KQFuzz: Knowledge-Guided Fuzzing for Quantum Libraries via Large Language Models

Fuyuan Xia[1,†] Qixin Zhang[2,†] Chenhao Ying[1,*] Haojin Zhu[1] Shuai Wang[3] Yuan Luo[1,*] Pingchuan Ma[4,*] Yuxuan Du[2,*]

[1]Shanghai Jiao Tong University, {fuyuanxia, yingchenhao, zhu-hj, yuanluo}@sjtu.edu.cn
[2]Nanyang Technological University, {qixin.zhang, yuxuan.du}@ntu.edu.sg
[3]Hong Kong University of Science and Technology, shuaiw@cse.ust.hk
[4]Zhejiang University of Technology, pma@zjut.edu.cn

Abstract

As quantum computing continually improves, ensuring the reliability and correctness of quantum libraries has become increasingly critical. To this end, many LLM-based fuzzing approaches towards quantum libraries have been proposed to uncover potential bugs. However, these methods still suffer from limitations such as insufficient flexibility and low efficiency, which hinder the progress of the quantum computing field. To address these challenges, we propose **KQFuzz**, a novel knowledge-guided fuzzer for quantum libraries. It leverages comprehensive codebase knowledge to ground LLM-based test generation, synergizing this with fitness-guided evaluation and two-level mutations to explore complex execution paths and trigger potential bugs. Firstly, KQFuzz introduces a novel prompting scheme tailored to quantum programs, which strategically incorporates knowledge of the codebase to efficiently generate high-quality quantum seed programs. Moreover, we develop evaluation and mutation strategies to handle the generated seed programs, facilitating efficient fuzzing execution while further enriching the diversity of the resulting test cases. We implement KQFuzz and conduct fuzzing on three popular quantum libraries, including Qiskit, PennyLane, and Cirq. Experimental results demonstrate that our approach significantly outperforms other state-of-the-art methods, with coverage improved by up to 18.44%. During the development of KQFuzz, we discovered 13 bugs, all of which have been confirmed and 12 have already been fixed by the developers.

Keywords

Quantum Library, Fuzzing, Large Language Model.

ACM Reference Format:
Fuyuan Xia, Qixin Zhang, Chenhao Ying, Haojin Zhu, Shuai Wang, Yuan Luo, Pingchuan Ma, and Yuxuan Du. 2026. KQFuzz: Knowledge-Guided Fuzzing for Quantum Libraries via Large Language Models. In *Proceedings of 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

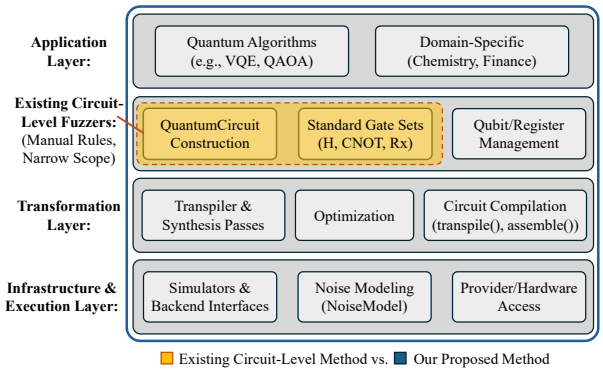

Figure 1: Comparison of Fuzzing Scopes.

1 Introduction

Diverse quantum libraries have been developed by academic institutions and technology companies to accelerate the design and implementation of quantum algorithms, motivated by the promise that quantum computing offers capabilities beyond classical computation across diverse domains [18, 61]. However, an often overlooked issue is that bugs in such quantum libraries may produce unexpected results, leading to flawed scientific or engineering conclusions and obscuring the true potential of quantum computing [2, 5, 14, 30, 33, 49]. Scarce quantum hardware further amplifies the impact of software bugs, leading to significant resource waste. In this regard, much like in classical software engineering, there is an increasing need for systematic quantum library testing to ensure correctness and reliability.

However, porting classical testing methodologies to the quantum domain is far from straightforward. Conventional techniques like static analysis [45] often fail because they lack the domain-specific semantics required to comprehend complex quantum states, making verification difficult. Furthermore, the rapid evolution of quantum libraries quickly renders manual testing rules obsolete, necessitating automated and adaptive approaches that can systematically explore the complex logic of these systems. To meet these requirements, fuzzing [25, 35] has emerged as a leading candidate for automatically discovering bugs. By generating and executing a vast number of randomized test cases, fuzzing can effectively explore the complex input space and identify unexpected software behaviors across diverse execution scenarios, which is uniquely suited for quantum libraries.

*Corresponding authors: Chenhao Ying, Yuan Luo, Pingchuan Ma and Yuxuan Du.
†Both authors contributed equally to this research.
This work was supported by Grant RS3/26 from the Ministry of Education (MOE), Singapore; the RGC GRF grant under the contract 16214723; and the National Natural Science Foundation of China (NSFC) under Grant 62402313.

ASE '26, Munich, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Limitations of existing fuzzing-based quantum library testing. In this work, quantum-library testing treats the library or platform implementation as the system under test, while quantum programs serve as test inputs. Existing quantum-library fuzzers mainly rely on domain-specific test generation [22, 28, 43], using dedicated generators, transformation rules, templates, or constraint specifications to construct structurally valid programs. However, these approaches necessitate extensive domain expertise to manually formulate generation and mutation rules. Moreover, as shown in Figure 1, such manual curation confines their scope to circuit-related APIs, rendering them inflexible and unable to exercise emerging software features or high-level orchestration logic. Consequently, they are incapable of identifying entire classes of bugs in the rapidly evolving quantum computing ecosystem.

Alternatively, general-purpose LLM-based fuzzers such as [56] offer a promising path forward, as they can use their domain knowledge to generate seeds invoking diverse APIs without manual rule design. However, instantiating LLM-based fuzzing for quantum programming libraries faces unique **C**hallenges compared to classical software.

C1. Low-validity seed generation. Unlike classical software with stable interfaces, the frequent restructuring of quantum APIs hinders LLMs from maintaining up-to-date internal representations of the evolving API surface and its associated semantic constraints. As illustrated in Figure 3 in § 3.2, our pilot study reveals that LLMs achieve only 35–46% validity for quantum programming libraries compared to 64–86% for classical programming libraries with disproportionately high import and attribute errors, averaging 118 such errors per 600 programs in each library.

C2. Constrained exploration in quantum programs. Unlike classical software, quantum programs are subject not only to hardware-level constraints such as limited qubit counts and restricted connectivity, but also to framework-level execution semantics and hybrid quantum–classical control flows. Conventional mutation strategies operating at the byte or abstract-syntax-tree level often fail to preserve these structural and semantic constraints, thereby generating invalid or non-executable programs. The core challenge is therefore to design domain-aware mutation mechanisms that systematically explore the input space while respecting quantum-specific structural, semantic, and complicated constraints.

Our solution. To address these challenges, we propose **KQFuzz**, a Knowledge-guided Quantum Fuzzing framework, which incorporates codebase-specific structural and semantic information to guide LLM-driven seed program generation for quantum libraries. Moreover, KQFuzz integrates fitness-guided evaluation and mutation mechanisms to systematically explore complex execution paths and expose latent defects.

To address **C1**, KQFuzz constructs an API corpus capturing four types of knowledge: (1) static metadata, such as signatures and locations; (2) API associations via three dimensions, *i.e.*, proximity, type coupling, and call relationships; (3) semantic models via LLM-based summarization; and (4) evolution metrics tracking API modifications across versions. During generation, KQFuzz employs a probabilistic selection strategy that prioritizes APIs with strong semantic associations and high evolutionary activity, steering the LLM to synthesize valid, version-aligned *seed programs* that target potentially bug-prone execution paths.

In response to **C2**, KQFuzz employs a two-level mutation strategy, where parameter-level mutation targets numerical corner cases and gate-level structural mutation explores variations through entanglement-aware substitutions. Furthermore, we introduce a fitness function that integrates gate diversity, entangled qubits, API diversity, and call depth. This function acts as a selection criterion to prioritize high-quality seeds, steering the LLM toward generating programs in more complex and potentially bug-prone regions. Seeking a balance between precision and throughput, KQFuzz adopts a tiered synergy: a high-capacity model (*i.e.*, Gemini-3 [48]) performs one-time API semantic modeling to ground a cost-effective LLM (*e.g.*, Qwen 2.5-Coder) for iterative fuzzing generation. Finally, we execute the resulting *test cases*—comprising both the generated seed programs and their mutated variants—and apply heuristic filtering to the execution results to identify impactful bugs within the target quantum libraries.

We evaluate the performance of KQFuzz on three mainstream quantum libraries: Qiskit[1], PennyLane[2], and Cirq[3]. KQFuzz significantly outperforms existing baselines in code validity and bug-finding effectiveness, discovering 13 bugs, all of which have been confirmed and 12 already fixed by the developers. Our contributions are summarized as follows.

- We propose KQFuzz, a novel knowledge-guided quantum fuzzing framework that integrates codebase-specific insights with LLM-based test generation and domain-specific mutation to systematically explore complex execution paths in quantum libraries.
- We construct a comprehensive API corpus encompassing four dimensions of knowledge. Based on this, a seed program generation strategy is designed to steer the LLM toward generating valid, version-aligned, and potentially bug-prone quantum programs.
- We introduce a two-level mutation strategy (parameter and gate-level) and a multi-dimensional fitness function that accounts for gate diversity and entanglement. This approach prioritizes high-quality seeds to effectively navigate the complex state space of quantum programs.
- We perform a comprehensive evaluation of KQFuzz on three mainstream quantum libraries. The results demonstrate its superiority over existing baselines, surpassing the state-of-the-art coverage by a margin of up to 18.44%, successfully identifying 13 new bugs, all of which have been confirmed by developers and 12 of which are already fixed.

2 Background

2.1 Basic Concepts in Quantum Computing

Quantum computing leverages principles such as superposition and entanglement to enable new computational capabilities [17]. In practice, these capabilities are commonly formulated within the *quantum circuit model* [16], in which computation is represented as a sequence of elementary quantum gates, *e.g.*, single-qubit and

[1] https://github.com/Qiskit/qiskit
[2] https://github.com/PennyLaneAI/pennylane
[3] https://github.com/quantumlib/Cirq

entangling two-qubit operations, acting on quantum states. After that, a measurement process is applied to extract information from the final quantum state into classical registers, yielding probabilistic outcomes that encode the solution to a given problem [1, 9]. Refer to Supplementary Material (**SM**) A for more details.

Within the quantum computing ecosystem, Qiskit [24], PennyLane [6], and Cirq [10] are three major quantum programming libraries maintained by major technology companies or quantum-focused companies. While these libraries share extensive functional commonalities in circuit synthesis, simulation, and hardware integration, they provide alternative development environments that cater to different hardware-specific optimizations and research workflows. Despite their widespread adoption, these libraries remain in a state of perpetual and volatile transformation to keep pace with rapid hardware and algorithmic breakthroughs [43]. These rapid shifts often result in hidden defects, leading to persistent logic anomalies and unexpected behaviors that deviate from intended specifications. These recurring bugs represent a significant bottleneck that necessitates rigorous testing and validation to ensure the integrity of the quantum computing ecosystem.

