QChecker：通过静态分析检测量子程序中的错误

Pengzhan Zhao、Xiongfei Wu、Zhuo Li、Jianjun Zhao*九州大学
研究生院与信息科学与电气工程学院，日本

arXiv:2304.04387v1 [cs.SE] 10 Apr 2023

摘要—**静态分析是在不执行软件的情况下分析软件代码的过程。它有助于发现软件中的错误和潜在问题，而这些问题可能只有在运行时才会出现。尽管人们已经为经典软件开发了许多静态分析工具，但由于量子程序的特性，这些现有工具并不适用于分析量子程序。本文介绍了 QChecker，这是一种用于检测 Qiskit 量子程序中错误的静态分析工具。QChecker 由两个主要模块组成：一个基于抽象语法树（AST）提取程序信息的模块，以及一个基于模式检测错误的模块。我们使用 Bugs4Q 基准测试评估了 QChecker 的性能。评估结果表明，QChecker 能够有效检测量子程序中的各种错误。**

索引词—**量子编程、静态分析、软件测试、程序调试、Qiskit**

一、引言

由于相较于经典计算具有独特且极具前景的优势，量子计算已应用于许多前沿领域，例如量子机器学习 [4, 9], 大数据分析 [24], 和分子模拟 [14] 。量子编程是设计和构建可执行的量子程序，以实现特定的计算结果。随着量子程序复杂度的快速增长，减轻编写此类量子程序所需的工作量至关重要。量子程序员可以使用多种量子编程语言，例如 Qiskit [2],Sliq [5], Cirq [12], Quipper [13], 和Q# [27], ，从而使研究人员和开发者能够快速实现并试验各种量子计算技术。

鉴于量子编程的重要性及其广泛应用，确保量子程序的正确性对于量子软件开发至关重要。然而，近期的实证研究 [7, 23, 35, 36] 表明，当前的量子程序开发过程仍然容易出错。尽管对量子程序进行调试和测试已受到广泛关注 [3, 11, 15, 17, 19, 20], ，现有的调试和测试技术通常需要动态执行底层量子程序。考虑到当前大多数量子程序都在云端提供的量子计算机和模拟器上执行，现有的调试和测试技术可能繁琐且成本高昂。

在经典软件开发实践中，静态分析技术凭借其速度快、成本低的优势，被广泛用于检测各种类型的经典程序中的错误 [10, 16, 25, 26, 30]。然而，通过静态分析检测量子程序中的错误可能颇具挑战性。由于量子计算逻辑以量子电路的形式表达，且量子寄存器的状态是以概率方式测量的，为经典程序设计的静态分析工具难以检测量子程序中的错误。

为弥合这一差距，我们提出了QChecker，这是一种用于检测量子程序中缺陷的静态分析工具，尤其针对Qiskit。该方法首先从以往研究中的真实量子程序缺陷 [35, 36]中提炼出一组常见缺陷模式，然后构建八个检测器，用于检测量子程序中的这些缺陷模式，从而应对上述挑战。整个过程并非易事，因为提炼出的缺陷模式必须仔细考虑量子计算的领域特定约束，才能确保准确且有效。

我们在 Bugs4Q [36], 上评估 QChecker。Bugs4Q [36], 是一个由 42 个真实世界中的含缺陷量子程序组成的真实基准测试集。实验结果表明，QChecker 能够高效地检测量子程序中的缺陷。此外，我们还讨论了 QChecker 在其他基于 Python 的量子编程语言中的可扩展性。

总而言之，本文作出了以下贡献：

- 我们介绍了 Qiskit 中首个专用于量子程序的错误检测工具。利用静态分析技术，QChecker 可以生成诊断消息，帮助开发者快速定位其程序中的潜在错误。
- 我们实现了QChecker，并在真实世界的 Bugs4Q基准测试中评估了其有效性和性能。结果表明，QChecker 能够有效检测量子程序中的各种类型错误。

本文其余部分的组织如下。第II节介绍量子编程的一些基础知识。第III节描述了我们针对量子程序静态分析的 QChecker 方法。第 IV 节介绍 QChecker 在 Bugs4Q 上的性能。第 V 节回顾了我们的有效性威胁。第 VI 节讨论相关工作，最后第 VII 节对本文进行总结。

II. 背景

我们简要介绍 Qiskit 编程的背景信息以及量子比特的基本概念。

*Jianjun Zhao(zhao@ait.kyushu-u.ac.jp) 是通讯作者。

```
模拟器 = Aer.get_后端("qasm_simulator")

量子寄存器 = QuantumRegister(3)
经典寄存器 = ClassicalRegister(3)
电路 = 量子电路(qreg, creg)

电路.h(0)
电路.h(2)
电路.cx(0, 1)
电路.measure([0,1,2], [0,1,2])
作业 = execute(电路, 模拟器, shots=1000)
结果 = 作业.result()
计数 = 结果.get_counts(电路)
print(计数)
```

图 1。Qiskit 中的一个简单量子程序

A. Qiskit

Qiskit [2] 是最广泛使用的量子计算开源框架之一，使我们能够为量子计算机创建算法。作为一个Python 软件包，它提供了用于创建和操作量子程序，以及在原型量子设备和模拟器上运行量子程序的工具，还可以使用内置模块进行噪声表征和电路优化，以减少噪声的影响。Qiskit 还提供了用于机器学习、优化和化学领域的量子算法库。

