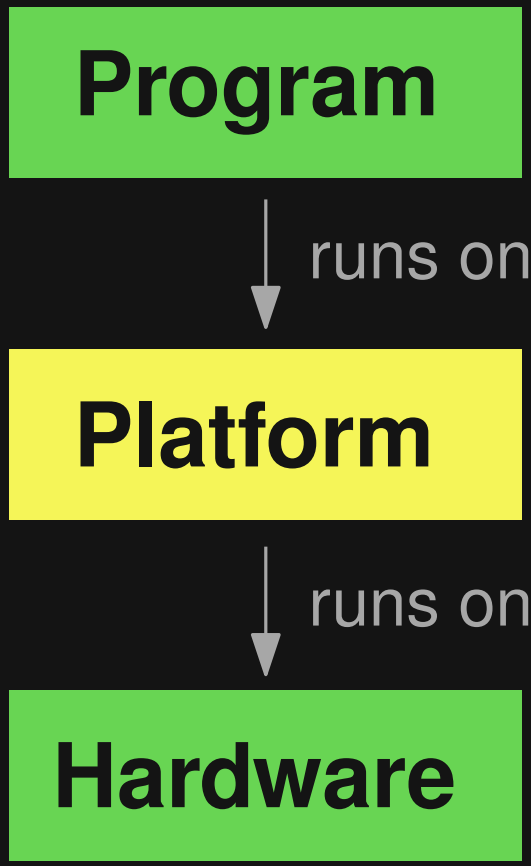
Software Lab – University of Stuttgart

Joint work with Matteo Paltenghi

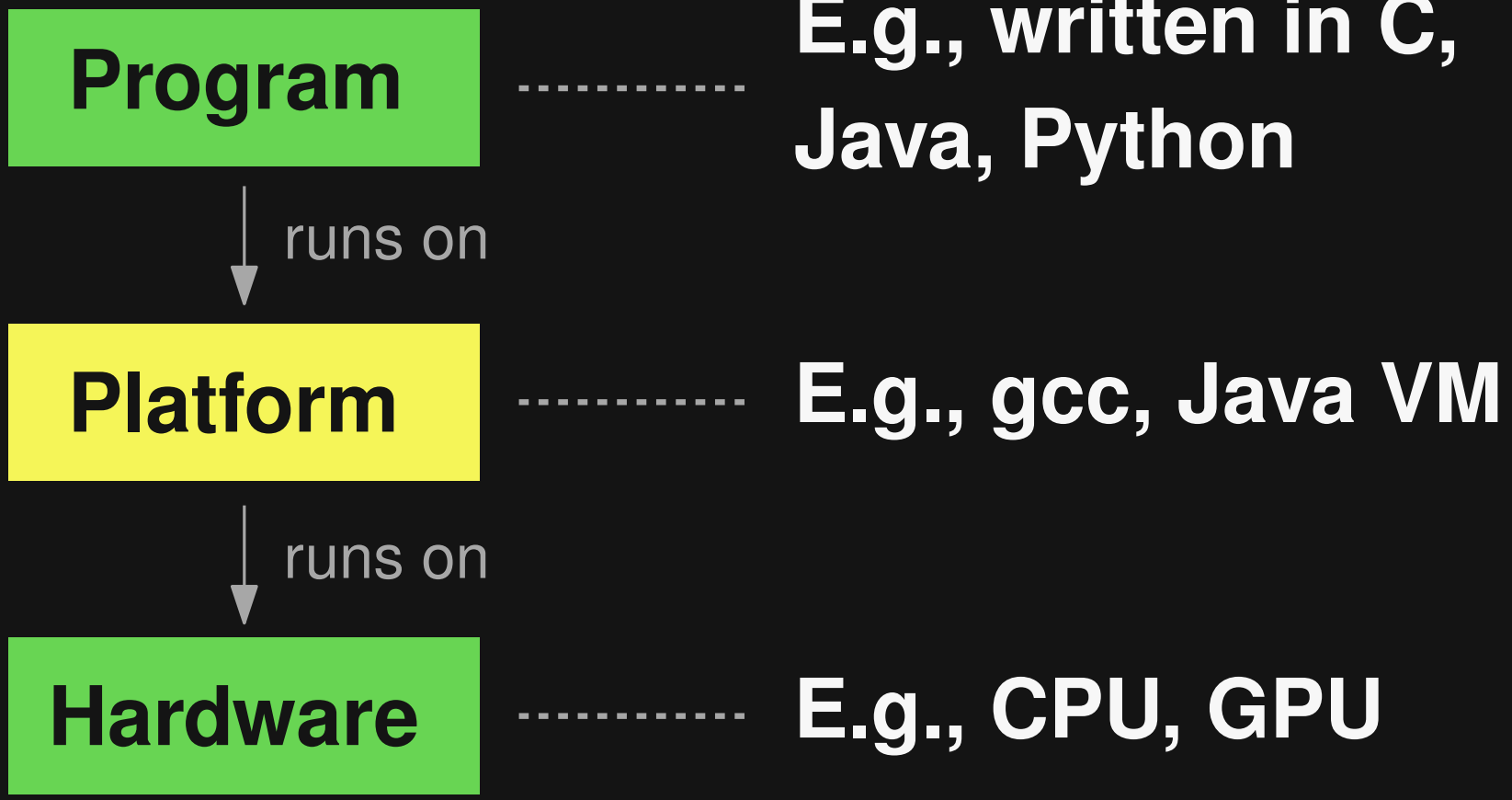
Quantum Computing

- **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

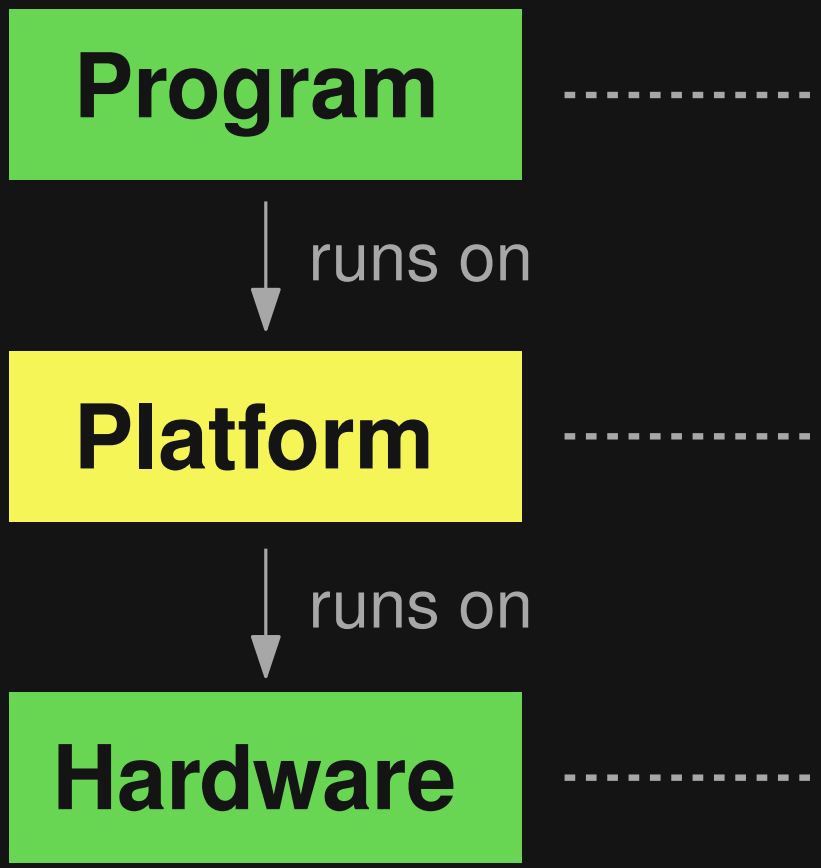
Quantum Computing Platforms



Quantum Computing Platforms



Quantum Computing Platforms

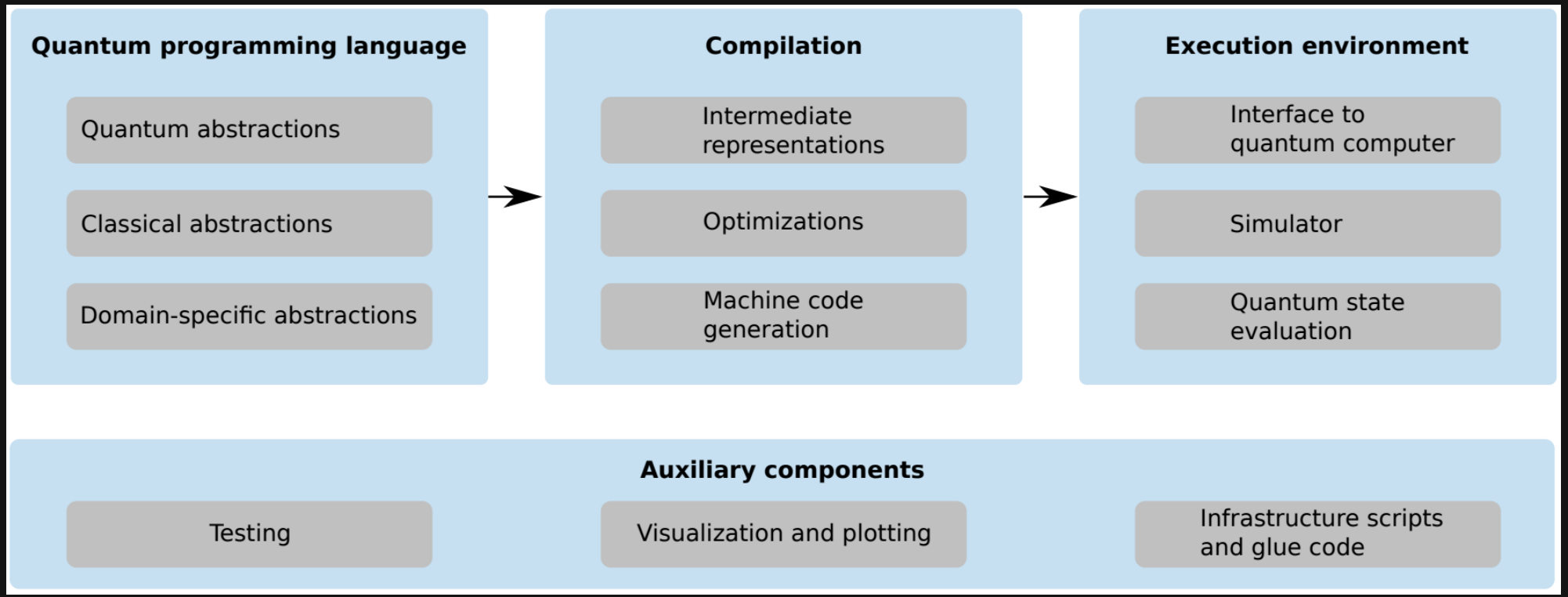


E.g., written in Python with Qiskit

E.g., compiler, execution environment, simulator

Quantum computer

Typical Components of Platforms



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:

```
# 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:

```
# 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])
```

Qubits: Superpositions of 0 and 1

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

Example

Program written in Qiskit:

```
# 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:

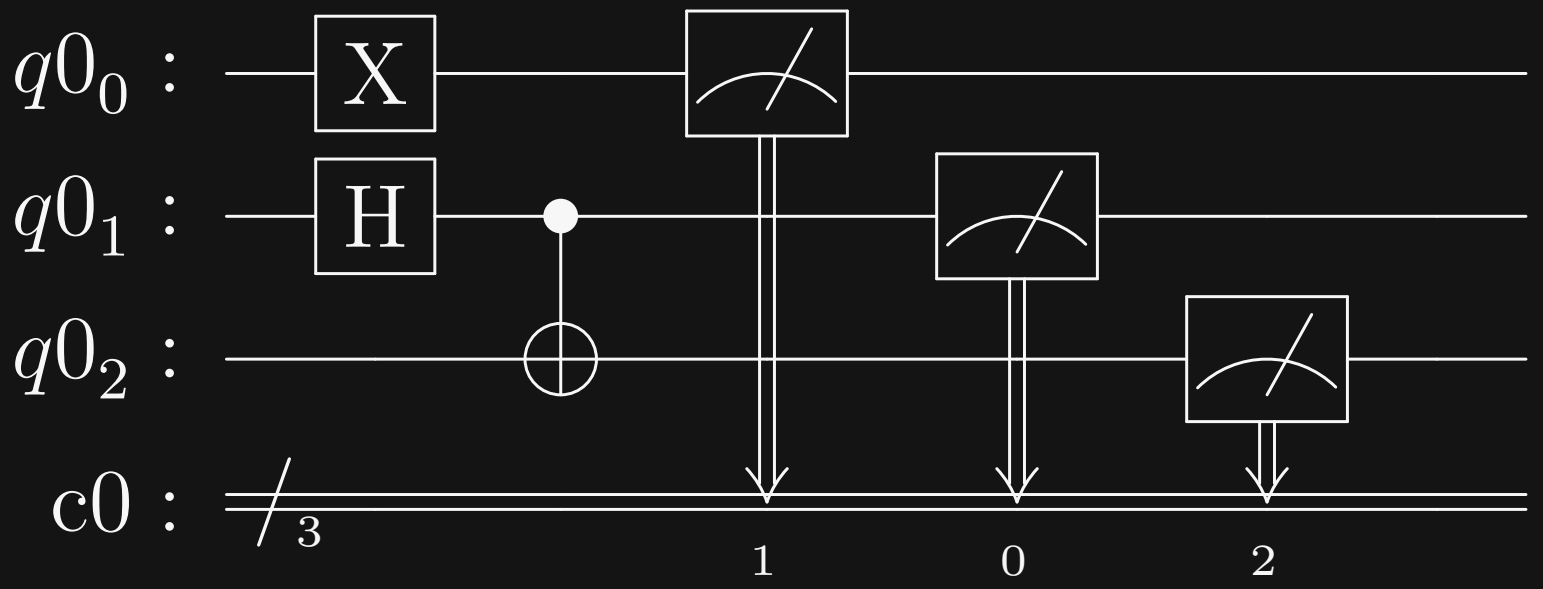
```
# initialize quantum registers and
#   classical registers
q_registers = QuantumRegister(3)
c_registers = ClassicalRegister(3)

# 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

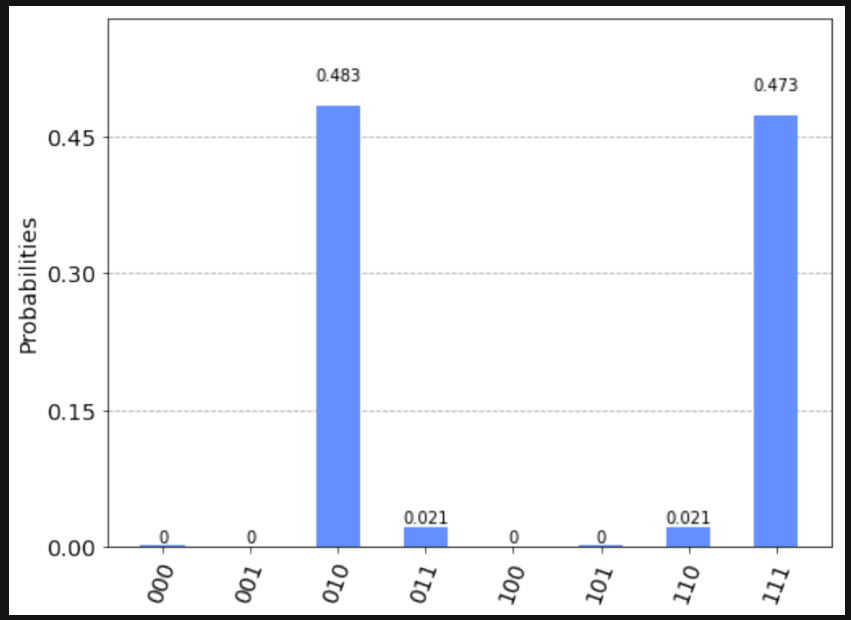
Example

Visual representation of the program:

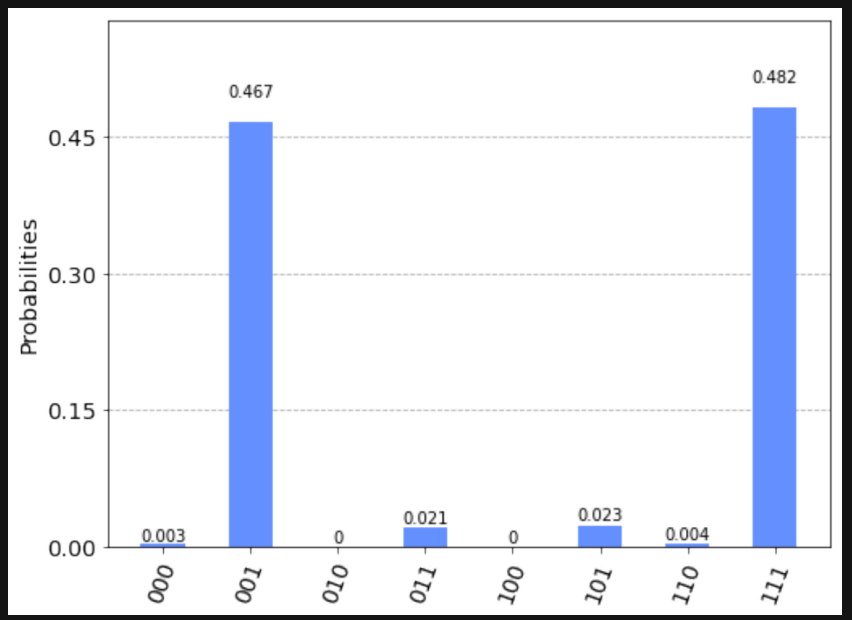


Example

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 c

2,140 bu

223 bugs

Repository	Components											Languages
	QA	CA	DA	IR	OPT	MCG	INTER	SIM	QSE	TEST	VIZ	
PennyLane	✓	✓	✓					✓	✓	✓	✓	Py
ProjectQ	✓	✓			✓	✓	✓	✓	✓	✓	✓	Py
OpenQL	✓	✓		✓	✓	✓	✓			✓	✓	C++, Py
Qiskit Aer	✓	✓			✓	✓		✓	✓	✓		C++, Py
Qiskit Ignis	✓		✓			✓			✓	✓		Py
Qiskit Terra	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Py
Tequila	✓	✓	✓		✓	✓	✓			✓		Py
Braket	✓	✓	✓	✓		✓	✓		✓	✓	✓	Py
Dwawe-System	✓	✓		✓		✓			✓	✓		Py
XACC	✓	✓		✓	✓	✓	✓	✓	✓	✓		C++, Py
QDK Libraries		✓	✓						✓	✓	✓	Q#
QDK Q# Compiler		✓		✓	✓	✓	✓			✓		C#, F#, Q#
QDK Q# Runtime	✓	✓						✓	✓	✓		C#, C++, Q#
Cirq	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	Py
Qualcs	✓	✓	✓					✓		✓		C++, C
Pyquil	✓					✓	✓	✓	✓	✓	✓	Py
mitiq	✓						✓		✓	✓		Py
StrawberryFields	✓		✓			✓	✓	✓	✓	✓	✓	Py

Methodology

18 platforms with

24,857 commits

2,140 bug candidates

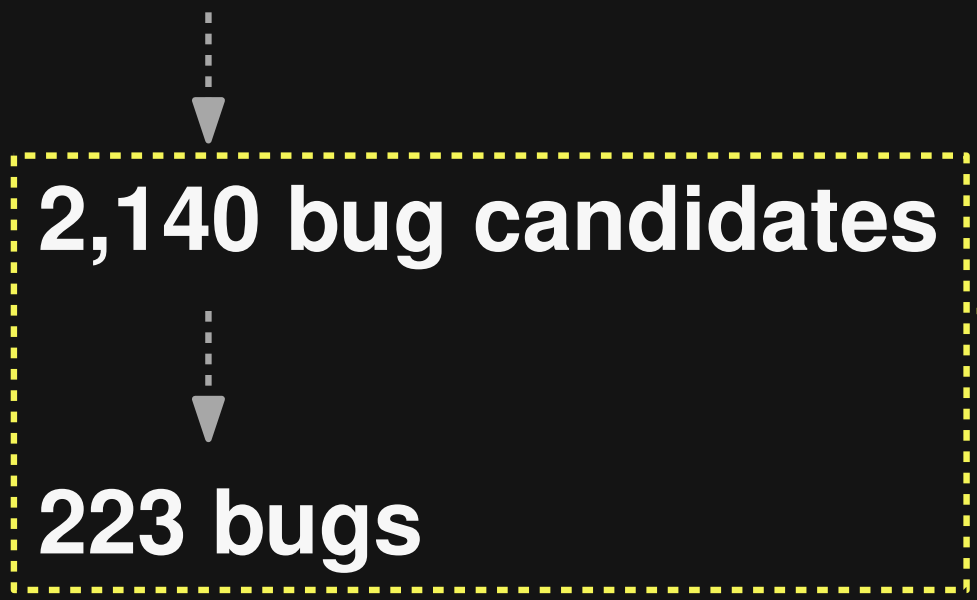
223 bugs

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



- 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:

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

Misuse of Python API

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

Function not in namespace

Examples

Quantum-specific bugs:

```
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

```
def fold_global(circuit: QPROGRAM, stretch: float, **kwargs) -> QPROGRAM:
    """Gives a circuit by folding the global unitary of the input circuit."""
    ...
    # Fold remaining gates until the stretch is reached
    ops = list(base_circuit.all_operations())
-    num_to_fold = int(round(fractional_stretch * len(ops)))
+    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)

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

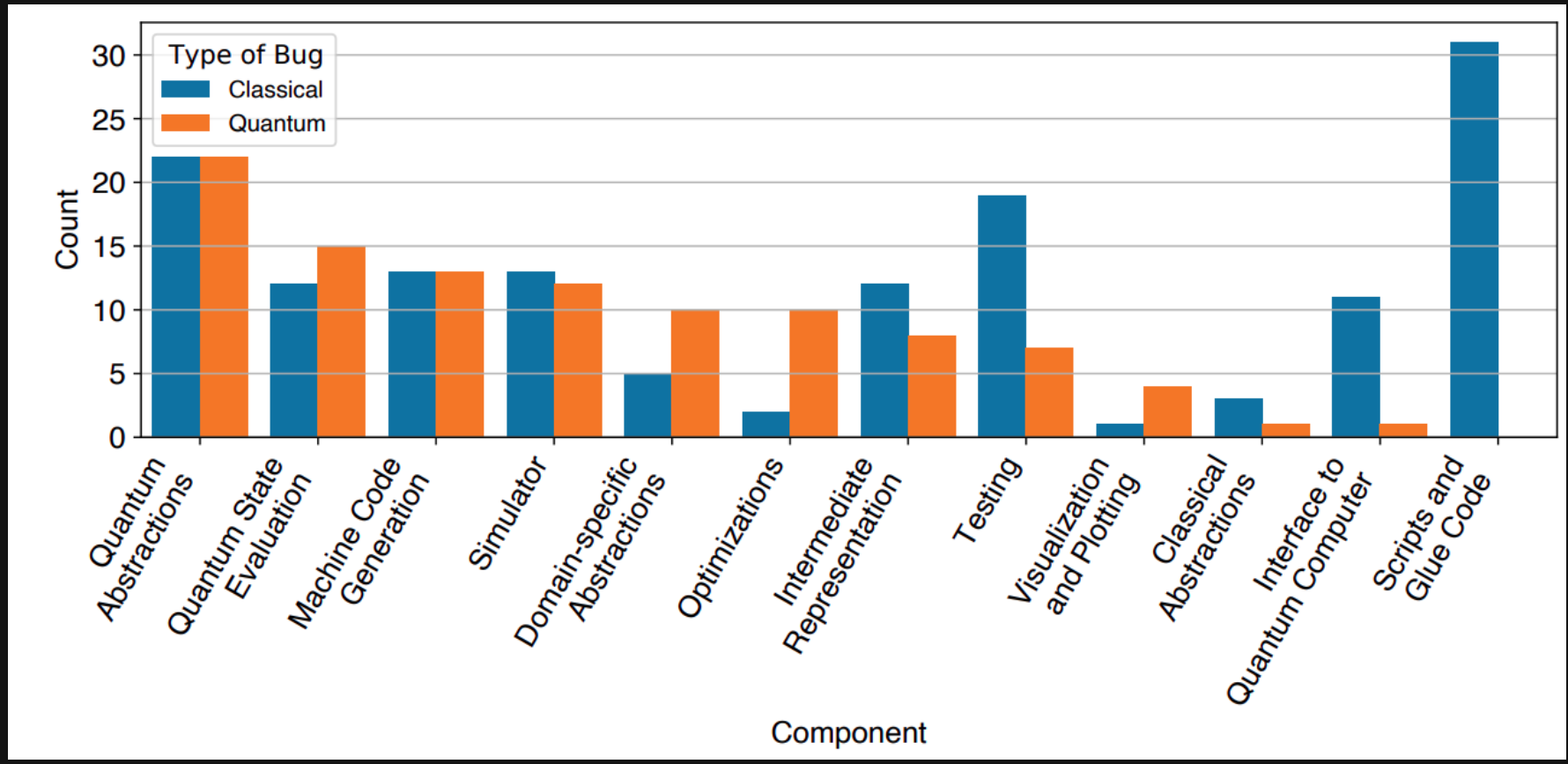
RQ1: Quantum-Specific Bugs (2)

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?

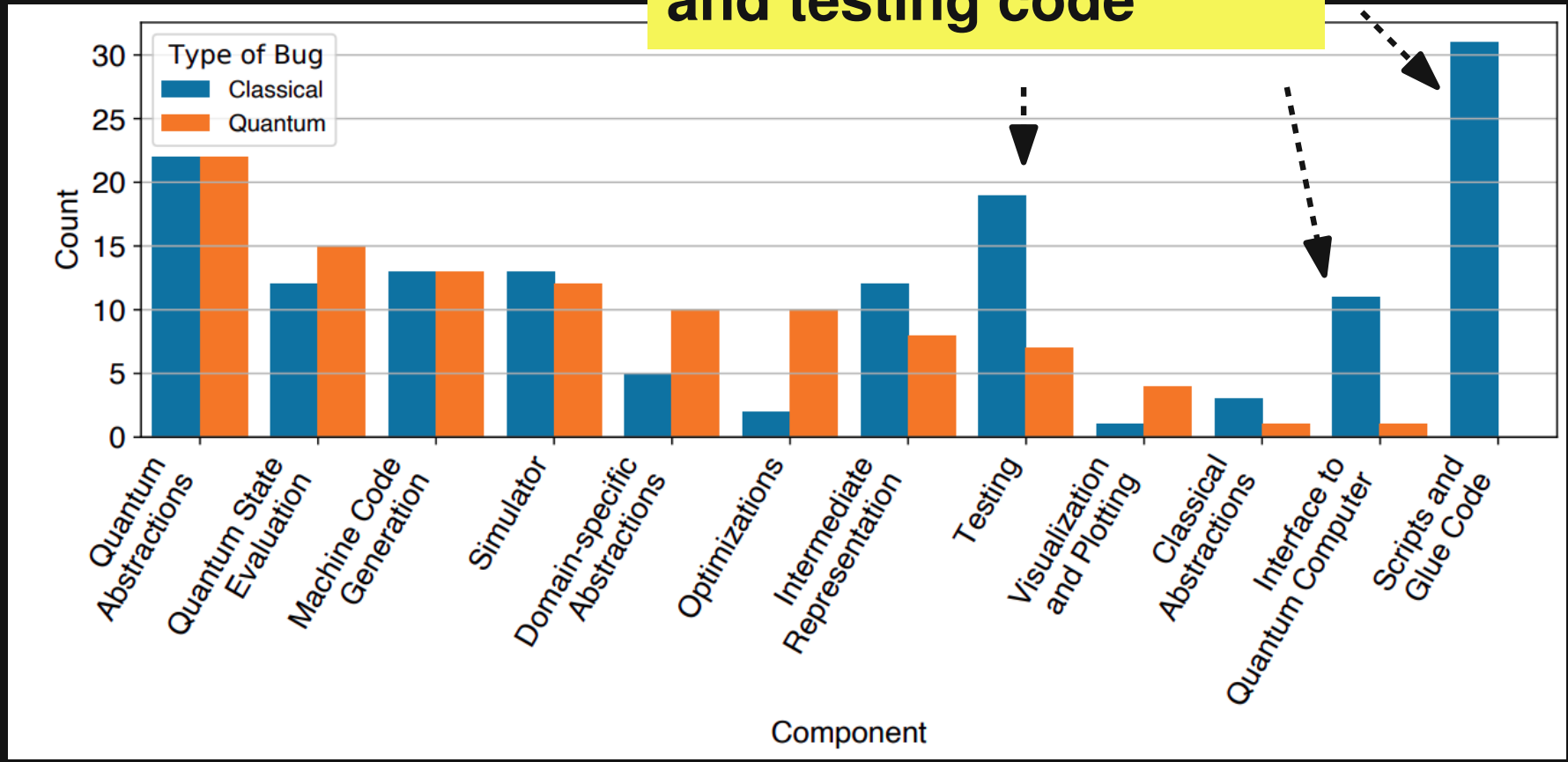
Distribution across components:



RQ2: Where Do Bugs Occur?

Distribution acros

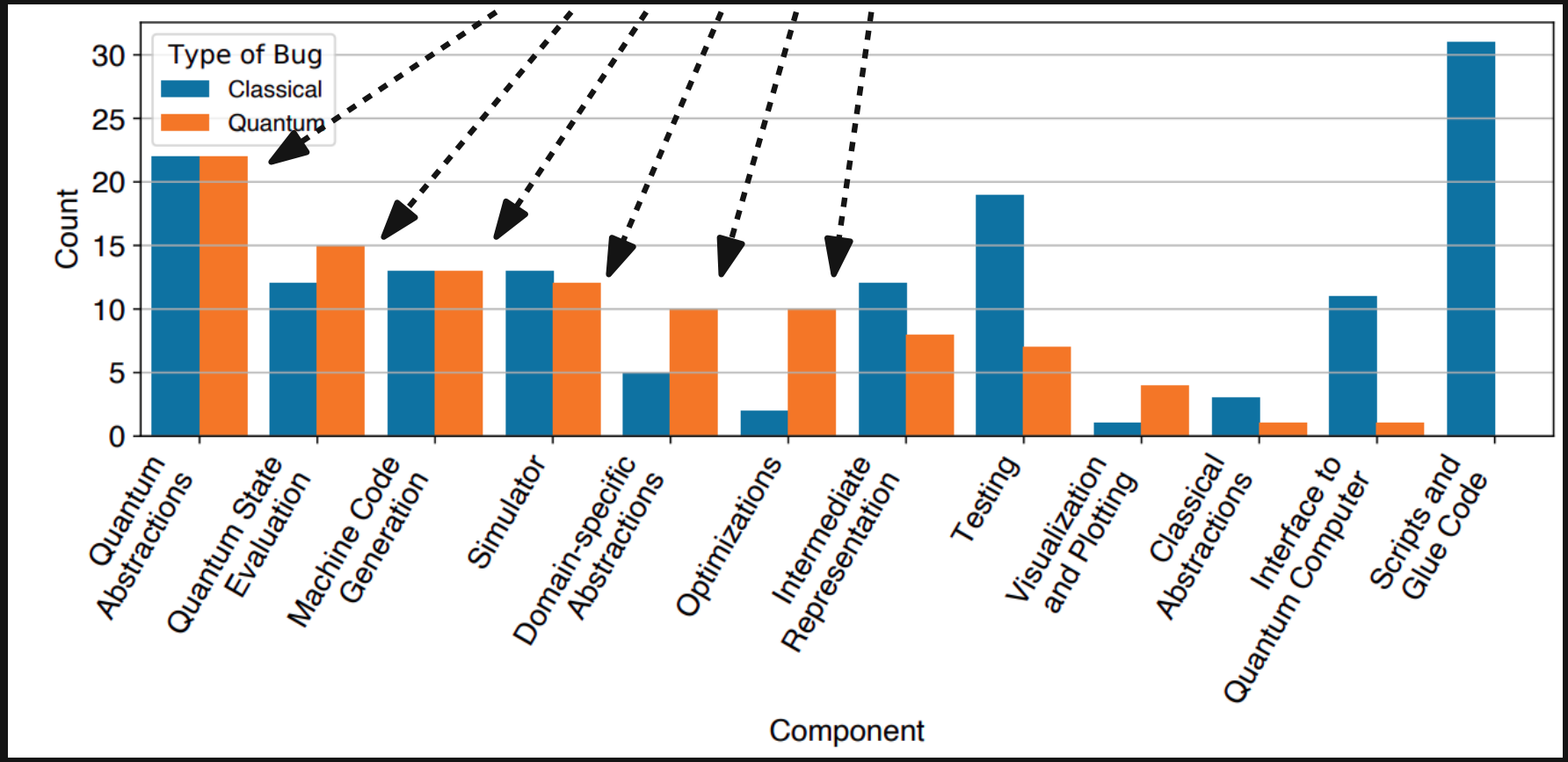
Classical bugs affect infrastructural scripts and testing code



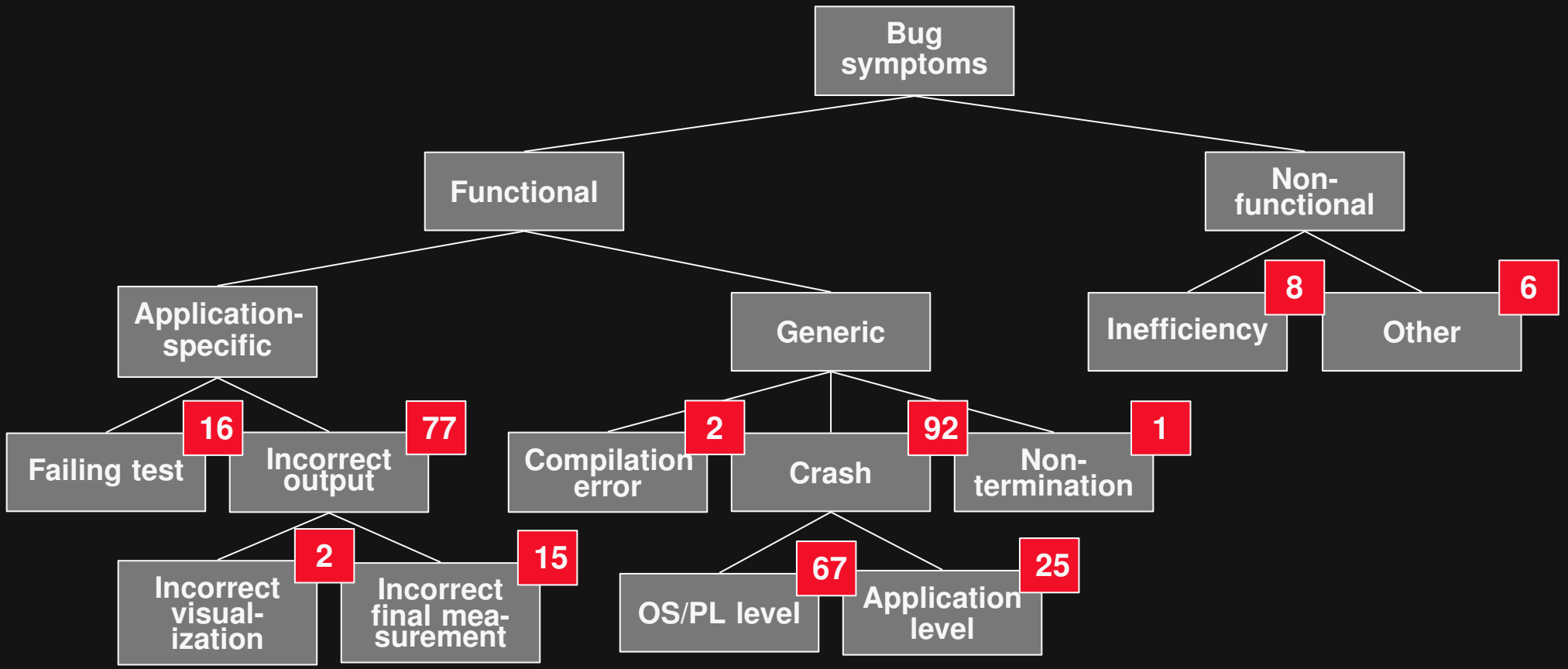
RQ2: Where Do Bugs Occur?

Distributior

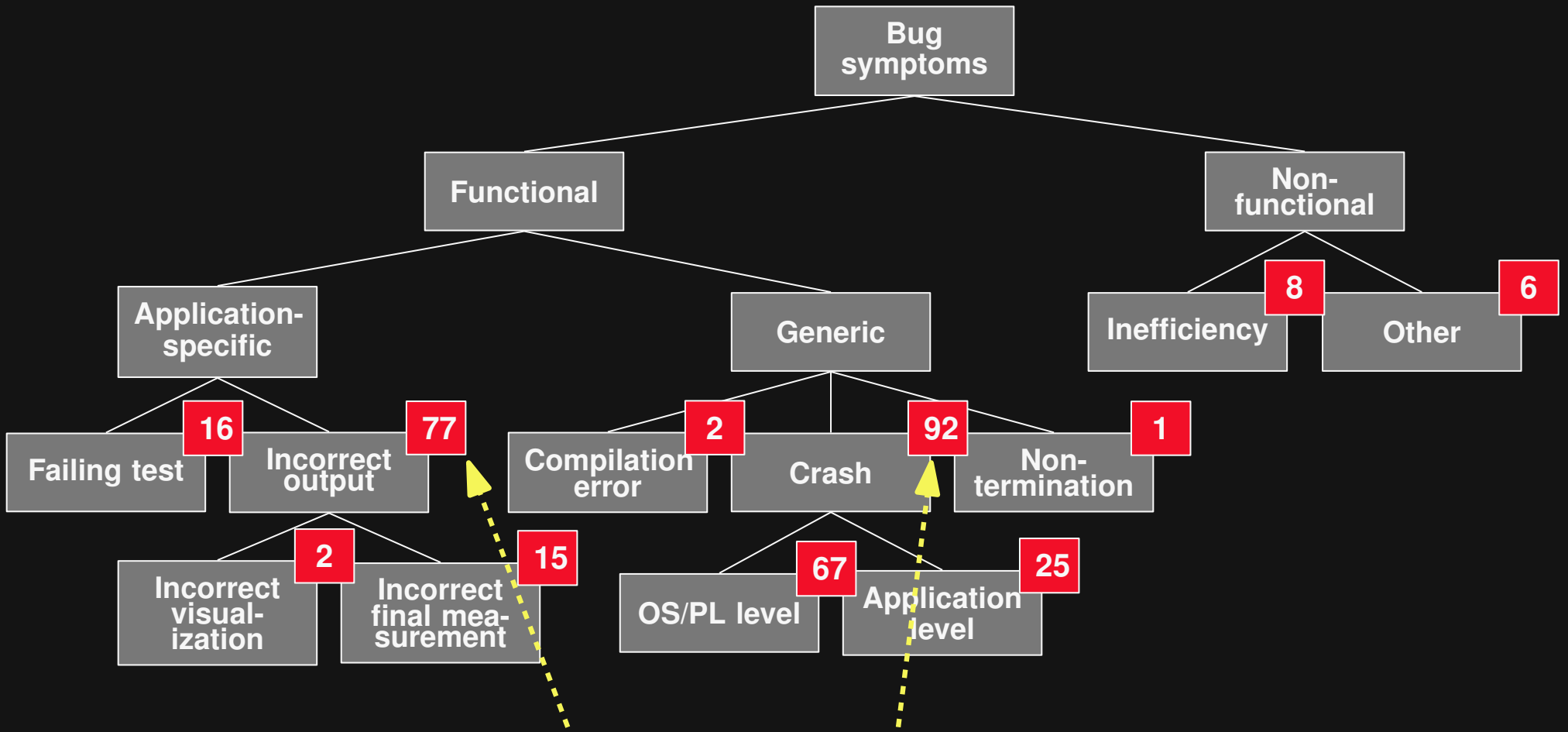
Bug detection opportunities in compilation and optimization



RQ3: How Do Bugs Manifest?



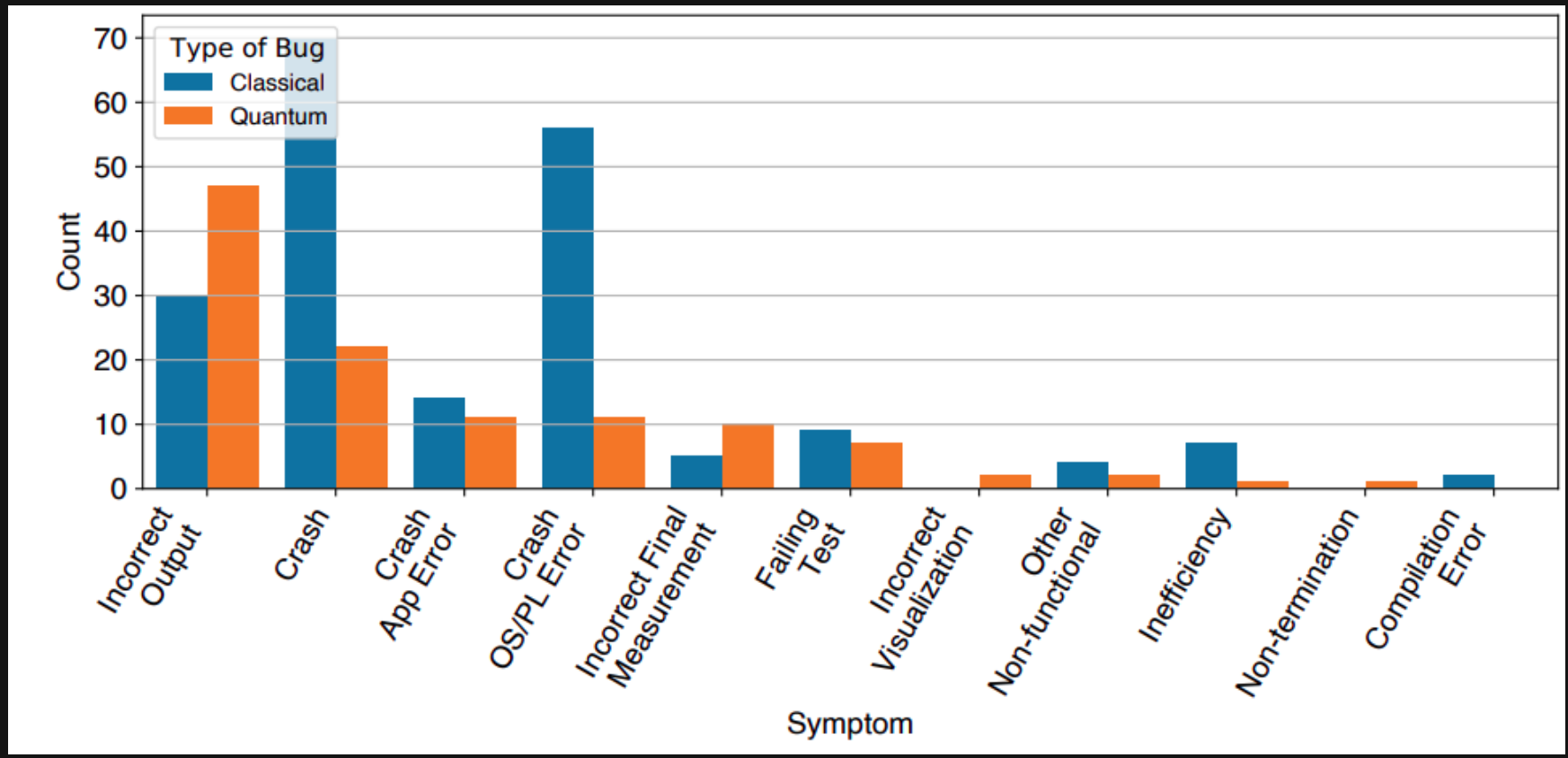
RQ3: How Do Bugs Manifest?



Most common: Crashes and incorrect output

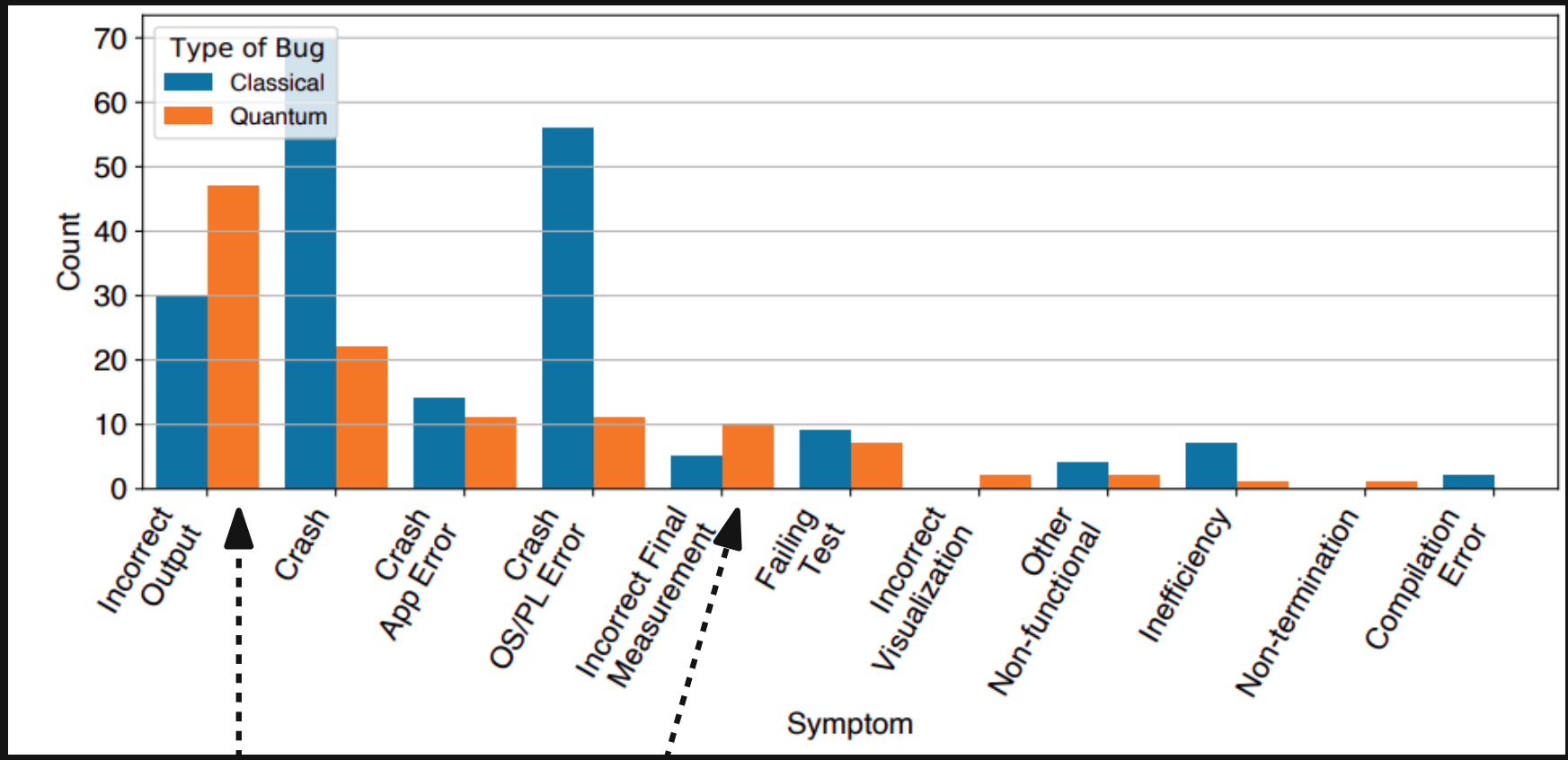
RQ3: How Do Bugs Manifest? (2)

Symptoms vs. kind of bug:



RQ3: How Do Bugs Manifest? (2)

Symptoms vs. kind of bug:



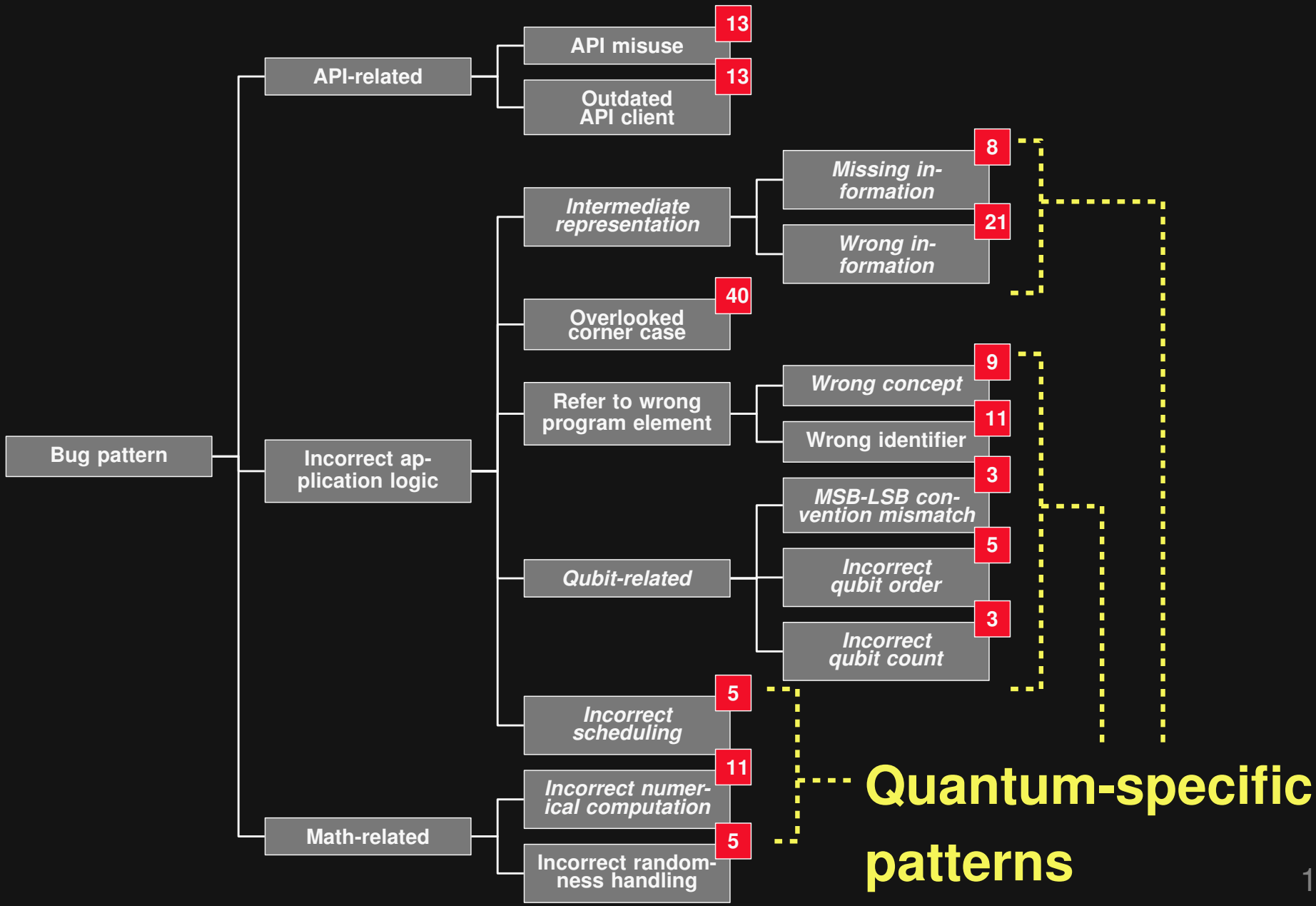