2.2 Fuzzing

Fuzzing foundations and LLM-based evolution. Fuzzing is a cornerstone of automated software testing, designed to identify bugs and edge-case behaviors by executing a target system with a vast array of synthesized inputs [7, 8, 15, 46, 47, 55, 58, 68]. Traditional fuzzing taxonomies primarily distinguish between generation-based approaches, which construct inputs from predefined grammars, and mutation-based strategies, which apply stochastic transformations to existing seeds to maximize code coverage [13, 35]. While effective for low-level protocols, these conventional methods often struggle with high-level software that requires complex, semantically valid structures. Recently, LLMs have redefined this paradigm. By leveraging their inherent semantic comprehension and code synthesis capabilities, LLM-driven fuzzers can autonomously generate syntactically sophisticated and contextually rich test cases [12, 56]. This shift alleviates the burden of manual rule engineering and enables the exploration of deep logic within complex API sequences.

Fuzzing in the quantum libraries. The application of fuzzing to quantum libraries has bifurcated into two primary paradigms: circuit-level construction and LLM-driven exploration. Circuit-level fuzzers typically employ domain-specific generators, transformation rules, or templates to generate gate sequences, ensuring high validity but often remaining confined to low-level gate logic [28, 43]. In contrast, LLM-based methods harness extensive semantic knowledge to probe high-level behaviors in quantum programming libraries. A persistent objective in this field is to effectively bridge the gap between the expressive flexibility of LLM-generated test cases and the stringent syntactic requirements of quantum libraries, ensuring that the generated programs remain both diverse and executable.

3 Motivation & Challenges

```
# <IMPORT TEMPLATE>
qc = QuantumCircuit(11, 11, name='qc')
qc.append(ECRGate(), qargs=[qc.qubits[2], qc.qubits[5]], cargs=[])
qc.append(CHGate(), qargs=[qc.qubits[5], qc.qubits[7]], cargs=[])
qc.append(ZGate(), qargs=[qc.qubits[5]], cargs=[])
qc.append(ZGate(), qargs=[qr[7]], cargs=[])
qc.append(RCCXGate(), qargs=[qr[8], qr[2], qr[6]], cargs=[])
qc.append(ZGate(), qargs=[qr[0]], cargs=[])
qc.append(iSwapGate(), qargs=[qr[1], qr[3]], cargs=[])
qc.append(SdgGate(), qargs=[qr[2]], cargs=[])
qc.append(CXGate(), qargs=[qr[3], qr[1]], cargs=[])
# <RESULT TEMPLATE>
```

(a) Code Generated by Circuit-Level Method

```
# <QUANTUM CODE>
from qiskit.circuit.library import grover_operator
oracle = QuantumCircuit(3)
oracle.h(2)
oracle.ccx(0, 1, 2)
oracle.h(2)
grover_op = grover_operator(oracle) # Involved API
qc = QuantumCircuit(3)
qc.h([0, 1, 2])
qc.append(grover_op, [0, 1, 2])
qc.measure_all()
# <QUANTUM CODE>
```

(b) Realistic Quantum Program

Figure 2: Comparison between circuit-level fuzzer-generated code and realistic quantum program.

3.1 Revisiting Fuzzing for Quantum Libraries

We summarize two typical families of existing solutions for fuzzing quantum libraries as follows. First, **circuit-level fuzzers** aim to generate random quantum circuits to trigger bugs in the quantum software stack [28, 43]. Their primary approaches use domain-specific generators, transformation rules, templates, or constraint specifications to rapidly produce valid quantum programs, prioritizing the efficient discovery of structural defects such as crashes or incorrect circuit outputs. Nevertheless, this design entails an inherent trade-off between efficiency and generality. By enforcing predefined structural rules to ensure efficient and valid circuit generation, such approaches limit their applicability to a broader range of quantum programs. Consequently, these approaches face two key challenges: (1) manually designed rules become increasingly difficult to maintain as quantum programming libraries evolve and introduce new operations; and (2) their highly specialized generation mechanisms primarily target structural circuit anomalies, limiting their ability to expose the diverse defects that arise in complex and real-world quantum libraries. As indicated in Figure 2(a), MorphQ [43] can generate legal circuits containing a large number of random quantum gates. However, in practice (*e.g.*, the deployment of the Grover algorithm on real quantum hardware), additional APIs beyond circuit gates are required, such as `grover_operator` in Figure 2(b). For such APIs that go beyond the circuit level, potential bugs cannot be uncovered by these methods, thus constraining the applicability of circuit-level fuzzers to quantum libraries.

While fuzzers at the circuit level often suffer from limited flexibility, approaches based on LLMs have emerged as a promising alternative [11, 56]. Attributed to the extensive pretraining on diverse code corpora, LLMs have the potential to effectively capture the complex semantic logic of quantum libraries to generate diverse test cases. However, *the syntactic and semantic validity of quantum code generated by LLMs remains a significant bottleneck.* As reported

in [56], only 24.9% of Qiskit code generated by LLMs is valid, a figure that pales in comparison to the 100% validity rate achieved by circuit-level fuzzers such as MorphQ [43]. This high failure rate severely constrains the practical utility of LLMs in the domain of quantum library testing.

3.2 Motivating Study

As discussed in (§ 1), LLMs can be leveraged to generate fuzzing inputs targeting a given library. To systematically assess the capability of LLMs to generate fuzzing inputs for quantum libraries, we conduct a preliminary empirical comparison between quantum libraries and their well-established classical analogues.

Experimental configuration. We select three prominent quantum libraries, Qiskit, Cirq, and PennyLane, as our primary targets. To provide a comparative baseline, we also include three widely used classical libraries: Numpy[4], Pandas[5], and PyTorch[6]. We adopt a representative LLM-based fuzzing pipeline similar to [31], utilizing a Chain-of-Thought (CoT) [54] prompting paradigm to guide the models through library importation, data preparation, and API invocation. For each library, we randomly sample 300 distinct APIs and employ two open-source models, CodeLlama-13B [44] and Qwen2.5-Coder-14B [23], to generate one test case per API. We select models at this scale as they strike an optimal balance between robust reasoning capabilities and local reproducibility. This setup yields a total of 600 test cases per library, allowing us to evaluate the models' zero-shot generation capabilities without domain-specific knowledge.

Quantifying the performance gap. As illustrated in Figure 3, our results reveal a stark disparity in code validity between domains. While classical libraries achieve a pass rate of 64.50% to 86.50%, the validity for quantum libraries plummets to a range of 35.33% to 46.00%. Our empirical results in Figure 3 show that *TypeError*, *AttributeError*, and *ValueError* are the most frequent errors (*e.g.*, 164 TypeErrors for Qiskit), while *ImportError* and *ModuleNotFoundError* are also common. These findings demonstrate that despite their strong general coding capabilities, LLMs exhibit a pronounced lack of domain awareness when encountering quantum-specific syntax and semantics.

Root cause analysis. A representative example illustrated in Figure 4 reveals that such failures are primarily attributed to the volatile evolution and frequent breaking changes inherent in quantum libraries. Unlike classical domains where software updates are primarily driven by application logic, these reconfigurations are uniquely induced by *the rapid advancement of quantum hardware*. As new functionalities and physical architectures emerge, library maintainers must frequently reconfigure invocation patterns and structural placements to accommodate hardware-side capabilities. Such instability leads to inconsistent internal representations within LLMs, heightening their proclivity for generating deprecated or hallucinated code. Furthermore, unlike the mature classical computing ecosystem, there is a pronounced scarcity of publicly available codebases aligned with the latest versions of quantum libraries. Consequently, the available resources for knowledge acquisition are largely restricted to the libraries themselves and the LLM's own (often flawed) outputs. These observations motivate us to investigate the following guiding questions: ❶ *How can library-specific knowledge be leveraged to guide LLMs in generating valid seed programs,* and ❷ *how can these seeds be systematically evolved to expose latent bugs in quantum libraries?*

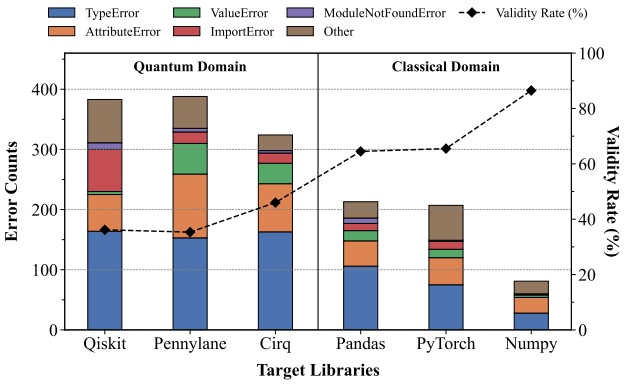

Figure 3: Empirical evaluation of the validity and error patterns in LLM-generated fuzzing test cases across quantum and classical libraries. The bars represent absolute error counts (left y-axis), while the dashed line indicates the validity rate (right y-axis).

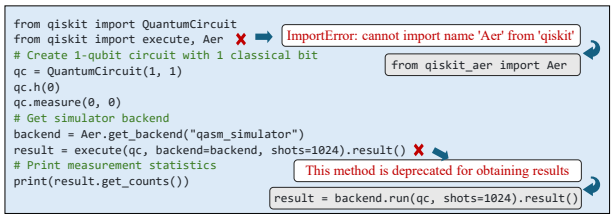

Figure 4: Code example of typical errors in quantum computing libraries generated by LLMs.

3.3 Challenge

To address the aforementioned guiding questions, a fundamental prerequisite is to understand the key challenges that arise when applying LLM-based fuzzing to quantum libraries. These challenges are inherently progressive: generating valid, library-aligned test cases is the foundation, upon which effective exploration through mutation must be built.

Challenges in generating valid quantum seeds. LLMs struggle to generate valid test cases for quantum libraries due to the rapid evolution and frequent restructuring of APIs, which lead to outdated or hallucinated invocations. This issue is further exacerbated by the scarcity of up-to-date quantum code, limiting the model's ability to learn stable usage patterns and diverse invocation semantics. Furthermore, since fuzzing is a resource-intensive process requiring high-volume iterations, a fundamental challenge lies in balancing computational cost with testing effectiveness. As a result, seed generation becomes a knowledge-grounding problem, requiring alignment with current library implementations (*i.e.*, source code), while also balancing cost and effectiveness for large-scale fuzzing with open-source models.

[4] https://github.com/numpy/numpy
[5] https://github.com/pandas-dev/pandas
[6] https://github.com/pytorch/pytorch

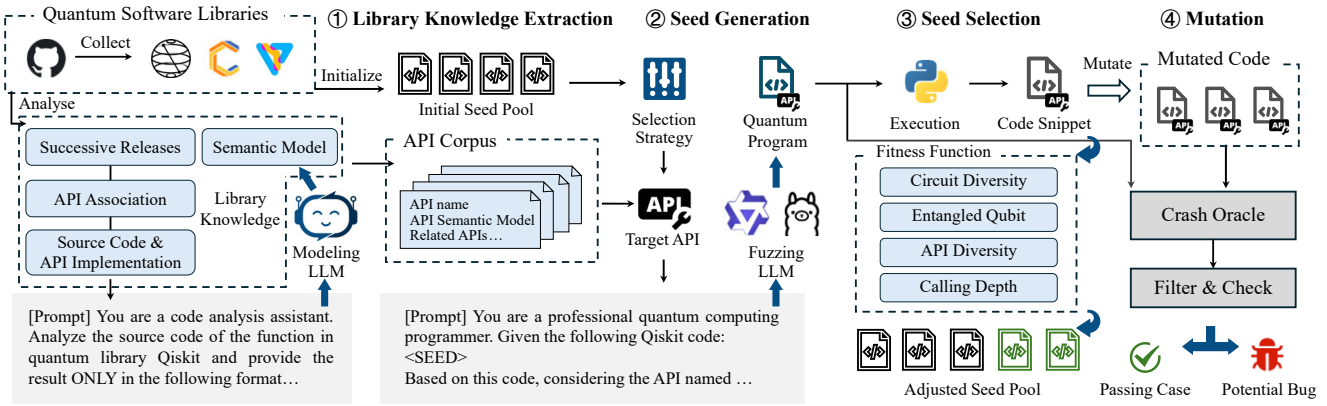

Figure 5: The workflow of KQFuzz for detecting bugs in quantum libraries.