在 Qiskit 中，程序由一个量子对象数据结构定义，其中包含配置信息和实验序列。该对象可用于获取状态信息并检索结果 [22]。图 1 展示了一个简单的 Qiskit 程序，说明了量子程序的完整工作流程。函数`Aer.get_后端('qasm_simulator')`会返回给定后端名称（`qasm_simulator`）对应的后端对象。后端 类是模拟器的接口，而此类在`Aer` 中的实际名称是`AerProvider`。实验设计完成后，指令会通过`execute` 方法运行。模拟的`shots` ，即运行电路的次数，被设置为 1000，而默认值为 1024。在输出测量结果时，使用`job.result()` 方法来检索测量结果。我们可以通过`get_counts(circuit)`方法访问计数结果，该方法给出实验的汇总结果。

B. 量子比特的基本性质

在本小节中，我们以 Qiskit 为例，解释量子比特（简称量子比特 ）的特征以及完整量子程序的必要执行过程。

量子计算中的基本信息单位是量子比特。如图1所示，`qreg = QuantumRegister(3)`表示分配一个包含三个量子比特的量子寄存器，并且每个量子比特的值默认为 $|0\rangle$ 。因此，这三个量子比特的初始值为 $|000\rangle$。接下来，让第一个和第三个量子比特通过H（Hadamard）门，如`circuit.h(0)`和`circuit.h(2)`所示。这样就实现了量子比特的独特属性——叠加态，它表示该量子比特包含 $|0\rangle$ 和 $|1\rangle$的状态。还有一种只有多个量子比特才能实现的量子比特属性量子纠缠 。示例程序中的代码是`circuit.cx(0,1)`。也就是说，第一个量子比特通过 CNOT（受控非）门操作与第二个量子比特发生纠缠。我们测量第一个量子比特，其输出为`0` 的概率是 50%，为`1` 的概率是50%。之后，测量第二个量子比特的结果与第一次测量结果完全相同，概率为100%。由于第三个量子比特与前两个量子比特无关，最后一个量子比特的测量结果仍然是`0` 的概率为 50%，是`1` 的概率为 50%。图 1 所示的量子比特测量语句是`circuit.measure([0,1,2],[0,1,2])`。测量可能导致量子叠加态坍缩为经典态。量子测量有许多种类，这里使用单个量子比特的投影测量。也就是说，每个量子比特都被投影到由基向量 $|0\rangle$ 或 $|1\rangle$组成的状态空间中。在此程序中，最终输出是一个三位数组。

III. QCHECKER 工具

本节介绍 QChecker 的构建过程，该工具基于 Python 开发。如图 2 所示，QChecker首先基于量子程序的 AST，对其进行全面的信息提取。相应的操作位于模块`Ast_Operator`中。所提取的信息主要包括变量赋值操作和函数调用，这些信息随后将存储在`QP_Attribute` 和`QP_Operation` 中。然后，QChecker 将提取的信息传递给缺陷检测器。缺陷检测器能够检测出各种缺陷模式，如表 I 所示。最后，QChecker 生成错误检测报告，其中包括存在错误的程序、行号和错误描述。

A. 信息提取

之前的静态分析工具启发了我们（例如PyLint [28]）：使用 AST 提取程序信息是有效且高效的。然而，不同于经典静态分析工具，QChecker 中的`AST_操作符` 能够提取量子程序特有的语义和功能信息。以图 1 所示的程序为例，我们对每个量子程序文件进行结构化解析，即生成 AST。我们采用两个名为`QP_属性` 和`QP_操作` 的模块，分别存储所有变量和函数调用的 AST 信息。此外，QChecker还支持处理复杂的语法和数据结构，例如字典、列表、函数定义、循环和条件分支。这样设计的目的是，基于结构化 AST 的信息提取能够帮助 QChecker跟踪每个变量与函数调用之间的关系。例如，一个变量可能被多次修改，或者在函数内部作为参数传递时其名称可能发生变化。尽管如此，我们仍然可以追溯程序中变量的初始值。我们分别在图 3 和图 4中绘制了`QP_属性`和`QP_操作` 的实例。

