MindLint: A Static Analysis Tool for MindQuantum-Based Applications

Xuan Mao[1,2], Zijie Huang[2], Jianxin Ge[2], Chao Wang[2], and Lizhi Cai[1,2,*]
[1]Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, China
[2]Shanghai Key Laboratory of Computer Software Testing & Evaluating, Shanghai Development Center of Computer Software Technology, Shanghai, China
maoxuan@mail.ecust.edu.cn, {huangzj, gjx, wc, clz}@sscenter.sh.cn
*corresponding author

Abstract—The rapid advancement of quantum computing has led to diverse implementation frameworks. However, existing academic research focuses on limited frameworks such as IBM Qiskit. To address industrial quality assurance needs for MindQuantum-based applications, we propose MindLint, the first static analysis tool specifically designed for MindQuantum. MindLint comprises three key components: 1) A semantic feature extractor using Abstract Syntax Tree (AST) analysis, 2) A rule-based checker to detect potential code issues, and 3) A reporting system with categorized warnings. Experiments with our collected real-world dataset show that MindLint effectively identifies errors and raises actionable warnings.

Keywords–Quantum Software Engineering; Static Analysis; MindQuantum; Software Quality Assurance

1. Introduction

Quantum computing is advancing so quickly that many call this century the quantum era [1]. Even so, quantum programs are fragile: a single wrong qubit index or early measurement can break a whole algorithm.

Executing quantum programs on real quantum hardware consumes significant computational resources, making it essential to perform comprehensive code verification prior to execution. To address similar problems, classical software development proposed numerous static code analysis methodologies [2], while the quantum computing domain lacks static analysis tools for quantum code verification.

Existing static analysis tools [3], [4], [5] primarily developed for IBM's Qiskit cannot be directly applied for semantic verification of code based on MindQuantum, a Huawei-proposed framework which is dominating certain markets (e.g., China). To bridge this gap, we propose **MindLint**, a static analyzer designed for MindQuantum programs. MindLint parses source code into an AST, extracts key features such as variable assignments and function calls, and feeds them to rule-based checkers for quantum-specific issues. By catching defects before execution, MindLint aims to raise the reliability and productivity of MindQuantum development. **MindLint and dataset are available at https://github.com/backordinary/MindLint.**

2. The Proposed MindLint Tool

2.1 Design Motivation and Architecture

Debugging quantum software is challenging because small mistakes can affect results silently before simulation or hardware execution. **MindLint** identifies such issues at compile time based on three **framework-agnostic** principles: (1) **Zero run-time overhead**, the analysis is purely syntactic; no quantum circuit is executed or simulated. (2) **Modular rule plugins**, i.e., each checker operates as an independent function registered in a rule map, allowing easy addition of new rules for MindQuantum, other quantum frameworks, or classical DSLs. (3) **Language-level portability**, i.e., MindLint only depends on the host language's AST, making it easy to extend to other Python-based frameworks or different programming languages. Porting MindLint requires only updating the feature extractor, without changing the core logic.

MindLint is built on a modular design to support these goals: (1) **Parsing and Feature Extraction**, each source file is parsed into an AST using Python's `ast` module in linear time. MindLint scans each source file once to extract two types of lightweight analysis data. For the first feature **MQ-Attributes**, it captures all variable assignments as `(name, value)` pairs, covering scalars, lists, and dictionaries. For the second feature **MQ-Operations**, it captures the text form of function calls, including chained invocations. For example, `Circuit().h(0).cx(1,0)` is represented as `["Circuit()", "h(0)", "cx(1,0)"]`. Together, these two types of features preserve critical semantic information while remaining neutral to specific frameworks. It ensures that MindLint's analysis is both accurate and widely adaptable. (2) **Rule Engine**, two feature streams obtained above are fed to three concurrent checkers (detailed in §2.2). Since each checker processes plain strings, extending MindLint is as simple as adding a new function to the rule map, without requiring front-end changes. Thread-level parallelism ensures that analysis remains sub-second even for projects of typical academic scale. (3) **Reporting**, findings are classified as **Errors** (mandatory fixes) or **Warnings** (advisory), tagged with the corresponding rule name and source line, and written to `./result/result_<filename>.txt`. If no issues are found, the report simply contains the line **"no error or warning"**, providing a clear signal.

2.2 Checker Suite and Rule Logic

MindLint evaluates two feature streams, i.e., **MQ-Attributes** (variable assignments) and **MQ-Operations** (function / gate calls), with three plug-in checkers executed in parallel.

- **Incorrect Initial State (IIS)**: verifies legal simulator backends, positive qubit counts, and ensures gate operands stay

DOI 10.1109/QRS-C65679.2025.00098

within the declared range.