Constraints and inefficiencies in quantum exploration. The subsequent exploration phase remains challenging even with valid seeds. Effective exploration of the program space typically relies on an iterative process of mutation, fitness evaluation, and input regeneration. However, directly applying traditional fuzzing strategies in the quantum setting results in severely limited exploration. First, at the mutation level, quantum programs are governed by hardware-imposed structural constraints and framework-defined execution semantics. Generic mutation operators (*e.g.*, byte-level or AST-level transformations) often fail to preserve these constraints, producing invalid programs and significantly reducing the proportion of executable test cases. Second, at the evaluation and selection level, even when syntactically and structurally valid variants are generated, the lack of domain-specific fitness metrics limits the ability to effectively prioritize high-value seeds. Without quantum-aware fitness functions to measure traits like quantum circuit complexity, the fuzzer cannot prioritize high-potential candidates. Consequently, the entire exploration loop becomes inefficient, trapped in generating redundant inputs with limited capability to evolve simple seeds into complex test cases. In this regard, holistic and domain-specific adaptations across the mutation and selection mechanisms are highly demanded to expose latent bugs.

4 Approach

KQFuzz addresses a central challenge in quantum-library fuzzing: generating tests that remain executable under evolving APIs while being diverse enough to exercise deep library logic. Existing circuit-level fuzzers provide high validity but limited API coverage, whereas general-purpose LLM fuzzers cover broader APIs but often rely on outdated API knowledge and generate invalid tests. Moreover, generic mutation and selection strategies lack quantum-aware guidance for parameterized gates, entanglement structures, and multi-API interactions.

To bridge this gap, KQFuzz employs a four-stage validity-to-diversity pipeline as illustrated in Figure 5. First, in **Library Knowledge Extraction** (Stage ①, § 4.1), KQFuzz constructs an API corpus by extracting structural associations, semantic profiles, and evolution metrics from the library code. Subsequently, during the **Seed Generation** (Stage ②, § 4.2) phase, a Fuzzing LLM takes seeds from the pool and incrementally incorporates target APIs to generate diverse quantum programs. Next, in **Seed Selection** (Stage ③, § 4.3), the generated programs are executed and assessed via a multi-dimensional fitness function. The generated quantum programs with high semantic complexity are fed back into the adjusted seed pool for the next iteration. Finally, the **Mutation** (Stage ④, § 4.4) phase systematically transforms quantum programs to expand the search space, while a crash oracle and semantic filters are employed to identify potential bugs.

4.1 Knowledge Modeling from Library Code

As identified in our pilot study (§ 3.2), LLM-based fuzzers frequently generate invalid seed programs due to outdated library knowledge or hallucinated API usage. To mitigate these issues, KQFuzz constructs an explicit knowledge model directly from the source code of the target quantum library. Rather than relying on parametric knowledge embedded in the LLM, KQFuzz extracts lightweight, library-specific structural and semantic cues and organizes them into an API corpus that guides subsequent seed generation.

API inventory and static metadata. KQFuzz begins by constructing an inventory of available APIs from the official documentation of the target software. Let $\mathcal{I} = \{1, 2, \ldots, N\}$ denote the resulting set of APIs. For each API $i \in \mathcal{I}$, KQFuzz collects static metadata by inspecting the corresponding source files, including the API signature, its implementation body, and its file and module location. This information serves as the foundation for modeling relationships among APIs and for providing accurate and version-aligned context to the LLM during seed synthesis.

API association modeling. To capture semantic and structural relationships between APIs, KQFuzz models pairwise API associations using three complementary dimensions. These dimensions are designed as lightweight signals derived from source-level inspection. In particular, *Proximity*, denoted by $S_p(a, b)$, measures the structural locality via file and module boundaries; *Type Overlap*, denoted by $S_t(a, b)$, quantifies the shared domain-specific input and output types; and *Call Relationships*, denoted by $S_c(a, b)$, evaluates direct reachability and shared invocation patterns. We aggregate these three dimensions into a joint association score, *i.e.*,

$$S(a, b) = w_1 S_p(a, b) + w_2 S_t(a, b) + w_3 S_c(a, b), \quad (1)$$

where w_1, w_2, and w_3 control the relative importance of these dimensions. Detailed mathematical formulations for each dimension are provided in SM B.

API implementation of semantic modeling. KQFuzz introduces an API implementation of a semantic model by prompting an LLM with source code to infer descriptions, constraints, outputs, and fuzzing points (detailed prompt in SM C). The modeling LLM is limited to this semantic summarization; API inventory construction and the computation of association and evolution metrics remain non-LLM source-analysis tasks. The resulting semantic model is a structured semantic summary grounded in the library source code. Unlike resource-intensive seed generation, semantic extraction is a *one-time preprocessing step*. We employ Gemini-3 [48] to synthesize library profiles, using its extended context window to capture long-range dependencies in complex codebases. To clarify the distinct roles of LLMs within our framework, we designate the LLM used for this extraction as the *modeling LLM*, while the LLM responsible for subsequent seed generation is referred to as the *fuzzing LLM*.

API evolution modeling. In addition to structural associations, KQFuzz models API evolution across library versions. APIs that undergo frequent modifications may expose unstable or under-tested behaviors and thus represent promising fuzzing targets. For a given API $i \in \mathcal{I}$, denote B'_i and B_i as the implementation in two successive versions, with B'_i preceding B_i. We quantify the evolution weight $M'(i)$ as

$$M'(i) = 1 - \frac{|\mathcal{G}_n(B_i) \cap \mathcal{G}_n(B'_i)|}{|\mathcal{G}_n(B_i) \cup \mathcal{G}_n(B'_i)|},$$

where $\mathcal{G}_n(\cdot)$ extracts n-gram sets from the *normalized Abstract Syntax Tree (AST) node sequences* of the implementations. To further prioritize unstable APIs, we map $M'(i)$ to the degree of modification:

$$M(i) = 1 + \delta \cdot \tanh(\gamma(M'(i) - 0.5)). \tag{2}$$

This non-linear mapping amplifies the testing priority of high-evolution APIs while suppressing that of stable ones, where γ and δ are hyper-parameters controlling the sensitivity and influence range, respectively.

Remark. All extracted metadata, association scores, semantic model, and evolution metrics are integrated into a unified API corpus, which is subsequently used to guide API selection and prompt construction during seed generation for fuzzing.

4.2 LLM-Guided Seed Generation

Building on the API corpus constructed in (§ 4.1), KQFuzz iteratively generates seed programs for fuzzing by combining probabilistic API selection with LLM-based code synthesis. The key idea is to incrementally extend existing quantum programs with carefully selected APIs, while using library-specific knowledge to constrain and guide the LLM toward valid and semantically meaningful code.

Probabilistic target API selection. Given a seed program containing an invocation of API $i \in \mathcal{I}$, KQFuzz queries the API corpus to retrieve the set of related APIs, *i.e.*,

$$\mathcal{S}_i = \{j \in \mathcal{I} \mid S(i, j) \neq 0\},$$

where $S(i, j)$ is the association score defined in Eqn. (1). Instead of selecting a related API uniformly at random, KQFuzz leverages the metrics obtained from knowledge modeling to bias selection toward APIs that are both strongly associated with the seed API and actively evolving. Specifically, it assigns each candidate API $k \in \mathcal{S}_i$ a selection probability:

$$\Pr[k; i] = \frac{M(k) \cdot S(i, k)}{\sum_{j \in \mathcal{S}_i} M(j) \cdot S(i, j)}, \tag{3}$$

where $S(i, k)$ and $M(k)$ are defined in Eqn. 1 and Eqn. 2, respectively. As a result, KQFuzz prioritizes semantically meaningful API combinations while avoiding the forced integration of unrelated APIs that often lead to invalid or trivial programs.

Prompt construction and code generation. Given a selected seed program and a target API $i_t \in \mathcal{I}$ sampled according to the probabilities in Eqn. (3), KQFuzz invokes the fuzzing LLM to synthesize a new seed program that incorporates i_t into the seed program. The goal of prompt construction is to (i) ground the LLM in the *exact* library version under test, (ii) provide sufficient semantic context to avoid hallucinated usage, and (iii) steer generation toward either introducing i_t or exercising it thoroughly, depending on the stage of fuzzing.

Accordingly, KQFuzz constructs prompts by integrating three key sources of information: (i) the selected seed program, which provides the execution context and structural skeleton; (ii) the semantic correlation between existing APIs in the seed and the target API i_t, ensuring that the newly introduced API is contextually relevant; and (iii) the API implementation semantic model of i_t (detailed prompt in SM C). Unlike approaches that rely on the LLM's internal parametric knowledge, KQFuzz provides an explicit knowledge representation synthesized from the source code. This semantic model encapsulates inferred descriptions, argument constraints, expected outputs, and critical fuzzing points. By presenting these structured semantic profiles, KQFuzz explicitly grounds the generation process in the exact library logic, exposing intricate dependencies and usage patterns that are often absent from outdated training data. Moreover, by promoting the generation of programs that integrate multiple semantically related APIs, KQFuzz produces seed programs that exercise more complex and diverse execution behaviors, thereby increasing coverage of the target quantum library.

Comparison with standard RAG. While the integration of an API corpus shares the high-level philosophy of Retrieval-Augmented Generation (RAG) [29], namely augmenting LLMs with external knowledge, our mechanism differs fundamentally from standard RAG pipelines. Specifically, standard RAG relies on vectorizing documents and retrieving text chunks based on semantic similarity to a user query. However, in the highly constrained domain of quantum libraries, semantic similarity alone does not ensure structural compatibility or valid program execution. In addition, instead of treating knowledge as isolated text snippets, our approach constructs a structured and multi-dimensional knowledge model that captures API relationships, semantic properties, and historical evolution. This design enables the fuzzer to preserve structural constraints while prioritizing components with higher defect potential. As a result, our approach provides a more principled and robust basis than conventional text-based retrieval methods.

4.3 Seed Selection

KQFuzz follows an iterative generation-selection loop. At each iteration, a batch of new seed programs is generated from the current seed pool, after which high-quality programs are selected and promoted as seeds for subsequent iterations. Below, we separately

describe how the seed pool is initialized and how the generated programs are prioritized.