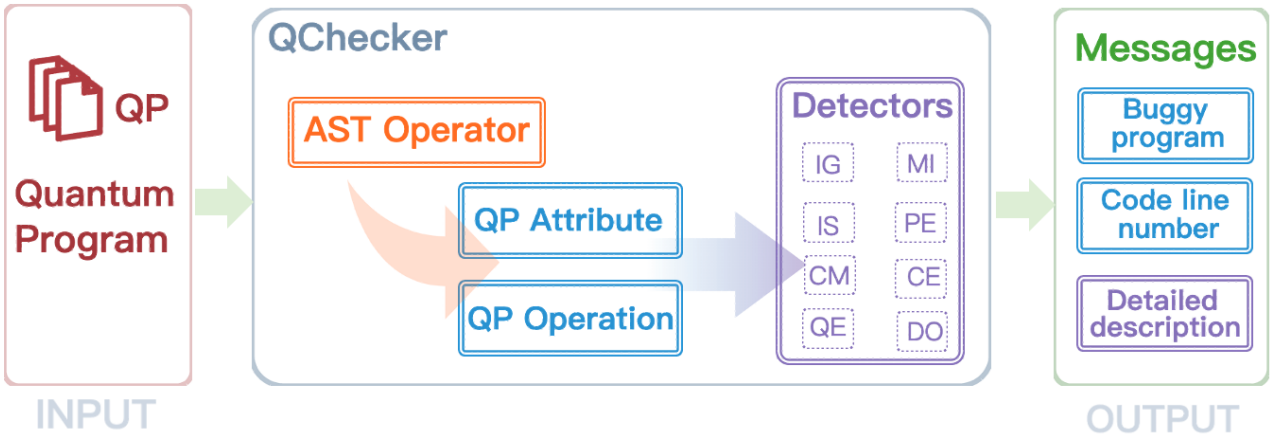


图 2. QChecker 的结构。

('模拟器','**Aer.get_后端**("qasm_simulator")') ('量子寄存器', '**量子寄存器**(3)') ('经典寄存器', '**经典寄存器**(3)') ('电路', '**量子电路**(qreg,creg)') ('作业', '**execute**(circuit,simulator,shots=1000)') ('结果', 'job.**result**()') ('计数', 'result.**get_counts**(circuit)')

图 3. 从QP_属性中提取的程序信息。

'**Aer.get_后端**("qasm_simulator")' '**QuantumRegister**(3)' '**ClassicalRegister**(3)' '**量子电路**(qreg,creg)' 'circuit.**h**(0)' 'circuit.**h**(2)' 'circuit.**cx**(0,1)' 'circuit.**measure**([0,1,2],[0,1,2])' '**execute**(circuit,simulator,shots=1000)' 'job.**result**()' 'result.**get_counts**(circuit)' '**print**(counts)'

图 4. 通过QP_Operation提取的程序信息。

• **QP 属性**：变量的 AST 结构包含变量名、变量值（可以来自常量、另一个变量或函数计算结果）、类型和位置（代码行）。如图 3 所示，QP_属性 模块采用键值对的方式设计。键是变量名，可用作变量值的索引。

• **QP 操作**：函数调用的 AST 结构包含其参数列表、每个参数的类型和值、位置以及其他信息。如图 4 所示，QP_操作 模块是一个列表，包含量子程序文件中的所有函数调用。具体而言，每个函数调用还可进一步拆分为函数调用名称、参数和值。这些信息存储在 QChecker 提供的一个更全面的表格中，该表以QP_操作 中的函数调用字符串作为索引。

这两个模块包含量子程序的所有信息，使后续的错误检测更加直接。此外，用户可以通过我们发布的API，基于 QChecker 直接获取上述信息。值得一提的是，包含基本语法错误（例如 Python缩进错误、无法识别的运算符、未定义的变量和函数等）的程序将不会被QChecker 处理。相反，系统会将它们提示为语法错误，因此会将其排除在静态检查之外。

B. 错误模式检测器

缺陷模式是指反复被证明会导致失败的错误代码习惯用法或不良编码实践，通常由误解编程语言的特性、使用错误的设计模式，或具有共同行为的简单错误造成。以往研究已针对 Qiskit 编程语言 [35, 36] 识别出一些缺陷模式。在本研究中，我们对这些缺陷模式进行了改进，并构建了八个检测器来检测它们。表 I 展示了检测器的名称及缺陷模式的描述。下面简要介绍每个检测器。

*1)*量子门的不正确使用（*IG*）：该检测器主要检查量子门是否被正确调用。它确- 定某个量子门是否被 Qiskit识别，以及该量子门是否已被定义。此外，还会检查自定义量子门和三量子比特量子门是否符合规范。

*2)*测量相关问题（*MI*）：错误的测量不仅指不当使用测量操作导致的错误，也包括测量之后执行了错误的操作。如图5 所示，用户希望实现一个量子隐形传态程序。被测量的量子比特被用作控制量子比特，与其他量子比特进行纠缠。该检测器主要在调用 测量方法之后发挥作用。它会遍历后续的

表 I 每个检测器负责检测的缺陷模式。

检测器名称	错误模式描述
IG	- 量子门不属于后端的基门。 - 处理自定义多量子比特门。 - 未定义随机门。
MI	- 忽略测量的影响。
IS	- 量子比特数量多于所定义的寄存器数量。 - 量子比特数量不足。 - 经典寄存器长度不足。
PE	- 指令不属于基门。 - 量子门参数不正确。 - 使用经典比特进行量子纠缠。 - 在一次操作中使用了同一个物理量子比特。 - 未为耦合映射提供列表。 _
CM	- 无法识别的参数。 - 量子电路交互错误。 - 创建冗余的经典寄存器。 - 使用了错误的命令。
CE	- 对象调用错误。 - 导入错误。 - 后端错误。 - 翻译错误。
QE	- qasm str() 方法中的问题。
DO	- 方法已弃用。

qc = **量子电路**(3, 3) qc.**x**(0) qc.**h**(1) qc.**cx**(1, 2)
qc.**cx**(0, 1) qc.**h**(0) qc.**测量**(0, 0) qc.**测量**(1, 1) qc.**cx**(1,
2) <- 有问题的操作qc.**cz**(0, 2) <- 有问题的操作