Many quantum-specific bugs: Difficult to detect

RQ4: Recurring Bug Patterns

- **Manually annotated** based on code change, commit message, issue and/or pull request
- **One bug** may fit **multiple patterns**

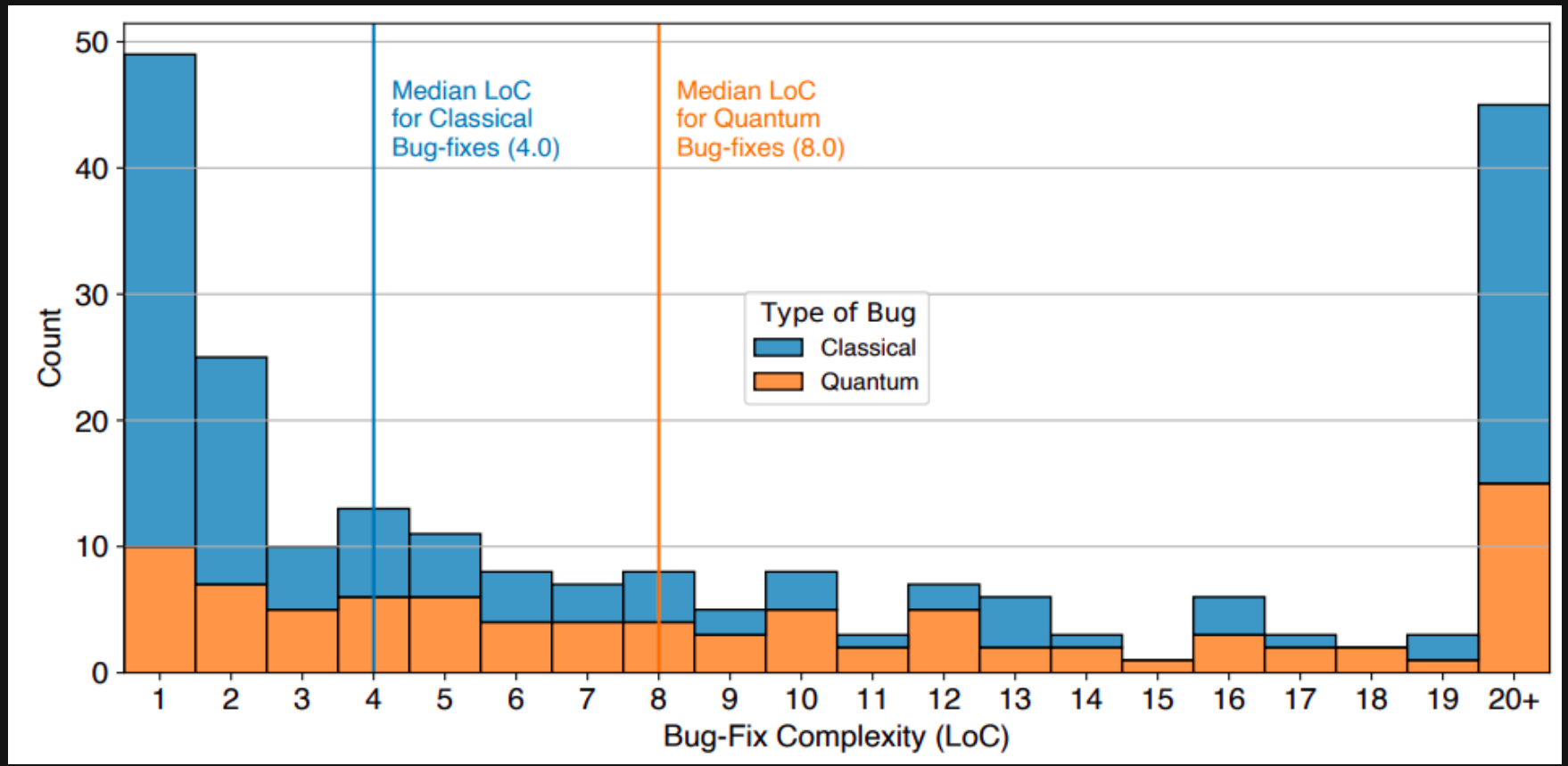
RQ4: Recurring Bug Patterns (2)

RQ4: Recurring Bug Patterns (2)



RQ5: Complexity of Bug Fixes

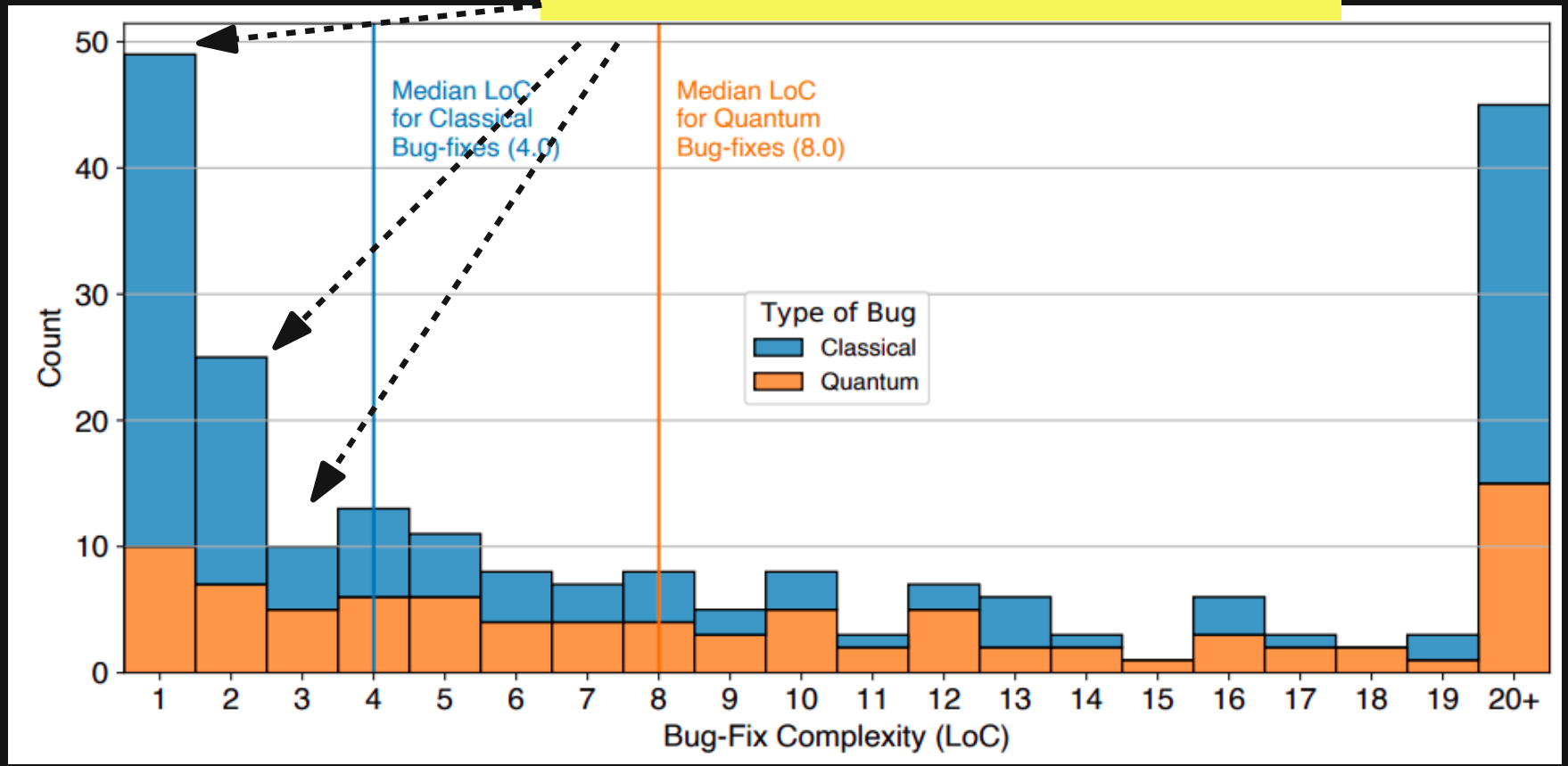
How complex are the code changes that fix bugs?



RQ5: Complexity of Bug Fixes

How complex are the code changes that fix bugs?

Possibly in reach for today's repair techniques



Implications

39.9% quantum-specific	⇢	Need new bug detection techniques
Many bugs in “inner” components	⇢	Can find bugs on simulators (no need for quantum hardware)