- **Incorrect Measurement (IM)** : tracks measured qubits and warns if they are later reused as controllers.
- **Parameter Error (PE)** : flags duplicate use of the same qubit as both control and target or any misused parameters for gate and function call.

Each checker returns a list of findings in the form {`type, lineno, msg`} where `type`$\in$ {`Error, Warning`}. The reporting layer groups these findings; a completely clean run produces the single line **"no error or warining"**.

Since checkers consume plain-text tokens, adding a new rule requires implementing one function and registering it in the rule map, leaving the parser and existing detectors untouched.

2.3 End-to-End Workflow and CI Integration

MindLint is invoked from the command line with two flags: `--mode` (`1` for a single file, `0` for a recursive folder scan) and `--path` (target location). For every Python source file, the front-end immediately parses the code into an abstract syntax tree (AST) and extracts two feature streams, i.e., MQ-Attributes (variable–value pairs) and MQ-Operations (function and chained gate calls). These streams feed the rule engine, where the IIS, IM, and PE checkers execute concurrently through a lightweight `ThreadPoolExecutor`, allowing multi-file projects to be analysed in a few seconds on standard hardware. Each checker tags its findings with the rule identifier and source-line number, then hands them to the reporting layer. The reporter groups findings into **Errors** (must-fix defects) and **Warnings** (potential issues), renders each item as a concise, human-readable message, and writes the aggregated report to the destination text file.

MindLint introduces negligible overhead to development pipelines because it does not execute code on a quantum backend or simulator. It can be easily integrated into continuous integration (CI) workflows, such as a pre-commit hook, a GitHub Actions job, or a Jenkins stage, where any non-empty **Errors** section can automatically fail the build. MindLint's extensibility is also **CI-friendly**, i.e., adding a new rule requires only implementing a checker function and registering it in the `Rules` dictionary, with no further configuration needed for the next pipeline run.

3. Evaluation & Results

We create **the first MindQuantum defect corpus** by collecting code from MindQuantum tutorials, community repositories, contests held for MindQuantum, and research examples. After manual inspection we retained 28 programs: 18 that execute correctly and 10 that fail due to fatal faults (classified by MindLint as `Error`). Running MindLint with its IIS, IM, and PE rules produced the confusion matrix in Table I. The tool correctly flagged 5 / 10 defective programs and passed 18 / 18 clean ones, yielding an accuracy of 0.82, precision 1, recall 0.50 and F1 score 0.67. We further analysed the five defective programs that were not flagged as errors. Most failures were due to previously unconsidered API usage patterns that fell outside the scope of the current rules. Additionally, in some cases, although MindLint did not classify the issue as an `Error`, it still emitted a `Warning` on the relevant line, effectively guiding the programmer to review potentially problematic code manually. This suggests that even when defects escape strict error detection, MindLint can still assist users through softer, advisory signals. For the 18 clean samples, MindLint remains silent on correct code.

TABLE I
Performance of MindLint

Model	Accuracy	Precision	Recall	F1 Score
MindLint	**0.82**	**1.00**	**0.50**	**0.67**

4. Conclusion and Future Work

This paper[1] introduced **MindLint**, the *first* static-analysis tool designed specifically for Huawei's MindQuantum framework. MindLint combines an AST-based feature extractor with a modular, parallel rule engine and achieves 82 % accuracy on the first publicly documented MindQuantum defect corpus.

Our contribution is two-fold: 1) Mindchecker and the built dataset enable cross-framework comparative studies, offering further research opportunities to derive conclusions across different platforms, 2) MindLint lowers the barrier to writing reliable quantum code on MindQuantum and paves the way for framework-agnostic extensions.

Future work will (i) expand the defect corpus to improve statistical confidence, (ii) add cross-rule reasoning to capture composite fault patterns, and (iii) generalize the feature extractor so that MindLint can analyse other quantum programming ecosystems with minimal effort.

References

[1] M. Piattini, M. Serrano, R. Perez-Castillo, G. Petersen, and J. L. Hevia, "Toward a quantum software engineering," *IT Professional*, vol. 23, no. 1, pp. 62–66, 2021.

[2] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

[3] M. Paltenghi and M. Pradel, "Analyzing quantum programs with lintq: A static analysis framework for qiskit," *Proceedings of the ACM on Software Engineering*, vol. 1, pp. 95:1–95:23, July 2024.

[4] P. Zhao, X. Wu, Z. Li, and J. Zhao, "QChecker: Detecting bugs in quantum programs via static analysis," in *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*, pp. 50–57, 2023.

[5] Q. Chen, R. Câmara, J. Campos, A. Souto, and I. Ahmed, "The smelly eight: An empirical study on the prevalence of code smells in quantum computing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 358–370, 2023.

[1]Acknowledgment: this work was supported in part by the China Postdoctoral Science Foundation (2024M761927), the Shanghai Sailing Program (24YF2719900, 24YF2720000).