图 5. 错误测量示例

测量语句之后的操作，并确定被测量的量子比特是否作为双量子比特门操作中的控制量子比特。

3）不正确的初始状态（*IS*）：此检测器并不只是检查`QuantumRegister` 和`ClassicalRegister` 的定义是否符合规范。它会确定初始化是否满足整个量子程序对量子比特的操作。有时，Qiskit 程序会在模拟量子程序时限制所使用的量子比特数量。例如，经过我们的验证，使用`Aer.get_后端('qasm_simulator')`作为后端时，测量操作支持的量子比特数少于 30 个，而`BasicAer.get_后端('qasm_simulator')`支-

量子寄存器 = qk。**量子寄存器**(7)

布局 = {量子寄存器[0]: 12, <- 有问题的操作
量子寄存器[1]: 11,
量子寄存器[2]: 13,
qreg[3]: 17,
量子寄存器[4]: 14,
qreg[5]: 12, <- 有问题的操作
qreg[6]: 6}

图 6。参数错误示例

相位 = **参数**('相位')

使用pulse.build(FakeAlmaden())**作为**phase_test_sched:

pulse.**shiftphase**(<- 无法识别
phase, pulse.**驱动_通道**(0))

phase_test_sched.**指令**

图 7. 命令误用示例。

支持少于 24个量子比特。在这种情况下，检测器首先检查用户选择的后端，并识别已初始化的量子比特是否超出限制。当已初始化的量子比特数量设置为`n` 时，检查器将跟踪程序中调用的量子比特数量。

4）参数错误 *(PE)*：调用量子门后，此检测器负责判断量子门中的参数是否正确，包括多量子比特门中不存在的参数，以及数值类型的错误使用。不过，有些缺陷模式不容易发现。从图 6 可以看出，用户希望将寄存器中的量子比特分配给物理量子比特，两个量子寄存器`[0]`和量子寄存器`[5]` 都被分配给物理量子比特`12`。因此，检测器会遍历参数值，并检查物理量子比特占用是否重复。

5）命令误用 *(CM)*：此检测器可以检测命令的错误或不当使用。有时，由于方法名拼写错误，参数无法被识别。另一方面，有些方法无法识别参数并会引发错误。如图 7 所示，模块`qiskit.phase`中不存在属性`pulse.shiftphase()` 。Qiskit 中有些命令难以检测。例如，当量子电路超过两个，而用户想将一个电路嵌套到其他电路中时：1）命令`to_gate()`可用于将电路转换为嵌入其他电路中的量子门组合。2）命令`decompose()` 可用于自动分解电路以执行嵌入式操作。

*6)*调用错误（*CE*）：此检测器负责调用错误，包括 Python 包调用、后端模拟器调用和转换器调用。此外，该检测器还可以检查参数声明方面的问题。如图 8 所示，此错误并不是对`PauliMeasurementBasis()`的重复调用。运行代码后，我们发现存在用于无效量子比特的 *ba-*

```
circ  = 量子电路(1,1) circ.x(0)

层析 = 过程层析( 电路=circ,测量_基=泡利测量基(),测量_量子比特=None, 制备_基=泡利测量基(), <-制备_量子比特=None, 基_索引=None, 量子比特=None)
```

图8. 调用错误示例

sis。对`PauliMeasurementBasis()` 的调用对于制备_基无效。这类错误很难由 QChecker 检测到，因为检测器目前只能判断一种情况。

*7) QASM*错误（*QE*）：此检测器用于检测将`qasm`_模拟器用作后端时，或使用 Qiskit 编程语言构建*qasm*程序时出现的问题。

8) 已弃用操作（*DO*）：此检测器用于判断是否调用了已弃用的方法；当旧操作或变量类型因版本更新而被弃用时，该检测器便会发挥作用。

C. 错误检测

表 I 中所展示的缺陷模式代表了量子程序中的常见错误。除了语法错误之外，其中还包含一些量子相关操作中的错误逻辑。基于程序信息提取模块（`QP`_属性和 `QP`_操作），以及检测器，我们可以对量子程序文件执行全面的静态分析。为了实现全面且有用的错误检测并有利于调试过程，我们报告存在错误的程序的详细信息（代码行、内容等）、错误类型（模式）以及具体描述。我们希望这些信息能够帮助用户提高量子程序的质量。

IV. 评估

本节介绍 QChecker的实证性能。我们在 Bugs4Q [36]，其中包含 42 个基于Qiskit 的真实世界量子程序错误案例上对QChecker 进行了评估。实验在一台配备 Inteli9-10940X CPU、128GRAM、运行 Ubuntu 20.04 并安装 Python 3.10 的服务器上进行。