Crash is most common symptom	⇢	“Easy” oracle for testing
New, quantum-specific bug patterns	⇢	Opportunities for bug detectors
Many few-line fixes	⇢	Opportunities for program repair

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**

MorphQ

Automated, metamorphic testing of quantum computing platforms *

- Generator of quantum programs
- Ten metamorphic transformations

* Current target platform: Qiskit

MorphQ

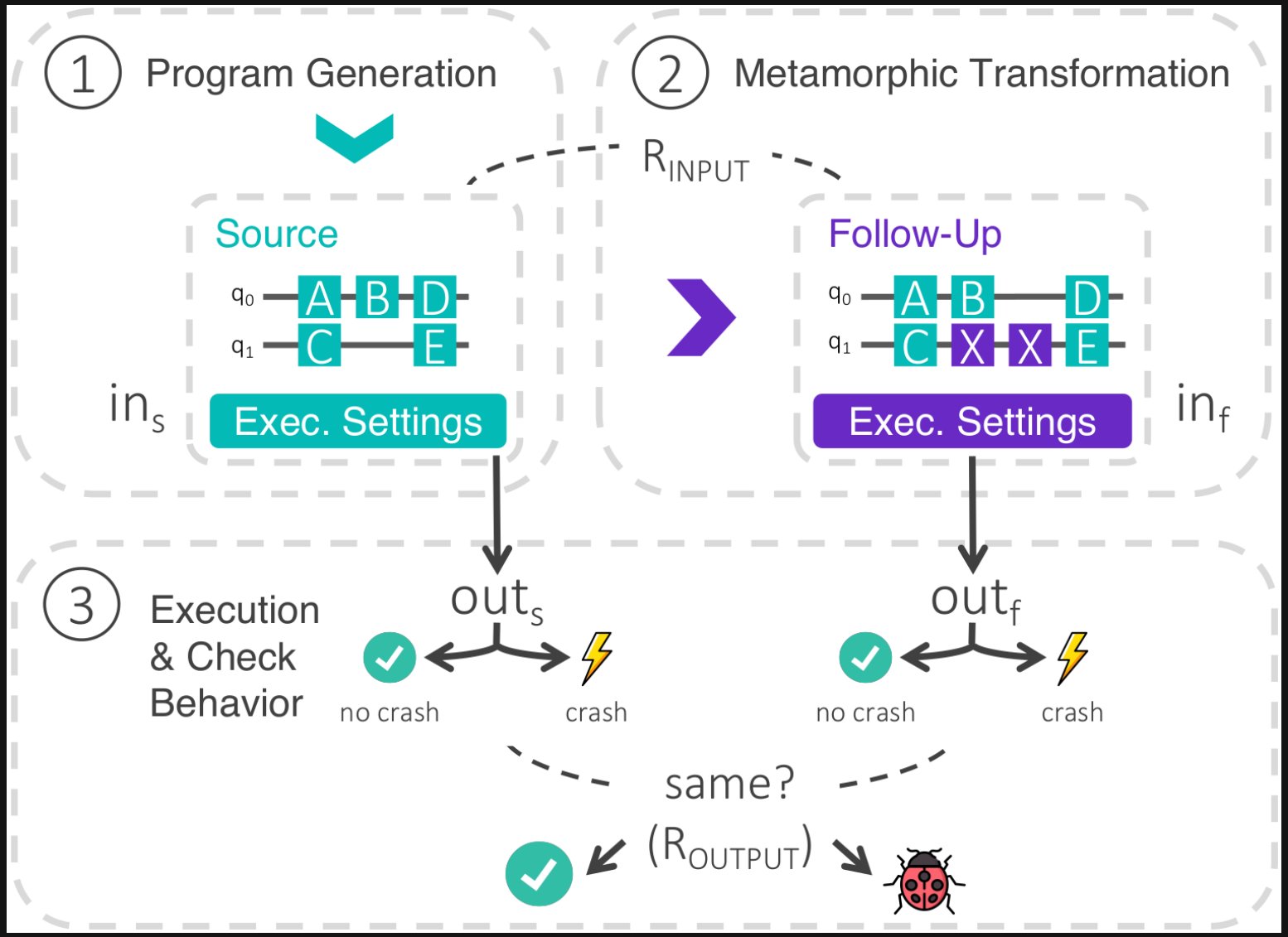
Automated, metamorphic testing of quantum computing platforms *

- 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



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

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)
```