Seed pool initialization. KQFuzz initializes the seed pool using officially provided usage examples extracted from API docstrings of the target quantum library. The number of initial seeds is comparable to the number of APIs, ensuring broad coverage of library functionality. Importantly, all seed programs are derived exclusively from the target library itself, avoiding reliance on external quantum code that may be outdated or incompatible.

Fitness-guided seed selection. KQFuzz evaluates generated seed programs using a fitness function and prioritizes high-quality programs for reseeding. For each generated program, KQFuzz first extracts the largest executable fragment using a greedy strategy, ensuring that fitness is computed only on runnable code. For example, consider a 15-line program that crashes at line 12. In this case, the fragment consisting of the first 11 lines constitutes the largest executable fragment. It then analyzes the structure of each generated seed program and assigns a fitness score that quantifies its exploration potential. The fitness function is designed to favor programs that exercise complex quantum operations and diverse API interactions. Specifically, KQFuzz evaluates each program along the following dimensions: ***Circuit Gate Diversity (G).*** The number of distinct quantum gate types appearing in the circuit. Higher diversity indicates broader coverage of quantum operations, increasing the likelihood of exposing bugs triggered by specific gate combinations. ***Number of Entangled Qubits (Q).*** The number of qubits participating in entangled operations. Entanglement amplifies circuit complexity and may expose subtle compiler or execution errors, although excessively large entangled states may incur the hardness of efficient simulation. ***API Diversity (Δ).*** The number of distinct APIs invoked in the quantum program. This metric reflects semantic coverage across different library components, such as circuit construction, simulation, and backend configuration. ***API Calling Depth (L).*** The maximum depth of API call chains, capturing the temporal and logical complexity of API interactions. Longer call chains stress long execution paths and stateful behaviors. Combining these factors, the fitness score of a generated program C is defined as

$$\text{FitnessScore}(C) = G + \min(Q, \tau) + L + \eta \cdot \Delta. \tag{4}$$

To prevent the fuzzer from over-optimizing for entanglement at the expense of executability, the term Q is capped at τ. This design penalizes excessively deep entanglement that threatens to crash the target library. Furthermore, η ($\eta \geq 1$) serves as a tunable weight parameter designed to balance the diversity of API invocations (Δ) against the other evaluation metrics, ensuring that API diversity is appropriately prioritized without overwhelming the fitness score.

In each iteration, KQFuzz selects the top-κ programs and adds them to the seed pool for the next round. This fitness-guided prioritization biases the fuzzing process toward semantically rich test cases, enabling progressively broader and more complex exploration of the target quantum library over time.

4.4 Mutation and Test Oracle

Following LLM-guided generation, KQFuzz expands the search space through structured mutation and utilizes a crash-based test oracle for bug detection. Mutation complements generation by systematically exploring semantic variations that are unlikely to be produced by prompting alone, while the test oracle provides automated bug detection indicators during execution.

Mutation. KQFuzz applies semantics-aware mutations to generated test cases by directly operating on quantum circuits. The mutation strategy consists of two levels: parameter-level mutation and gate-level structural mutation.

Parameter-level mutation. Many quantum gates require numerical parameters (*e.g.*, rotational angles in quantum gates R_x, R_y, and R_z). In parameter-level mutation, KQFuzz replaces such parameters with values sampled from a predefined set of special constants. These constants include values historically associated with bugs as well as typical boundary or representative values, such as ± 1, 0, and π. This mutation targets numerical corner cases while preserving syntactic and semantic validity of the program.

Gate-level structural mutation. Unlike parameter-level mutation, which changes numerical values while preserving circuit topology, gate-level mutation modifies multi-qubit interaction structures. KQFuzz substitutes circuit gates using three classes of rules: (i) entanglement-preserving, (ii) entanglement-expanding, and (iii) entanglement-reducing. These mutations generate structurally different quantum inputs that exercise topology-sensitive logic in compilation, decomposition, simulation, and backend-specific processing. Although such variants may execute already-covered code regions, they can expose failures that are difficult to trigger through parameter mutation alone.

Test oracle. KQFuzz utilizes a crash-based oracle to identify runtime anomalies within the executed test cases. To maintain high detection precision, KQFuzz performs exception filtering based on error messages and applies predefined heuristic rules to eliminate common false-positive patterns. The remaining candidate anomalies are then manually inspected to determine if they trigger genuine bugs in the target library. This workflow ensures that KQFuzz identifies impactful bugs while minimizing noise from trivial execution failures.

5 Experimental Setup

5.1 Research Questions

To enable a comprehensive and rigorous evaluation of the proposed KQFuzz framework, we articulate three key research questions as follows.

RQ1: How does KQFuzz perform compared to existing state-of-the-art fuzzing techniques?
RQ2: What drives KQFuzz's effectiveness, and how robust is it across different LLM backends and model scales?
RQ3: To what extent can KQFuzz uncover previously unknown bugs in widely used quantum programming libraries?

5.2 Implementation

We implement a prototype of KQFuzz as a Python-based framework and conduct a multi-faceted evaluation on a high-performance computing platform. All experiments are performed on a Linux server powered by an Intel Xeon Platinum CPU (3.80 GHz, 192 physical cores), 512 GB RAM, and an NVIDIA Tesla H800 GPU, operating

Table 1: Overview of mainstream quantum libraries.

Library	Version	GitHub Stars	LoC	API Number	Token
Qiskit	2.3.0	7.1K	45928	788	0.38M
PennyLane	0.44.0	3.1K	69514	1206	0.61M
Cirq	1.6.1	4.9K	31774	866	0.41M

Table 2: Comparison with baseline fuzzers on three quantum libraries.

Target	Fuzzer	# Test Cases	Coverage	# Unique Crashes
Qiskit	FuzzQ	42985	11636(25.34%)	5
	MorphQ	25714	11396(24.81%)	13
	Fuzz4All	23478	24344(53.00%)	339
	KQFuzz	35952	29078(63.31%)	685
Pennylane	Fuzz4All	21823	31584(45.44%)	426
	KQFuzz	32068	40814(58.71%)	1027
Cirq	FuzzQ	27067	10027(31.56%)	3
	Fuzz4All	22547	18286(55.35%)	391
	KQFuzz	34224	23446(73.79%)	706

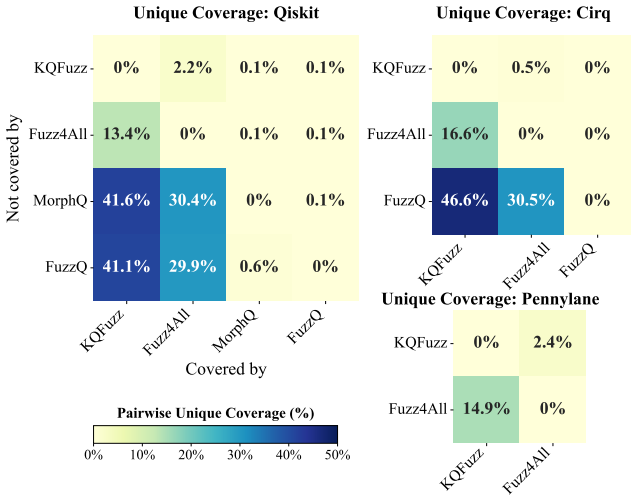

Figure 6: Pairwise unique coverage comparison.

under Ubuntu 24.04 LTS. Our evaluation targets three representative quantum computing libraries: Qiskit, PennyLane, and Cirq. The detailed specifications and statistics of these target libraries are summarized in Table 1. Specifically, the Token column denotes the total number of output tokens generated by Gemini-3 [48] during API semantic modeling. For seed program generation, we leverage the Ollama platform to deploy the INT8-quantized versions of CodeLlama [44] and Qwen2.5-Coder [23], balancing inference efficiency with model performance. To maximize hardware throughput and fully utilize the multi-core CPU resources, we implement a multi-processing execution engine that facilitates parallel test execution. Furthermore, to prevent potential resource exhaustion or hangs caused by malformed programs, a strict 20-second execution budget is enforced for each test case. To evaluate the effectiveness of KQFuzz, we compare it against three state-of-the-art baselines: two quantum-specific fuzzers, MorphQ [43] and FuzzQ [28], and one LLM-based fuzzer, Fuzz4All [56]. A detailed summary of the core hyperparameters, configuration settings, and description of these baselines is provided in SM C.

6 Evaluation

6.1 Comparison with Other Fuzzers

To answer RQ1, we compare KQFuzz with each baseline only on targets evaluated by its released implementation: MorphQ on Qiskit, FuzzQ on Qiskit and Cirq, and Fuzz4All on all three libraries. Although the FuzzQ artifact includes a PennyLane adapter, it is presented only as a future-work demonstration rather than an evaluated configuration. We do not port the baselines to additional target libraries, because the resulting performance could be influenced by our implementation choices and might not faithfully represent the original released tools. For every applicable comparison, each fuzzer was executed for a continuous 24-hour period on the same pinned versions (Qiskit 2.3.0, PennyLane 0.44.0, and Cirq 1.6.1), with baseline configurations and hyperparameters set to the defaults recommended in the original papers. Table 2 summarizes the number of generated programs, the resulting code coverage, and the number of unique crashes. Here, we measure Python line coverage using `coverage.py` while excluding the libraries' internal test files, and we regard a generated program as valid only if it executes without runtime exceptions in a properly configured environment and invokes the target API at least once. To better compare how much new behavior each fuzzer reaches, Figure 6 further visualizes the pairwise unique coverage relationships between KQFuzz and the baselines. Each cell in the heatmap represents the percentage of the target library's total code lines that are covered by one fuzzer but missed by another.

Experimental results show that KQFuzz consistently achieves the strongest overall testing effectiveness across all targets. On Qiskit, KQFuzz reaches 63.31% coverage, outperforming Fuzz4All (53.00%) and more than doubling the coverage of MorphQ (24.81%) and FuzzQ (25.34%). On PennyLane, KQFuzz improves coverage from 45.44% to 58.71% over Fuzz4All. On Cirq, KQFuzz achieves 73.79% coverage, compared with 55.35% for Fuzz4All and 31.56% for FuzzQ. This broader exploration also translates into more bug-finding opportunities: KQFuzz reports the highest number of unique crashes on every framework, including 685 on Qiskit, 1027 on PennyLane, and 706 on Cirq. Moreover, it generates the largest number of programs in all three settings, indicating that KQFuzz combines higher throughput with stronger exploration. A detailed analysis of the overall time allocation under the default configuration used in RQ1 is provided in SM E.1.

The pairwise heatmaps in Figure 6 confirm that KQFuzz's gains represent substantial new behaviors rather than redundant paths. On Qiskit, KQFuzz covers an additional 13.4% of the total library code that Fuzz4All fails to reach. In contrast, KQFuzz misses only 2.2% of the library code covered by Fuzz4All. The gap is even more pronounced against specialized quantum fuzzers: MorphQ and FuzzQ fail to cover 41.6% and 41.1% of the total library code, respectively, that is otherwise successfully explored by KQFuzz. Similarly, on Cirq, KQFuzz misses only 0.5% of the code reached by Fuzz4All, while covering an additional 16.6% of the library. On PennyLane, KQFuzz contributes 14.9% unique library coverage beyond Fuzz4All, while missing only 2.4%. These results indicate that KQFuzz largely subsumes the coverage achieved by prior works while significantly

Table 3: Ablation results on three quantum libraries.