A. 指标

我们采用精确率、召回率和*F1*分数 来评估 QChecker 的性能。具体而言，对于 Bugs4Q 中的每个漏洞*b* ，我们将 Bugs4Q 中的漏洞类型作为真实值，并将 QChecker 应用于其源量子程序。如果 QChecker 的检测结果与 Bugs4Q 中对应的漏洞类型相匹配，则称该*b* 为真正例 （TP）。否则，该*b*为假正例（ FP ）。假负例 （ FN ）是指真实值中的漏洞，但 QChecker 无法检测到该漏洞。精确率 计算为 $TP/(TP+FP)$，召回率 计算为 $TP/(TP+FN)$，而*F1*分数 计算为 $2\times Precision\times Recall/(Precision+Recall)$。

B. 在真实世界*Qiskit*程序上的性能

首先，QChecker 检测了 42 个 Qiskit 程序，并发现了 24 个漏洞。图9展示了将 QChecker 应用于 Bugs4Q 后得到的实证结果。检测器`PE` 发现了 10 个漏洞，而`CE` 和`IS` 分别发现了 7 个和 2 个漏洞。其他检测器各发现了 1 个漏洞。众所周知，在 Bugs4Q 的 42 个漏洞中，有 22 个是输出错误，即程序的输出与用户预期不符。这种情况使 QChecker 无法检测到这些漏洞。因此，我们认为，QChecker 在 Bugs4Q 中发现的其余漏洞数量符合预期。结合表 I 和图9中的结果，我们分析并总结出了关于每个检测器性能的两点：

- 检测器能够覆盖的案例越多，能够发现的缺陷就越多。由于我们根据缺陷模式的相关性构建检测器，因此每个检测器覆盖的案例数量可能有所不同。检测器 `PE` 和 `CE`能够检测比其他检测器更多的案例。事实上，这两个检测器表现更好。
- 检测器的性能还取决于程序员所犯错误的类型。鉴于 Bugs4Q 中的错误数量有限，我们认为某些检测器表现不佳，是因为它们覆盖的情况很少发生。例如，检测器`IG`、`CM`和`IS` 可以检测多种情况，但结果仍不尽如人意。

此外，仅能解决一种情况的检测器并不意味着性能不佳。相反，随着功能不断增强，这些专用检测器将发挥其潜力，更好地检测错误。

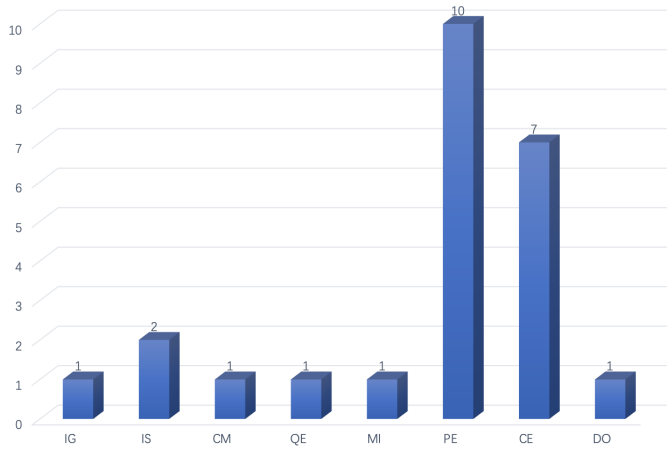


图9。每个检测器发现的错误分布。

Bugs4Q 提供的42 个 Qiskit量子程序上 QChecker的效率如表II 所示。如 [21],所示，在模拟器上执- 行量子程序很容易消耗超过 10^3 毫秒。因此，QChecker 展现出了很高的效率，在单个量子程序上完成检测的平均耗时仅为 48.2 毫秒。为了更好地体现这一点，我们调查了其中的 42 个量子程序

表 II QChecker 在 Qiskit 程序上的性能。

性能	精确率	召回率	*F1*分数	平均时间
QChecker	0.625	0.882	0.731	48.2 毫秒

导入 cirq qubit = cirq.**NamedQubit**("myqubit") circuit = cirq.**Circuit**(cirq.H(qubit))**对于**i **在** range(10): result2 = cirq.**measure**(qubit, key='myqubit')**打印**(result2)**打印**(circuit) result = cirq.**Simulator()**.simulate(circuit)**打印**(result2)

图 10. 一个 Cirq 程序示例。

Bugs4Q。对于这些程序，平均执行时间为 2.14 秒，而每个程序的平均代码量为31 行代码。从第 II节中，我们已经知道，获取一个量子比特的状态需要对输出进行大量迭代，以获得其概率分布，我们认为这是一项程序执行的主要成本。

总而言之，QChecker 在执行效率和相对有效性方面具有优势，而缺点是QChecker 无法避免假正例问题。随着未来量子程序中量子比特数量的增加，我们认为，通过静态分析在程序执行前发现错误是必要且有效的。

C. QChecker 的可扩展性

接下来我们讨论 QChecker的可扩展性。图 10 和图 11 分别展示了一个 Cirq 程序以及QChecker 对应的检测结果。仔细检查这些结果后，我们发现，QChecker 的信息提取部分仍可在其他基于 Python 的量子语言（例如 Cirq）上运行。这些量子语言之间的语法差异可能导致为 Qiskit 设计的检测器无法正常工作，从而无法保证其性能。

总之，实验结果表明，QChecker能够成功检测现实世界中的Qiskit 量子程序中的错误，展示了将静态分析应用于量子程序的有效性。此外，中间结果表明，QChecker 能够从底层Cirq程序中正确提取`QP_`属性 和`QP_`操作信息，表明QChecker 可以轻松扩展到常见的基于 Python 的量子编程语言。