Randomly picked from pre-defined sets

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

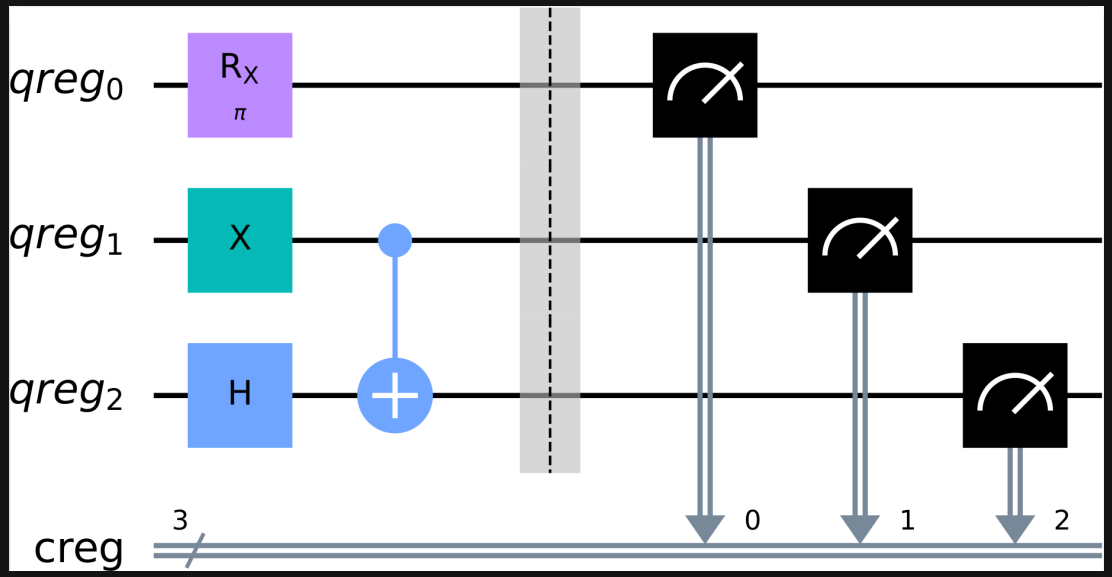
Metamorphic Transformations

Category	Name	Relationship
Circuit transf.	Change of qubit order	Equivalence modulo mapping of measurements
	Inject null-effect operation	Equivalence
	Add quantum register	Equivalence
	Inject parameters	Equivalence
	Partitioned execution	Cartesian product of output distributions
Repres. transf.	Roundtrip conversion via QASM	Equivalence
Execution transf.	Change of coupling map	Equivalence
	Change of gate set	Equivalence
	Change of optimiz. level	Equivalence
	Change of backend	Equivalence