Target	Configuration			% valid	Coverage	Unique Crashes
	Seed	Repo	Mut.			
Qiskit	×	×	×	33.72%	22379(48.73%)	155
	✓	×	×	40.69%	23189(50.49%)	165(+10)
	×	✓	×	50.42%	27945(60.85%)	226(+71)
	✓	✓	×	53.71%	28316(61.65%)	250(+95)
	✓	✓	✓	53.71%	28530(62.12%)	307(+152)
PennyLane	×	×	×	38.48%	33227(47.80%)	192
	✓	×	×	44.06%	33632(48.38%)	206(+14)
	×	✓	×	49.53%	37910(54.54%)	454(+262)
	✓	✓	×	51.34%	38287(54.86%)	476(+284)
	✓	✓	✓	51.34%	38330(55.14%)	587(+395)
Cirq	×	×	×	41.37%	17222(54.20%)	165
	✓	×	×	48.19%	17427(54.85%)	179(+14)
	×	✓	×	50.23%	22154(69.72%)	251(+86)
	✓	✓	×	53.08%	22540(70.94%)	283(+118)
	✓	✓	✓	53.08%	22683(71.39%)	344(+179)

expanding the test frontier into previously unreachable regions of the codebases.

Answer to RQ1: KQFuzz achieves the best results on all three libraries: 63.31% coverage on Qiskit, 58.71% on PennyLane, and 73.79% on Cirq, all above the strongest baselines, triggering 2.1× more unique crashes and adding 15.0% unique coverage on average.

6.2 Component Contributions and Cross-Model Robustness

To address RQ2, we first examine how the core components of KQFuzz contribute to its effectiveness and then evaluate whether its advantage over Fuzz4All persists across different LLM backends and model scales.

Ablation study of core components. We form an ablation over Seed Selection and Codebase Knowledge, while the final configuration measures the incremental benefit of adding Mutation to Seed+Repo. Each configuration is evaluated using 6,000 generated test seeds. Table 3 reports validity, line coverage, and unique crashes for each configuration.

Codebase Knowledge is the primary contributor to effectiveness. Compared with the base configuration, Repo-only improves validity rate by 8.86–16.70 percentage points (pp), coverage by 6.74–15.52 pp, and unique crashes by 71–262 across the three libraries. For example, on Cirq, it increases coverage from 54.20% to 69.72% and validity from 41.37% to 50.23%. These results show that library-specific API and constraint information substantially improve the generation of valid and coverage-effective tests.

Seed Selection and Mutation provide complementary benefits. Compared with the base configuration, Seed-only improves validity by 5.58–6.97 pp, while producing smaller coverage and crash gains. Adding Mutation to Seed+Repo improves coverage by only 0.28–0.47 pp, but discovers 57, 111, and 61 additional unique crashes on Qiskit, PennyLane, and Cirq, respectively. Thus, Seed Selection primarily improves seed quality, whereas Mutation mainly expands crash-triggering exploration. Mutation is evaluated on top of Seed+Repo because it is designed to refine selected, codebase-aware seeds rather than operate as an independent generator.

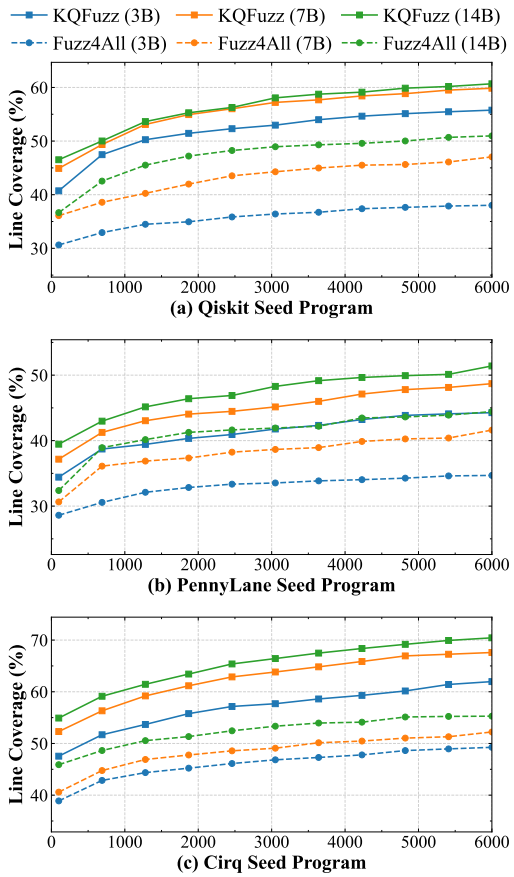

Figure 7: Line coverage comparison.

Table 4: Comparative analysis of validity and coverage for KQFuzz and Fuzz4All across various LLMs.

Model	Method	Val. Rate (%) ↑	Total Coverage (Unique)
Qwen2.5-Coder-3B	Fuzz4All	21.04	57226 (4162)
	KQFuzz	**33.16** (+12.12)	73079 (20015)
Qwen2.5-Coder-7B	Fuzz4All	25.51	66866 (6004)
	KQFuzz	**40.64** (+15.13)	80203 (19341)
Qwen2.5-Coder-14B	Fuzz4All	28.65	71741 (7131)
	KQFuzz	**52.99** (+24.34)	82317 (17707)
CodeLlama-7B	Fuzz4All	14.42	53334 (1968)
	KQFuzz	**23.90** (+9.48)	70893 (19527)
CodeLlama-13B	Fuzz4All	9.66	57300 (3104)
	KQFuzz	**31.04** (+21.38)	73364 (19168)

Robustness across LLM backends and scales. To further verify the robustness of our framework, we evaluated KQFuzz across two prominent model families (detailed results for CodeLlama are available in SM D), and compared them against the baseline. As illustrated in Figure 7, KQFuzz consistently outperforms the baseline across all tested model scales and library targets. We observe a striking performance crossover in Qiskit and Cirq, where KQFuzz utilizing the smallest model scale achieves significantly higher code coverage than the baseline framework at its largest scale. This massive performance gap indicates that the effectiveness of KQFuzz is primarily driven by our specialized, library-aware generation strategy rather than the mere scaling of model parameters.

Table 5: Bugs found by KQFuzz

Target	ID	API Location	Type	Status
Qiskit	#15550	synthesis	BV	Fixed
	#4159	synthesis	SV	Confirmed
	#15657	circuit	SD	Fixed
	#15665	synthesis	BV	Fixed
	#15666	circuit.library	BV	Fixed
	#15780	synthesis	BV	Fixed
PennyLane	#8931	io.qasm_interpreter	SD	Fixed
	#8726	decomposition	SV	Fixed
	#9140	math	SD	Fixed
	#9141	transforms	SV	Fixed
Cirq	#7934	contrib.paulistring	SD	Fixed
	#7939	transformers	SV	Fixed
	#7941	neutral_atoms	SD	Fixed

While both methods tend to benefit from increased model capacity, KQFuzz exhibits higher scaling efficiency. Table 4 shows that KQFuzz maintains a higher validity rate, with absolute improvements over the baseline ranging from 9.48 to 24.34 pp. Notably, the advantage of KQFuzz often becomes more pronounced as the model size increases, with the validity improvement peaking at 24.34% on the 14B scale. This suggests that our framework can better unlock the reasoning potential of larger models. As shown by the growth curves in Figure 7, Fuzz4All coverage is frequently constrained by a high rate of execution exceptions. In contrast, KQFuzz maintains a higher validity baseline, which allows larger models to focus more on handling complex API constraints rather than basic syntax, thereby achieving higher final coverage. These results demonstrate that KQFuzz is a robust, model-agnostic framework that can effectively leverage the latent capabilities of different model families and scales. The corresponding per-sample token consumption and generation time of KQFuzz and Fuzz4All across these model configurations are reported in SM E.2.

Answer to RQ2: KQFuzz's effectiveness is driven by its core components, primarily Codebase Knowledge, which boosts coverage by up to 16.1% (*e.g.*, on Cirq). Furthermore, KQFuzz exhibits superior scaling efficiency across LLMs, improving validity rates by 9.48% to 24.34% and achieving significantly faster coverage convergence.

6.3 Bug Detection

Regarding RQ3, as of the submission date, KQFuzz has identified 13 bugs, with 13 confirmed by developers and 10 already fixed. Table 5 summarizes the identified bugs. We categorize these bugs into three distinct types based on their root causes: (i) **Boundary violations (BV).** These errors occur when the library fails to handle extreme input dimensions or edge-case configurations. Such bugs typically bypass high-level sanity checks and manifest as low-level system panics, memory exhaustion, or process hangs. (ii) **State divergence (SD).** This category encompasses defects rooted in flawed internal implementations, organizational structures, or data processing logic. This type of bug usually emerges during multi-stage transformations, where internal metadata fails to maintain synchronized states. (iii) **Semantic violations (SV).** These defects refer to implementation behaviors that deviate from documented specifications or logical protocols. This includes incorrect gate decomposition logic, inconsistencies between documentation and implementation, or violations of API invariants.

Listing 1 illustrates a bug case where assigning parameters to a circuit containing `ForLoopOp` leads to a state divergence, resulting in an inconsistency error. This defect arises from redundant tracking: the loop parameter is incorrectly registered in the outer circuit's parameter table via both the control-flow operator slot and the nested circuit body. Such dual-source registration creates a synchronization gap between the global and local parameter scopes, causing the assignment mechanism to fail. To understand the bug, we quote the maintainer's response to our bug report.

> *Huh, this seems like it should really have been an error on all Qiskit versions, and the error shouldn't get as far as the internal logic error.*

Their response confirms that this issue has silently existed across all Qiskit releases and can only be triggered through the specific parameter-reuse pattern generated by our fuzzer, representing a scenario unlikely to be covered by existing fuzzers. Additional case studies illustrating other bug categories are provided in SM D.

```python
from qiskit.circuit import QuantumCircuit, Parameter
from qiskit.circuit.controlflow import ForLoopOp

theta = Parameter('theta')
body = QuantumCircuit(1)
body.rx(theta, 0)
for_loop_op = ForLoopOp(range(3), theta, body)
qc = QuantumCircuit(1)
qc.append(for_loop_op, [0])
# This line triggers the error
qc.assign_parameters({theta: 3.14159 / 2}, inplace=True)
```

Listing 1: Nested Parameter Inconsistency in Qiskit

Answer to RQ3: KQFuzz successfully identified 13 bugs across major quantum libraries, with 12 already fixed by developers. By exposing critical boundary, state, and semantic violations that elude conventional testing, our tool demonstrates exceptional capability in ensuring the functional reliability of the evolving quantum libraries.

7 Threats to Validity

There are some threats to the validity of our results and the conclusions drawn from them. First, the nondeterministic nature of LLMs could introduce variability in our metrics. We mitigate this threat to validity by running each fuzzer for 24 hours to reduce measurement variability. Second, our crash oracle relies on heuristic exception filtering and deduplication rules to eliminate false positive patterns. This presents a threat because such mechanisms might inadvertently discard genuine bugs that share error signatures with trivial failures, leading to unreported bugs. Finally, our evaluation focuses on three quantum libraries and specific open source models, and performance might differ on other paradigms. We believe our framework can generalize to other platforms that provide sufficient source code transparency for knowledge extraction.