V. 对有效性的威胁

A. 外部威胁

即使对于使用最广泛的 Qiskit 量子编程语言来说，存在错误的程序仍然不够多。此外，现有的量子程序通常运行在模拟器上，而非实际的量子计算机上，这导致了The current paragraph ends at “the”, thus Chinese “这导致了” corresponds to cutoff. Need preserve weird spaces? They had usedQiskit quantumprogramming, likely concatenation artifacts; translation natural. But maybe "使用最广泛的 Qiskit 量子编程语言" okay. Ensure glossary "buggy programs" exact target 存在错误的程序, used. "quantum programs" 量子程序. Yes. Could translate "there are still not enough buggy programs" as "仍然缺乏足够多的存在错误的程序" more natural than currently "存在错误的程序仍然不够多". Either fine. Output.

('qubit', 'cirq.**NamedQubit**("myqubit")') ('circuit', 'cirq.**Circuit**(cirq.H(qubit))') ('result', 'cirq.**Simulator**().simulate(circuit)') ('result2', 'cirq.**measure**(qubit,key="myqubit")')
==========' cirq.**NamedQubit**("myqubit")' 'cirq.**Circuit**(cirq.H(qubit))' 'range(10)' '**print**(circuit)' 'cirq.**Simulator()**.simulate(circuit)' '**print**("result:")' '**print**(result2)' 'cirq.**H**(qubit)' 'cirq.**measure**(qubit,key="myqubit")', '**print**(result2)', 'cirq.**Simulator()**'。

图 11. QChecker 检测到的一个 Cirq 程序

`eng` =**MainEngine**`()`
量子比特 = eng.allocate_qureg(3)**H** | 量子比特`[0]`**CX** | (量子比特`[0]`, 量子比特`[2]`) eng.flush() 振幅 = np.array(eng.backend.cheat()`[1]`) 振幅 = np.abs(amplitudes)**全部(测量)** |量子比特

图 12。ProjectQ 程序示例。

当前量子程序规模较小。因此，缺陷模式的数量可能会对有效性构成威胁。我们投入了大量精力，从量子程序中收集缺陷，并尽可能从这些收集到的缺陷中提取更多缺陷模式。然而，由于当前量子程序开发与应用规模的限制，我们无法在 QChecker 中纳入更多缺陷模式。因此，我们将继续收集量子程序及其缺陷，丰富 QChecker 的检测能力，并持续更新该工具。

B. 内部威胁

QChecker 专为 Qiskit 设计，也可以通过少量修改扩展到其他基于Python 的量子语言（例如Cirq）。然而，它也存在局限性。例如，ProjectQ 重载了 | 运算符，这将导致信息提取器无法正常工作。如图 12 和图 13所示，QChecker 无法从底层的ProjectQ 程序中提取信息，例如 `H` 门和`CX`门。这些局限性将通过扩展 QChecker 得到解决。

C. 可验证性

这一威胁涉及复现本研究的可能性。我们提供了帮助研究人员复现这项工作的所有必要细节。复现软件包已公开发布于 https://github.com/Z-928/QChecker。

VI. 相关工作

大量基于静态分析的错误检测技术 [10, 16, 25, 26, 30] 已在经典

```
('引擎', 'MainEngine()')
('量子比特',    'eng.allocate_qureg(3)')
('振幅',                'np.array([1])')
('振幅',                'np.abs(amplitudes)')
==========================================

'MainEngine()'
'eng.allocate_量子寄存器(3)'
'eng.flush()'
'np.array([1])'
'np.abs(amplitudes)'
'全部（测量）'
'eng.backend.cheat()']
```

图 13. QChecker 检测到的一个 ProjectQ 程序。

软件开发中已开发出大量基于静态分析的错误检测技术 {v1}。然而，由于量子程序与经典程序之间存在本质差异，经典程序的静态分析技术难以直接应用于量子程序。

目前，针对量子程序的静态分析技术已相继出现。Yu 和 Palsberg [34] 提出了一种用于量子程序的抽象解释技术，并利用该技术检测断言，以发现程序中的错误。Xia 和 Zhao [32] 提出了一种实用的静态纠缠分析技术，可准确分析 Q# 量子程序中模块内部及模块之间的纠缠信息，从而帮助发现程序中与纠缠相关的错误。ScaffCC [18] 是一个面向量子编程语言 Scaffold [1], 的可扩展编译器框架，也支持纠缠分析。ScaffCC 探索了数据流分析技术，通过对 QASM-HL 程序（ScaffCC 的一种中间表示）的输出进行注释，自动跟踪代码中的纠缠情况，以标记可能发生纠缠的量子比特。该分析采用保守策略，因为它假定两个量子比特一旦相互作用，就可能已经彼此纠缠。与这些分析方法不同，QChecker 旨在通过静态分析，基于缺陷模式发现量子程序中的缺陷。