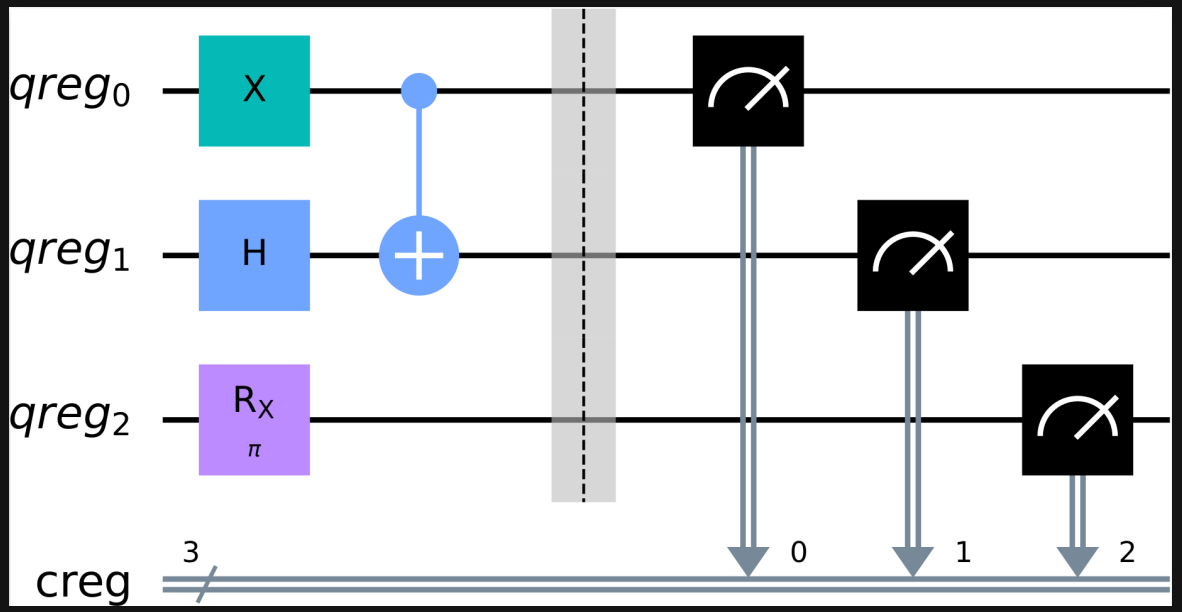
Metamorphic Transformations

Category	Name	Relationship
Circuit transf.	Change of qubit order	Equivalence modulo mapping of measurements
	Inject null-effect operation	Equivalence
	Add quantum register	Equivalence
	Inject parameters	Equivalence
	Partitioned execution	Cartesian product of output distributions
Repres. transf.	Roundtrip conversion via QASM	Equivalence
Execution transf.	Change of coupling map	Equivalence
	Change of gate set	Equivalence
	Change of optimiz. level	Equivalence
	Change of backend	Equivalence

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

```
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

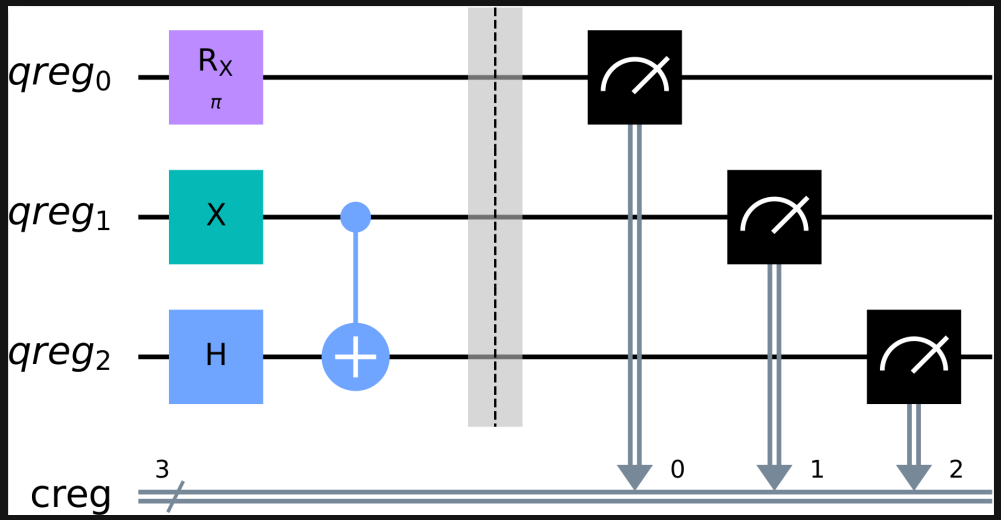
Exploit reversibility of quantum computations

Randomly generated new subcircuit

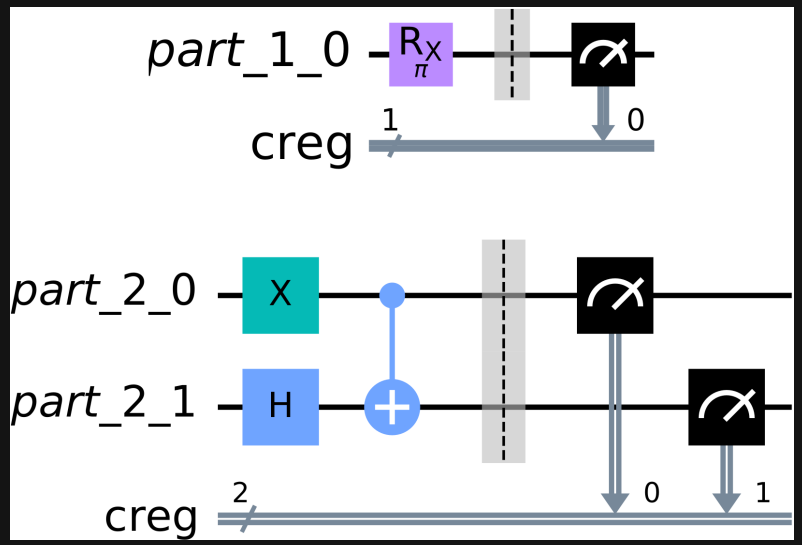
```
subcirc = QuantumCircuit(qr, cr, name='subcirc')
subcirc.append(RXGate(6.12), qargs=[qr[0]], cargs=[])
# ... sequence of additional gates
qc.append(subcirc, qargs=qr, cargs=cr)
qc.append(subcirc.inverse(), qargs=qr, cargs=cr)
```

Insert both subcircuit and its inverse

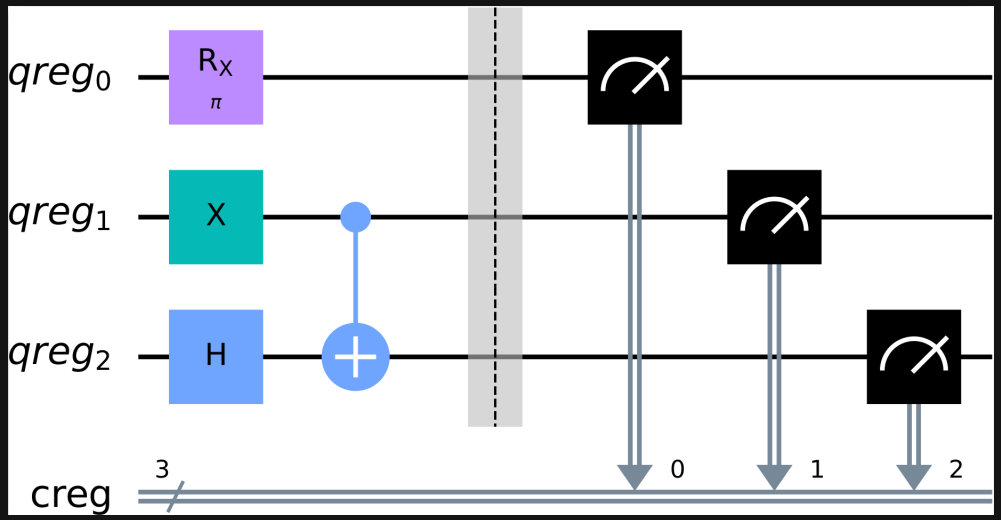
Partitioned Execution



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

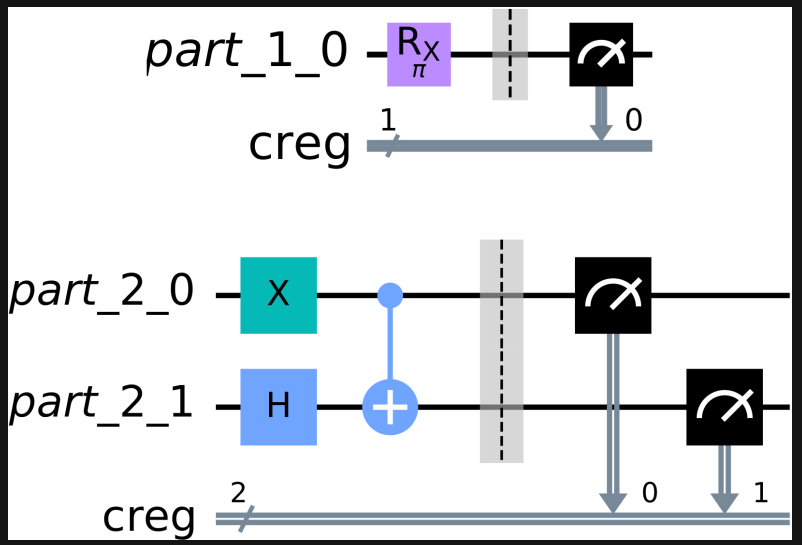


Partitioned Execution



Combine output distributions via Cartesian product

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



Roundtrip via QASM

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

```
qc = qc.from_qasm_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)

	Warnings	
	No.	%
Tested program pairs	8,360	100.0
↪ Crashes in source program	0	0.0
↪ Crashes in follow-up program	1,943	23.2
↪ Successful executions	6,417	76.8
↪ Distribution differences	56	0.7

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

```
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

```
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 ≥ 11 qubits

```
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

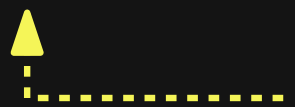
QasmError: 'subcirc' uses 4 qubits but is declared for 2 qubits

```
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

MorphQ: Metamorphic Testing of Quantum Computing Platforms (arXiv)