8 Related Work

Quantum software engineering. The rapid emergence of quantum computing has driven the need for robust software engineering practices across the entire development lifecycle [2, 4, 19, 21, 26, 32, 37–41, 53, 57, 60, 65, 66]. Recent advancements have pushed the boundaries of quantum software development, introducing LLM-assisted quantum code synthesis (*e.g.*, [20]), and specialized compiler optimizations [50]. Alongside these development tools, quality assurance mechanisms are actively being established to ensure platform reliability. Empirical studies [42] have characterized unique bug patterns in quantum software, motivating the creation of static analysis tools like Qchecker [67] to detect issues in quantum programs prior to execution. Program-level techniques such as Muskit and QuanFuzz respectively apply syntactically valid mutations and generate test inputs for quantum programs [36, 51]. In contrast, QDiff, MorphQ, and FuzzQ target quantum platforms or libraries and use quantum programs as test inputs [28, 43, 52]. QEMI adapts EMI to quantum software-stack testing by removing dead code constructed from quantum control-flow patterns and checking the resulting variants for crash and output-distribution inconsistencies [34]. KQFuzz instead uses source-derived API knowledge to guide program generation toward broad quantum-library API exploration.

Testing compilers and other developer tools. The critical role of complex software infrastructure has motivated extensive research into automated tool testing. Compiler testing frequently employs randomized code generation and equivalence modulo inputs to uncover optimization bugs. Recently, this scope has expanded to lower-level assemblers via error-driven grammar inference [27]. Beyond foundational compilers, specialized techniques target diverse input spaces: WebAssembly engines utilize stack-invariant transformations for semantic-preserving mutation [63], network protocols employ constrained query-response fuzzing to uncover stateful bugs [64], and database management systems leverage dynamic data-dependency analysis for generic query synthesis [59]. Furthermore, LLMs are revolutionizing the field by acting as universal, language-agnostic fuzzers [56] and automating the generation of complex API fuzz drivers [62].

9 Conclusion

This paper presents KQFuzz, a knowledge-guided fuzzing framework for quantum libraries that leverages LLMs to enhance their performance and efficiency. By leveraging a multi-dimensional API knowledge corpus to guide LLM-based seed generation, coupled with fitness-driven seed selection and two-level mutations, KQFuzz successfully overcomes the limitations of insufficient flexibility in traditional circuit-level fuzzers and the low validity rates of generic LLM-based approaches. Experimental evaluations on major quantum libraries, including Qiskit, PennyLane, and Cirq, demonstrate that KQFuzz significantly outperforms state-of-the-art baselines in both code coverage and quantum program validity. Furthermore, the discovery of 13 bugs underscores the framework's practical effectiveness in uncovering critical bugs within the rapidly evolving quantum computing ecosystem.

10 Data Availability

The source code and data involved in our study are publicly available in the Zenodo repository [3].

References

[1] Amira Abbas, Andris Ambainis, Brandon Augustino, Andreas Bärtschi, Harry Buhrman, Carleton Coffrin, Giorgio Cortiana, Vedran Dunjko, Daniel J Egger, Bruce G Elmegreen, et al. 2024. Challenges and opportunities in quantum optimization. *Nature Reviews Physics* 6, 12 (2024), 718–735.

[2] Shaukat Ali, Tao Yue, and Rui Abreu. 2022. When software engineering meets quantum computing. *Commun. ACM* 65, 4 (2022), 84–88.

[3] Anonymous Authors. 2026. Replication package for anonymous submission. https://doi.org/10.5281/zenodo.19230459

[4] Paolo Arcaini, Andriy Miranskyy, and Hausi Müller. 2025. Introduction to the Special Section on software engineering for hybrid quantum computing systems. , 112362 pages.

[5] Nicola Assolini, Alessandra Di Pierro, and Isabella Mastroeni. 2024. Static analysis of quantum programs. In *International Static Analysis Symposium*. Springer, 1–25.

[6] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M Sohaib Alam, Guillermo Alonso-Linaje, Bharath Akash-Narayanan, Ali Asadi, et al. 2018. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968* (2018).

[7] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 678–689.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.

[9] Marco Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, et al. 2021. Variational quantum algorithms. *Nature Reviews Physics* 3, 9 (2021), 625–644.

[10] Cirq Developers. 2025. *Cirq*. https://doi.org/10.5281/zenodo.4062499

[11] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[12] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*. 1–13.

[13] Ruiqi Dong, Fanke Tong, He Huang, Xiaogang Zhu, Xi Xiao, Shaohua Wang, Sheng Wen, and Yang Xiang. 2025. One Mutation Fits All: Exploring Universal Library Fuzzing based on Exogenous Mutation. *IEEE Transactions on Dependable and Secure Computing* (2025).

[14] Wang Fang and Mingsheng Ying. 2024. Symbolic execution for quantum error correction programs. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1040–1065.

[15] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*.

[16] Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. 2012. Surface codes: Towards practical large-scale quantum computation. *Physical Review A—Atomic, Molecular, and Optical Physics* 86, 3 (2012), 032324.

[17] Sukhpal Singh Gill, Oktay Cetinkaya, Stefano Marrone, Daniel Claudino, David Haunschild, Leon Schlote, Huaming Wu, Carlo Ottaviani, Xiaoyuan Liu, Sree Pragna Machupalli, et al. 2025. Quantum computing: Vision and challenges. In *Quantum computing*. Elsevier, 19–42.

[18] Sukhpal Singh Gill, Adarsh Kumar, Harvinder Singh, Manmeet Singh, Kamalpreet Kaur, Muhammad Usman, and Rajkumar Buyya. 2022. Quantum computing: A taxonomy, systematic review and future directions. *Software: Practice and Experience* 52, 1 (2022), 66–114.

[19] Xiaoyu Guo, Shinobu Saito, and Jianjun Zhao. 2025. M2QCode: A Model-Driven Framework for Generating Multi-Platform Quantum Programs. *arXiv preprint arXiv:2510.17110* (2025).

[20] Xiaoyu Guo, Minggu Wang, and Jianjun Zhao. 2025. QuanBench: Benchmarking Quantum Code Generation with Large Language Models. *arXiv preprint arXiv:2510.16779* (2025).

[21] Xiaoyu Guo, Jianjun Zhao, and Pengzhan Zhao. 2024. On repairing quantum programs using ChatGPT. In *Proceedings of the 5th ACM/IEEE International Workshop on Quantum Software Engineering*. 9–16.

[22] Tianmin Hu, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Huanting Wang, Meng Li, and Zheng Wang. 2024. Upbeat: Test input checks of q# quantum libraries.

In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 186–198.

[23] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[24] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D Nation, Lev S Bishop, Andrew W Cross, et al. 2024. Quantum computing with Qiskit. *arXiv preprint arXiv:2405.08810* (2024).

[25] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, et al. 2024. When fuzzing meets llms: Challenges and opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 492–496.

[26] Tiancheng Jin, Shangzhou Xia, and Jianjun Zhao. 2025. NovaQ: Improving Quantum Program Testing through Diversity-Guided Test Case Generation. *arXiv preprint arXiv:2509.04763* (2025).

[27] Hyungseok Kim, Soomin Kim, Jungwoo Lee, and Sang Kil Cha. 2024. AsFuzzer: Differential testing of assemblers with error-driven grammar inference. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1099–1111.

[28] Vasileios Klimis, Avner Bensoussan, Elena Chachkarova, Karine Even-Mendoza, Sophie Fortz, and Connor Lenihan. 2025. Shaking Up Quantum Simulators with Fuzzing and Rigour. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 1400–1428.

[29] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[30] Yuechen Li, Minqi Shao, Jianjun Zhao, and Qichen Wang. 2026. A Methodological Analysis of Empirical Studies in Quantum Software Testing. *arXiv preprint arXiv:2601.08367* (2026).

[31] Xingshuang Lin, Qinge Xie, Binbin Zhao, Yuan Tian, Saman Zonouz, Na Ruan, Jiliang Li, Raheem Beyah, and Shouling Ji. 2025. PROMFUZZ: Leveraging LLM-Driven and Bug-Oriented Composite Analysis for Detecting Functional Bugs in Smart Contracts. *arXiv preprint arXiv:2503.23718* (2025).

[32] Peixun Long and Jianjun Zhao. 2024. Equivalence, identity, and unitarity checking in black-box testing of quantum programs. *Journal of Systems and Software* 211 (2024), 112000.

[33] Peixun Long and Jianjun Zhao. 2024. Testing multi-subroutine quantum programs: From unit testing to integration testing. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), 1–61.

[34] Junjie Luo, Shangzhou Xia, Fuyuan Zhang, and Jianjun Zhao. 2026. QEMI: A Quantum Software Stacks Testing Framework via Equivalence Modulo Inputs. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 149–169.

[35] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[36] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. 2021. Muskit: A mutation analysis tool for quantum software testing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1266–1270.

[37] Eñaut Mendiluze Usandizaga, Shaukat Ali, Tao Yue, and Paolo Arcaini. 2025. Quantum circuit mutants: Empirical analysis and recommendations. *Empirical Software Engineering* 30, 4 (2025), 100.

[38] Asmar Muqeet, Shaukat Ali, Tao Yue, and Paolo Arcaini. 2024. A machine learning-based error mitigation approach for reliable software development on IBM's quantum computers. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 80–91.

[39] Asmar Muqeet, Tao Yue, Shaukat Ali, and Paolo Arcaini. 2024. Mitigating noise in quantum software testing using machine learning. *IEEE Transactions on Software Engineering* 50, 11 (2024), 2947–2961.

[40] Juan Manuel Murillo, Jose Garcia-Alonso, Enrique Moguel, Johanna Barzen, Frank Leymann, Shaukat Ali, Tao Yue, Paolo Arcaini, Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, et al. 2025. Quantum software engineering: Roadmap and challenges ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–48.

[41] Noah H Oldfield, Christoph Laaber, Tao Yue, and Shaukat Ali. 2025. Faster and better quantum software testing through specification reduction and projective measurements. *ACM Transactions on Software Engineering and Methodology* 34, 7 (2025), 1–39.

[42] Matteo Paltenghi and Michael Pradel. 2022. Bugs in quantum computing platforms: an empirical study. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27.

[43] Matteo Paltenghi and Michael Pradel. 2023. MorphQ: Metamorphic testing of the Qiskit quantum computing platform. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2413–2424.

[44] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[45] Md Shafiuzzaman, Achintya Desai, Laboni Sarker, and Tevfik Bultan. 2024. STASE: Static analysis guided symbolic execution for UEFI vulnerability signature generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1783–1794.

[46] Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, Prashast Srivastava, and Suman Jana. 2024. Fox: Coverage-guided fuzzing as online stochastic control. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 765–779.

[47] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.

[48] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[49] Krishna Upadhyay, Moshood Fakorede, and Umar Farooq. 2026. Understanding Bugs in Quantum Simulators: An Empirical Study. arXiv:2603.22789 [quant-ph] https://arxiv.org/abs/2603.22789

[50] Hanrui Wang, Daniel Bochen Tan, Pengyu Liu, Yilian Liu, Jiaqi Gu, Jason Cong, and Song Han. 2024. Q-pilot: Field programmable qubit array compilation with flying ancillas. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[51] 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)*. IEEE, 466–469.

[52] 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)*. IEEE, 692–704.