研究人员还将霍尔逻辑扩展到了量子领域，以支持量子程- 序的形式化验证 [6, 8, 33, 29, 37]。其中，Li等人 [37] 提出了应用量子霍尔逻辑（aQHL），这是量子霍尔逻辑（QHL） [33],的简化版本，尤其强调支持量子程序的调试和测试。aQHL通过将QHL 绑定到一类特定的前置条件和后置条件（断言），即投影，简化了 QHL，从而降低量子程序验证的复杂性，并提供了一种便于调试和测试量子软件的方法。然而，正如我们所知，形式化验证可能成本高昂，且难以扩展。

另一条研究路线是开发量子程序的调试 [17, 19]和测试 [3, 11, 15, 20, 31] 技术，这些技术原则上基于动态程序分析。虽然能够获得更准确的结果，但由于量子程序的特性，与静态分析技术相比，这些技术的运行成本相对较高。

VII. 结论

本文介绍了 QChecker，这是一款面向量子程序的静态分析工具，旨在有效、高效地检测量子程序中的潜在缺陷。QChecker 包含两个基于 AST 的程序信息提取模块和一套全面的缺陷检测器，能够检测各种缺陷模式。我们将 QChecker 应用于 Bugs4Q 基准测试套件，并评估了其有效性。结果表明，QChecker 能够检测量子程序中的多种缺陷。未来，我们计划扩展 QChecker，以检测更多 Qiskit 程序的缺陷模式，并支持对 Cirq 和 ProjectQ 等其他常见量子编程语言进行缺陷检测。

VIII. 致谢

本研究得到了JST SPRING（资助编号：JPMJSP2136和JPMJFS2132）的支持。

参考文献

[1] A. J. Abhari、A. Faruque、M. J. Dousti、L. Svec、O. Catu、A. Chakrabati、C.-F. Chiang、S. Vanderwilt、J. Black 和 F. Chong，《Scaffold：量子编程语言》，计算机科学系，普林斯顿大学，技术报告，2012 年。[2]G. Aleksandrowicz、T. Alexander、P. Barkoutsos、L. Bello、Y. Ben-Haim、D. Bucher、F. Cabrera-Hern´andez、J. Carballo-Franquis、A. Chen、C. Chen 等，《Qiskit：用于量子计算的开源框架》，访问日期：*3* 月，第 16 卷，2019 年。[3]S. Ali、P. Arcaini、X. Wang 和 T. Yue，《评估输入和输出覆盖准则对测试量子程序的有效性》，载于 *2021* 年第 *14* 届 *IEEE* 软件测试、验证与确认会议（*ICST*）。IEEE，2021 年，第 13–23 页。[4]J. Biamonte、P. Wittek、N. Pancotti、P. Rebentrost、N. Wiebe 和 S. Lloyd，《量子机器学习》，《自然》，第 549 卷，第 7671 期，第 195–202 页，2017 年。[5] B. Bichsel、M. Baader、T. Gehr 和 M. Vechev，《Silq：具有安全反计算和直观语义的高级量子语言》，载于 第 *41* 届 *ACM SIGPLAN* 编程语言设计与实现会议论文集，2020 年，第 286–300 页。[6] O. Brunet 和 P. Jorrand，《量子程序的动态量子逻辑》，《国际量子信息杂志》，第 2 卷，第 01 期，第 45–54 页，2004 年。[7]J. Campos 和 A. Souto，《QBugs：量子算法中可复现缺陷的集合及支持基础设施，以实现受控的量子软件测试和调试实验》，*arXiv* 预印本 *arXiv:2103.16968*，2021 年。[8] E. D'hondt 和 P. Panangaden，《量子最弱前置条件》，《计算机科学中的数学结构》，第 16 卷，第 3 期，第 429–451 页，2006 年。[9] V. Dunjko、J. M. Taylor 和 H. J. Briegel，《量子增强机器学习》，《物理评论快报》，第 117 卷，第 13 期，第 130501 页，2016 年。