[53] Xinyi Wang, Shaukat Ali, Tao Yue, and Paolo Arcaini. 2024. Quantum approximate optimization algorithm for test case optimization. *IEEE Transactions on Software Engineering* 50, 12 (2024), 3249–3264.

[54] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[55] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*. 1634–1645.

[56] 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*. 1–13.

[57] Shangzhou Xia, Jianjun Zhao, Fuyuan Zhang, and Xiaoyu Guo. 2025. Quantum concolic testing. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1146–1166.

[58] Yuchong Xie, Wenhui Zhang, and Dongdong She. 2025. ZTaint-Havoc: From Havoc mode to zero-execution fuzzing-driven taint inference. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 917–939.

[59] Yupeng Yang, Yongheng Chen, Rui Zhong, Jizhou Chen, and Wenke Lee. 2024. Towards generic database management system fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 901–918.

[60] Jiaming Ye, Xiongfei Wu, Shangzhou Xia, Fuyuan Zhang, and Jianjun Zhao. 2025. Is Measurement Enough? Rethinking Output Validation in Quantum Program Testing. *arXiv preprint arXiv:2509.16595* (2025).

[61] Mingsheng Ying, Li Zhou, and Gilles Barthe. 2025. Laws of Quantum Programming. *ACM Transactions on Software Engineering and Methodology* (2025).

[62] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How effective are they? exploring large language model based fuzz driver generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1223–1235.

[63] Lingming Zhang, Binbin Zhao, Jiacheng Xu, Peiyu Liu, Qinge Xie, Yuan Tian, Jianhai Chen, and Shouling Ji. 2025. Waltzz:{WebAssembly} Runtime Fuzzing with {Stack-Invariant} Transformation. In *34th USENIX Security Symposium (USENIX Security 25)*. 6159–6178.

[64] Qifan Zhang, Xuesong Bai, Xiang Li, Haixin Duan, Qi Li, and Zhou Li. 2024. {ResolverFuzz}: Automated Discovery of {DNS} Resolver Vulnerabilities with {Query-Response} Fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4729–4746.

[65] Jianjun Zhao. 2020. Quantum software engineering: Landscapes and horizons. *arXiv preprint arXiv:2007.07047* (2020).

[66] 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 (2023), 111805.
[67] Pengzhan Zhao, Xiongfei Wu, Zhuo Li, and Jianjun Zhao. 2023. Qchecker: Detecting bugs in quantum programs via static analysis. In *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*. IEEE, 50–57.
[68] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.

This supplementary material (**SM**) provides necessary details omitted from the main text, including quantum computing background (SM A), API association modeling definitions (SM B), implementation and experimental setup of KQFuzz (SM C), additional evaluation results with bug case studies (SM D), and efficiency analysis (SM E).

A Background of quantum computing

This section provides a more comprehensive overview of the fundamental concepts in quantum computing referenced in the main text.

Qubits and quantum states. The fundamental unit of quantum information is the quantum bit, or qubit. Unlike a classical bit, which must reside in a discrete state of either 0 or 1, a qubit can exist in a linear combination, or *superposition*, of these states. Mathematically, a single-qubit state is represented as a vector $|\psi\rangle$ in a two-dimensional complex Hilbert space $\mathcal{H} \cong \mathbb{C}^2$. Using Dirac notation, an arbitrary single-qubit state is expressed as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where $|0\rangle$ and $|1\rangle$ form the standard computational basis, and $\alpha, \beta \in \mathbb{C}$ are complex probability amplitudes satisfying the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. For an n-qubit system, the state space grows exponentially to 2^n dimensions, represented by the tensor product of the individual qubit Hilbert spaces ($\mathcal{H}^{\otimes n}$). Within this expanded space, *entanglement* manifests as multi-qubit states that cannot be factored into the tensor product of individual qubit states, representing non-local correlations unique to quantum mechanics.

Quantum gates. The evolution of a closed quantum system is governed by unitary transformations. In the quantum circuit model, these transformations are enacted by quantum gates, which are represented by unitary matrices U satisfying $U^\dagger U = UU^\dagger = I$. Single-qubit gates manipulate the state of individual qubits. Prominent examples include the Pauli matrices (X, Y, Z), which induce generalized rotations around the Bloch sphere, and the Hadamard gate (H), which creates an equal superposition from a computational basis state:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

To generate entanglement and enable universal quantum computation, multi-qubit gates are required. A ubiquitous two-qubit entangling operation is the Controlled-NOT (CNOT) gate, which flips the state of a target qubit if and only if the control qubit is in the $|1\rangle$ state. Table 6 summarizes the symbols and mathematical matrix representations of these typical quantum gates. By composing finite sets of single-qubit and entangling two-qubit gates, any arbitrary unitary operation can be approximated to a desired precision.

Measurement. To extract classical information from a quantum system, a measurement operation must be applied. According to the postulates of quantum mechanics, measuring a qubit in the computational basis $\{|0\rangle, |1\rangle\}$ forces its superposition state to collapse into one of the definite basis states. This outcome is inherently probabilistic; governed by the Born rule, the probability of observing a specific state is given by the squared magnitude of its corresponding amplitude. For instance, measuring $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ yields the outcome 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$. Following measurement, the quantum state irreversibly collapses into the observed state, destroying any prior superposition or entanglement.

Table 6: Summary of typical quantum gates and their matrix representations.

Gate Name	Symbol	Matrix
Hadamard	H	$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$
Pauli-X (NOT)	X	$\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$
Pauli-Y	Y	$\begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}$
Pauli-Z	Z	$\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$
Controlled-NOT	CNOT	$\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}$

Quantum circuits. A quantum algorithm is practically formulated as a quantum circuit—an ordered, acyclic sequence of operations applied to an initial state, typically initialized to $|0\rangle^{\otimes n}$. Read from left to right, a circuit comprises state preparation, a series of unitary quantum gates to execute the logical computation, and terminal measurements to map the final quantum state into classical registers. Due to the probabilistic nature of quantum measurement, a circuit is typically executed repeatedly to sample the output distribution, thereby allowing the estimation of expectation values that encode the solution to the given computational problem.

B Details of API Association Modeling

This section elaborates on the mathematical formulations for the API association model referenced in § 4.1 of the main text. To rigorously quantify the semantic and structural relationships between APIs, we formally define the three dimensions used to compute the combined association score.

Proximity. We use file- and module-level locality as a proxy for semantic relatedness. Let $S_p(a, b)$ denote the *proximity score* between APIs a and b, defined as:

$$S_p(a,b) = \begin{cases} \alpha, & \text{if } a \text{ and } b \text{ are defined in the same file,} \\ \beta, & \text{if } a \text{ and } b \text{ are in the same module but different files,} \\ \lambda, & \text{otherwise.} \end{cases}$$

Type overlap. Quantum software typically introduces numerous domain-specific types. APIs that share input or output types are likely to be used together. Let $\text{Typeset}(a)$ denote the set of non-native types appearing in the signature of API a. We quantify type-level similarity using the Jaccard index:

$$S_t(a,b) = J(\text{Typeset}(a), \text{Typeset}(b)),$$

where $J(A, B) = |A \cap B|/|A \cup B|$ measures the overlap between two sets.

Call relationships. We approximate API relatedness based on call relationships observed in the source code via *reachability* and *commonality*. Reachability is measured using a shortest call distance function $D(a, b)$, where $D(a, b) = 1$ indicates a direct call and $D(a, b) = +\infty$ indicates no call path. Commonality captures shared invocation patterns. Let $\text{Caller}(\cdot)$ and $\text{Callee}(\cdot)$ denote the sets of

You are a code analysis assistant. Analyze the source code of the function in quantum library **[<TARGET_LIB>]** and provide the result ONLY in the following JSON format:
{
"Input Constraints": ["constraint 1", "constraint 2", …],
"Output Description": "A brief description of the return value",
"Functionality Summary": "A concise summary of the function",
"Fuzzing Test Points": ["test point 1", "test point 2", …]
}
Here is the API source code: **[<SRC_CODE>]**

Figure 8: Prompt structure used by KQFuzz for API implementation semantic modeling.

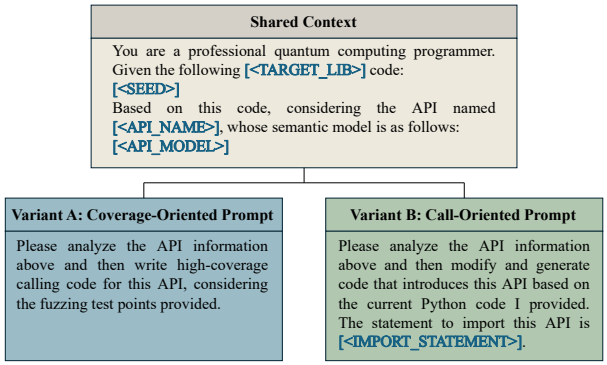

Figure 9: A structured view of the prompt template used for LLM-guided seed program generation. The template consists of a shared context block (seed program and target API information) and two parallel, specialized variant blocks: coverage-oriented prompt and call-oriented prompt.

APIs that invoke or are invoked by a given API, respectively. The resulting *call association score* is defined as:

$$S_c(a,b) = \frac{1}{2}\left[J(\text{Caller}(a),\text{Caller}(b)) + J(\text{Callee}(a),\text{Callee}(b))\right] + \frac{1}{2D(a,b)}.$$

C Implementation details of KQFuzz

This section elaborates on the implementation details and the experimental setup of KQFuzz omitted in § 4. Table 7 provides a comprehensive summary of the core hyperparameters and configuration settings used throughout the evaluation of KQFuzz. Subsequently, we describe the exact prompt structures designed for KQFuzz and define the primary metrics utilized to evaluate performance.

Table 7: Summary of Core Hyperparameters and Configuration Settings.

Section	Parameter	Symbol	Value
§ 4.1	Intra-file Score	α	1.0
	Intra-module Score	β	0.6
	Default Proximity Score	λ	0
	Evolution Sensitivity	γ	4
	Evolution Range	δ	0.3
§ 4.1	Locality Weight	w_1	2
	Type Overlap Weight	w_2	1
	Call Relationship Weight	w_3	2
§ 4.3	Entanglement Cap	τ	6
	API Diversity Weight	η	3
§ 4.3	Selection Top-κ	κ	10
	Programs per Iteration	-	30

C.1 Prompt

In this section, we detail the specific prompt structures utilized by KQFuzz, corresponding to the two main phases of our framework: API semantic extraction and seed program generation.

API implementation semantic modeling prompt. As illustrated in Figure 8, to isolate the exact logic of the target library, the prompt injects the library name (<TARGET_LIB>) and the relevant, filtered source code (<SRC_CODE>). To ensure the output is easily parsed by the automated testing pipeline, we constrain the LLM to return a strictly formatted JSON object. This JSON explicitly constructs the semantic model by extracting:

- *Input Constraints:* rules, types, and boundary conditions for the API's arguments;
- *Output Description:* expected return values and behaviors;
- *Functionality Summary:* a concise overview of the API's purpose; and
- *Fuzzing Test Points:* specific edge cases, parameter combinations, and structural targets to guide the fuzzing phase.