[10] C. Flanagan、K. R. M. Leino、M. Lillibridge、G. Nelson、J. B. Saxe 和 R. Stata，《Java 的扩展静态检查》，载于 *ACM SIGPLAN 2002* 编程语言设计与实现会议论文集，2002 年，第 234–245 页。[11] D. Fortunato、J. Campos 和 R. Abreu，《量子程序的变异测试：以 Qiskit 为例的案例研究》，《*IEEE* 量子工程汇刊》，第 3 卷，第 1–17 页，2022 年。[12]Google AI Quantum 团队，《Cirq》，2018 年。 [在线]。可访问：https://github.com/quantumlib/Cirq[13] A. S. Green、P. L. Lumsdaine、N. J. Ross、P. Selinger 和 B. Valiron，《Quipper：可扩展的量子编程语言》，载于 第 *34* 届 *ACM SIGPLAN*编程语言设计与实现会议论文集，2013 年，第 333–342 页。[14] H. R. Grimsley、S. E. Economou、E. Barnes 和 N. J. Mayhall，《量子计算机上精确分子模拟的自适应变分算法》，《自然通讯》，第 10 卷，第 1 期，第 1–9 页，2019 年。[15] S. Honarvar、M. Mousavi 和 R. Nagarajan，《Q# 中量子程序的基于属性的测试》，载于 首届量子软件工程国际研讨会（*Q-SE 2020*），2020 年。[16] D. Hovemeyer 和 W. Pugh，《发现缺陷很容易》，《*ACM SIGPLAN* 通知》，第 39 卷，第 12 期，第 92–106 页，2004 年。[17] Y. Huang 和 M. Martonosi，《用于验证模式和发现量子程序缺陷的统计断言》，载于 第 *46* 届国际计算机体系结构研讨会论文集，2019 年，第 541–553 页。[18] A. JavadiAbhari、S. Patil、D. Kudrow、J. Heckey、A. Lvov、F. T. Chong 和 M. Martonosi，《ScaffCC：量子程序的可扩展编译与分析》，《并行计算》，第 45 卷，第 2–17 页，2015 年。[19] G. Li、L. Zhou、N. Yu、Y. Ding、M. Ying 和 Y. Xie，《基于投影的运行时断言：用于测试和调试量子程序》，《*ACM* 编程语言汇刊》，第 4 卷，OOPSLA 专刊，第 1–29 页，2020 年。[20] P. Long 和 J. Zhao，《使用多个子程序测试量子程序》，*arXiv* 预印本 *arXiv:2208.09206*，2022 年。[21]P. Matteo 和 P. Michael，《MorphQ：量子计算平台的蜕变测试》，2022 年。 [在线]。可访问：https://arxiv.org/abs/2206.01111[22] D. C. McKay、T. Alexander、L. Bello、M. J. Biercuk、L. Bishop、J. Chen、J. M. Chow、A. D. C´orcoles、D. Egger、S. Filipp等，《面向 OpenQASM 和 OpenPulse 实验的 Qiskit 后端规范》，*arXiv* 预印本 *arXiv:1809.03452*，2018 年。[23] M. Paltenghi 和 M. Pradel，《量子计算平台中的缺陷：一项实证研究》，*arXiv* 预印本 *arXiv:2110.14560*，2021 年。[24] P. Rebentrost、M. Mohseni 和 S. Lloyd，《用于大数据分类的量子支持向量机》，《物理评论快报》，第 113 卷，第 13 期，第 130503 页，2014 年。[25] N. Rutar、C. B. Almazan 和 J. S. Foster，“Java 缺陷查找工具比较”，载于第 *15* 届国际软件可靠性工程研讨会。IEEE，2004 年，第 245–256 页。[26] H. Shen、S. Zhang、J. Zhao、J. Fang 和 S. Yao，“XFind-bugs：面向 AspectJ 的扩展版 FindBugs”，载于第 *8* 届 *ACM SIGPLAN-SIGSOFT* 软件工具与工程程序分析研讨会论文集，2008 年，第 70–76 页。[27] K. Svore、A. Geller、M. Troyer、J. Azariah、C. Granade、B. Heim、V. Kliuchnikov、M. Mykhailova、A. Paz 和 M. Roetteler，“Q#：借助高级领域特定语言实现可扩展量子计算与开发”，载于*2018* 年真实世界领域特定语言研讨会论文集，2018 年，第 1–10 页。[28] S. Th´enault等，“Pylint”，*Python* 代码分析，2001 年。[29] D. Unruh，“量子关系霍尔逻辑”，*ACM* 编程语言论文集，第 3 卷，POPL 第 1 期，第 1–31 页，2019 年。[30]J. Wang、Y. Huang、S. Wang 和 Q. Wang，“发现静态缺陷查找器中的缺陷”，载于第 *30* 届 *IEEE/ACM* 国际程序理解会议论文集，ICPC ’22 系列。美国纽约：计算机协会，2022 年，第 516–527 页。[31] X. Wang、P. Arcaini、T. Yue 和 S. Ali，“通过搜索为量子程序生成失败测试套件”，载于国际搜索式软件工程研讨会。Springer，2021 年，第 9–25 页。[32]S. Xia 和 J. Zhao，“量子程序的静态纠缠分析”，载于*2023* 年 *IEEE/ACM* 第四届量子软件工程国际研讨会（*Q-SE 2023*）。IEEE，2023 年。[33]M. Ying，“量子程序的 Floyd–Hoare 逻辑”，*ACM* 编程语言与系统汇刊（*TOPLAS*），第 33 卷，第 6 期，第 1–49 页，2012 年。[34] N. Yu 和 J. Palsberg，“量子抽象解释”，载于第 *42* 届 *ACM SIGPLAN* 编程语言设计与实现国际会议论文集，2021 年，第 542–558 页。[35] P. Zhao、J. Zhao 和 L. Ma，“识别量子程序中的缺陷模式”，载于*2021* 年 *IEEE/ACM* 第二届量子软件工程国际研讨会（*Q-SE*）。IEEE，2021 年，第 16–21 页。[36]P. Zhao、J. Zhao、Z. Miao 和 S. Lan，“Bugs4Q：量子程序真实缺陷基准测试”，载于*2021* 年第 *36* 届 *IEEE/ACM* 自动化软件工程国际会议（*ASE 2021*）新想法与新兴成果专题。IEEE，2021 年，第 1373–1376 页。[37]L. Zhou、N. Yu 和 M. Ying，“应用量子霍尔逻辑”，载于第 *40* 届 *ACM SIGPLAN* 编程语言设计与实现会议论文集，2019 年，第 1149–1162 页。