LLM-guided seed generation prompt. Figure 9 displays the prompt template used by the *fuzzing LLM*. To maintain contextual awareness and ensure the generation of valid seed programs, the prompt is divided into a **Shared Context** block and two task-specific variants:

- *Shared Context:* this foundational block establishes the persona of a professional quantum computing programmer. It grounds the generation process by providing the execution skeleton via the seed program (<SEED>), alongside the target library (<TARGET_LIB>), the target API name (<API_NAME>), and its comprehensive semantic model (<API_MODEL>) extracted during the modeling phase;
- *Variant A (Coverage-Oriented Prompt):* activated when the target API (i_t) is already present in the seed program. It directs the model to analyze the semantic profile and generate high-coverage calling code, specifically instructing the LLM to leverage the provided fuzzing test points to explore deeper, more diverse execution paths; and
- *Variant B (Call-Oriented Prompt):* activated when i_t is absent from the seed program. It instructs the model to modify the existing Python seed to seamlessly introduce a new invocation of the target API. To guarantee syntactic correctness and a fully runnable output, this variant explicitly injects the necessary import statement (<IMPORT_STATEMENT>).

C.2 Metric

We adopt three primary metrics in software testing to analyze the collected experimental results. For clarity, their key concepts and functionalities are introduced below.

Code coverage. Code coverage has been widely adopted in software testing and quantum library testing. We measure Python line coverage using the `coverage.py` tool while excluding the library's internal test files to ensure that the metrics accurately reflect the exploration of core functional logic. Furthermore, we evaluate **unique code coverage**, defined as the specific code segments triggered exclusively by a particular configuration. This metric allows

us to quantify the distinct exploratory effectiveness of each setting and its unique contribution to the overall code space exploration.

Validity. A generated case is defined as valid if it executes without runtime exceptions in a properly configured environment and invokes the target API at least once. We perform deduplication to count only unique instances, which further provides metrics such as the number of valid programs and validity rate. Note that because the validity of mutated samples inherently depends on the specific mutation strategy applied, mutants are excluded from the statistics for this metric.

Bug detection. Following prior work on quantum library fuzzing, we report the number of unique detected bugs.

C.3 Baseline

To evaluate the effectiveness of KQFuzz, we compared it against three state-of-the-art baselines, including two quantum-specific fuzzers and one LLM-based universal fuzzer:

- **MorphQ** [43] is a metamorphic testing framework specifically developed for the Qiskit platform, which designs quantum-specific metamorphic relations and a dedicated program generator to expose semantic inconsistencies.
- **FuzzQ** [28] encodes QASM semantics in Alloy to generate structurally constrained circuits and employs invariant checking, statistical tests, and cross-simulator unitary consistency as differential oracles.
- **Fuzz4All** [56] is a general-purpose fuzzer that proposes an LLM-based auto-prompting mechanism combined with an iterative fuzzing loop to automatically synthesize diverse and semantically meaningful inputs across programming languages, achieving high coverage and broad applicability beyond quantum systems.

D Experimental results

This section provides supplementary experimental data and qualitative examples that complement the primary evaluation presented in § 6 of the main text. Specifically, § D.1 presents additional code coverage results utilizing the CodeLlama model family to reinforce our primary coverage findings. Furthermore, as referenced in the main text, § D.2 details additional case studies illustrating other bug categories discovered by KQFuzz.

D.1 Coverage Results

As illustrated in Figure 10, the evaluation utilizing the CodeLlama model family (7B and 13B) corroborates the findings presented in the main text. KQFuzz consistently outperforms the Fuzz4All baseline across all three quantum libraries. Most notably, KQFuzz operating at the smaller 7B scale consistently achieves significantly higher line coverage than the baseline framework utilizing the larger 13B model throughout the entire generation process. These supplementary results further validate that the superiority of KQFuzz is fundamentally driven by its meticulously designed, library-aware generation strategy rather than the sheer scaling of model parameters.

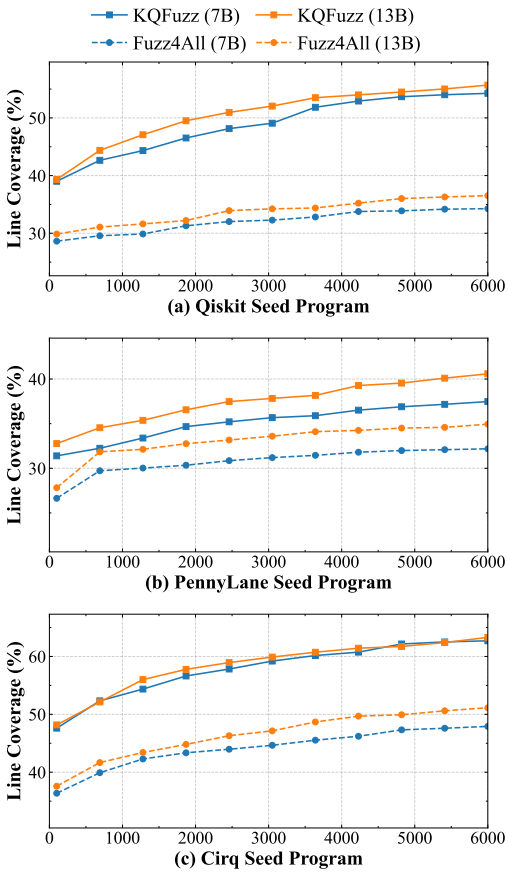

Figure 10: Line coverage comparison using CodeLlama.

D.2 Case Study

In the following, we showcase several representative bugs found by KQFuzz to illustrate their impact and root causes.

Listing 2 shows a boundary violation where synthesizing a 1-qubit Clifford operator triggers a Rust-level panic. Although the LNN (Linear Nearest Neighbor) synthesis should support any valid circuit dimension, the underlying logic failed to handle the $n = 1$ edge cases, assuming multi-qubit connectivity. It is confirmed as an unhandled boundary in the synthesis engine and has since been patched to support minimal qubit configurations.

```python
from qiskit.circuit import QuantumCircuit
from qiskit.quantum_info import Clifford
from qiskit.synthesis.clifford import synth_clifford_depth_lnn

qc = QuantumCircuit(1)
qc.h(0)
clifford_op = Clifford(qc)

# This call triggers the Rust panic
synthesized_circuit = synth_clifford_depth_lnn(clifford_op)
```

Listing 2: Single-qubit Clifford Synthesis Panic in Qiskit

Listing 3 illustrates a state divergence where checking gate compatibility triggers a `TypeError`. The callee function fails when processing a `UniformSuperpositionGate` because the gate class is unhashable. This bug stems from a flaw in the relevant `__contains__` implementation, which incorrectly assumes that all gate-derived objects are hashable.

```python
import cirq
from cirq.neutral_atoms import is_native_neutral_atom_gate
from cirq.ops import UniformSuperpositionGate

gate = UniformSuperpositionGate(m_value=3, num_qubits=2)

# This call triggers the TypeError
print(is_native_neutral_atom_gate(gate))
```

Listing 3: Unhashable Gate Membership in Cirq

Listing 4 illustrates a semantic violation where executing a QNode with the `sign_expand` transform results in an error. This defect stems from a packaging oversight where a critical metadata file (`sign_expand_data.json`) was omitted from the distribution configuration. This missing resource prevents the transformation API from loading its required logic, resulting in a runtime failure that violates the library's functional integrity and documented behavior.

```python
import pennylane as qml
dev = qml.device("default.qubit", wires=2)
obs = qml.Hamiltonian([2.0, -1.5], [qml.PauliZ(0), qml.PauliX(1
    )])

@qml.transforms.sign_expand
@qml.qnode(dev)
def circuit():
    qml.RX(0.5, wires=0)
    qml.RY(0.5, wires=1)
    qml.CNOT(wires=[0, 1])
    return qml.expval(obs)
# A FileNotFoundError error occurs during circuit execution.
result = circuit()
```

Listing 4: Missing Resource Dependency in PennyLane

E Efficiency Analysis

This section analyzes the computational efficiency of KQFuzz. First, we examine the one-time modeling overhead and the execution-time distribution of the complete pipeline under the default configuration used in RQ1. Second, we report the per-sample token consumption and generation time across the LLM backends evaluated in RQ2.

E.1 Overall Computational Efficiency

Cost of the modeling LLM. The modeling LLM is invoked exclusively during the one-time API corpus construction phase, in which the source code of each API is processed once to construct its semantic model. The resulting corpus is reused throughout all subsequent fuzzing iterations without any further invocation of the modeling LLM. As reported in Table 1 of the main text, this phase consumes 0.38M, 0.61M, and 0.41M tokens for Qiskit, PennyLane, and Cirq, respectively. Corpus construction takes only 2.3, 3.3, and 1.6 minutes for the three libraries, corresponding to 0.15%, 0.23%, and 0.11% of the fixed 24-hour budget. Therefore, the one-time preprocessing overhead is negligible in all three cases.

Time breakdown across the pipeline. To examine where computational time is spent, we divide the complete KQFuzz pipeline into four stages: Preparation, Seed Generation, Mutation, and Execution. Preparation corresponds to the one-time API corpus construction. Seed Generation invokes the fuzzing LLM to produce seed programs. Mutation performs source-level transformations on existing programs, whereas Execution runs the generated programs against the target library.

Table 8: Execution-Time Distribution.

Target	Preparation	Seed Gen.	Mutation	Execution
Qiskit	0.15%	57.50%	0.13%	42.22%
PennyLane	0.23%	50.83%	0.15%	48.79%
Cirq	0.11%	55.97%	0.14%	43.78%

Table 9: Average Per-Sample Token Consumption and Generation Time.

Model	Method	Input Tokens	Output Tokens	Time (s)
Qwen-3B	Fuzz4All	332.15	201.43	0.92
	KQFuzz	367.10	300.90	1.32
Qwen-7B	Fuzz4All	365.40	245.56	1.44
	KQFuzz	329.38	257.33	1.52
Qwen-14B	Fuzz4All	323.67	193.33	1.89
	KQFuzz	385.60	344.00	3.35
CodeLlama-7B	Fuzz4All	518.21	318.73	1.48
	KQFuzz	479.17	431.50	2.07
CodeLlama-13B	Fuzz4All	526.95	319.67	2.47
	KQFuzz	521.84	479.50	3.85

As shown in Table 8, the computational cost is almost entirely concentrated in Seed Generation and Execution, which together account for more than 99% of the total execution time across all three libraries. In contrast, Preparation and Mutation each consume less than 0.23%. Preparation is negligible because corpus construction is performed only once and reused throughout all subsequent iterations, while Mutation is implemented as a lightweight source-level transformation without LLM invocation. This design allows KQFuzz to efficiently diversify LLM-generated seed programs under the fixed experimental budget.

E.2 Per-Sample Generation Cost

We further analyze the cost of individual LLM generation calls across the model families and scales evaluated in RQ2. For both KQFuzz and Fuzz4All, we report the average input tokens, output tokens, and generation time per sample. Each result is averaged over Qiskit, PennyLane, and Cirq.

As shown in Table 9, both methods incur similar input token consumption, while the clear and consistent difference lies in output tokens, where KQFuzz consumes more than Fuzz4All on every model. This is a direct consequence of KQFuzz's generation paradigm: rather than generating short and standalone programs, KQFuzz iteratively extends existing seed programs by introducing new and semantically related API calls, resulting in longer and structurally richer programs. This design is precisely what enables KQFuzz to achieve significantly higher coverage and validity, and the additional output tokens represent meaningful semantic content rather than redundancy.