White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of Leeds.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/211224/

Version: Accepted Version

Proceedings Paper:
Hu, T., Ye, G., Tang, Z. et al. (2024) UPBEAT: Test Input Checks of Q# Quantum Libraries. In: ISSTA 2024: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 16-20 Sep 2024, Vienna, Austria. ACM, New York, NY, pp. 186-198. ISBN: 979-8-4007-0612-7.

https://doi.org/10.1145/3650212.3652120

This item is protected by copyright. This is an author produced version of a conference paper accepted for publication in Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) 2024, made available under the terms of the Creative Commons Attribution License (CC-BY), which permits unrestricted use, distribution and reproduction in any medium, provided the original work is properly cited.

Reuse
This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here: https://creativecommons.org/licenses/

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UPBEAT: Test Input Checks of Q# Quantum Libraries

Tianmin Hu
Northwest University, China
hutianmin@stumail.nwu.edu.cn

Guixin Ye
Northwest University, China
gxye@nwu.edu.cn

Zhanyong Tang
Northwest University, China
zytang@nwu.edu.cn

Shin Hwei Tan
Concordia University, Canada
shinhwei.tan@concordia.ca

Huanting Wang
University of Leeds, United Kingdom
schwa@leeds.ac.uk

Meng Li
Hefei University of Technology, China
mengli@hfut.edu.cn

Zheng Wang
University of Leeds, United Kingdom
z.wang5@leeds.ac.uk

ABSTRACT

High-level programming models like Q# significantly simplify the complexity of programming for quantum computing. These models are supported by a set of foundation libraries for code development. However, errors can occur in the library implementation, and one common root cause is the lack of or incomplete checks on properties like values, length, and quantum states of inputs passed to user-facing subroutines. This paper presents Upbeat, a fuzzing tool to generate random test cases for bugs related to input-checking in Q# libraries. Upbeat develops an automated process to extract constraints from the API documentation and the developer-implemented input-checking statements. It leverages open-source Q# code samples to synthesize test programs. It frames the test case generation as a constraint satisfaction problem for classical computing and a quantum state model for quantum computing to produce carefully generated subroutine inputs to test if the input-checking mechanism is appropriately implemented. Under 100 hours of automated test runs, Upbeat has successfully identified 16 bugs in API implementations and 4 documentation errors. Of these, 14 have been confirmed, and 12 have been fixed by the library developers.

1 INTRODUCTION

The dawn of quantum computing has arrived. Like classical computers, quantum computers also require programming models, development libraries, and compilers for software development. These tools allow developers to express program logic and transform human-written code into executable instructions or quantum circuits to run on quantum machines or simulators.

Q# is a C#-like programming model for quantum computing [22, 43, 58]. It supports two types of subroutines (referred to as *callables* in Q#): *operations* and *functions*. The former is a routine that affects quantum states, while the latter does not alter the quantum state, being deterministic, and will always return the same result for the same input. A core component of Q# is its Development Kit Libraries, which provide common programming routines for classical mathematics, quantum error correction, and domain-specific algorithms like quantum machine learning. Embracing a high-level programming approach, Q# simplifies the expression of complex quantum logic and operations, making it an ideal tool for exploring novel algorithms for quantum computing.

Like many application developers in other domains, users of Q# libraries typically treat the library implementation as a black box and place their trust in the robustness of the library. However, like any large software, Q# libraries are susceptible to bugs. A common issue in Q# libraries is insufficient checks on the properties of a user input passed into a callable – a problem referred to as *boundary bugs* in this work. These input properties might include the numerical value range, array length, quantum bit (*qubit*) state, or other assumptions made by library developers or defined in the API specifications. A well-implemented library should generate meaningful error messages at the earliest point to notify the library user when their input does not meet the requirements, assisting application developers in debugging their user code.

Boundary bugs in quantum libraries can arise if a callable does not verify that the input qubit state matches the expected conditions, leading to potential runtime crashes or incorrect results [52]. Such errors or crashes might also occur if the implementation forgets to check for overflow in an input qubit during measurement or conversion before it is passed to a callable. These boundary errors are often elusive, only happen under specific inputs, and often surface much later when a user-facing callable is invoked. However, they pose a significant challenge in debugging user code, especially when the user input has participated in multiple operations where the error only occurs on a different variable directly or indirectly resulting from the user input. This is because when errors eventually emerge, they might not directly link back to the initial callable inputs, making it difficult to trace and resolve the problem.

The current efforts of quantum software testing have been focused on the application-level algorithms [24, 32, 33, 38, 47, 61, 63] or the compiler-based code translation for optimizing and generating quantum circuits [53, 62]. Although these efforts are important, one critical area currently lacking sufficient attention is testing boundary bugs of quantum libraries, a common issue for quantum libraries. Doing so requires generating valid and invalid values and properties for multiple arguments of a subroutine. While there is a large body of work for conventional computers in automating software testing [7, 17, 25, 27, 28, 51, 54, 67, 71, 75], these prior methods are not immediately transferable to quantum libraries. Furthermore, existing methods with qubit modeling [4, 61] are ineffective for testing boundary bugs because it is difficult to generate bug-exposing test inputs without knowing the assumptions that library developers made. Our work aims to fill this gap by taking Q# as a case study. Addressing this testing gap is essential for improving the usability and reliability of quantum libraries.

Achieving our goal requires managing the unique characteristics and requirements of quantum computing. In our case, these require checking input properties at the programming language levels. For instance, a qubit's state can be represented as a point on the surface of a sphere. In this context, describing such a state requires specifying its position along the sphere's `X`, `Y`, or `Z` axis and its measured outcome (`Zero` or `One`), which exhibits a probability that can fluctuate within a defined noise tolerance range.

We present Upbeat, the first automated and open-source framework for testing boundary bugs of quantum libraries. Upbeat generates test cases – executable Q# programs containing code and callable-specific input data – to exercise the input-checking mechanisms of the test target. To generate the test code, we first built a corpus containing Q# API call sequences from Q# examples extracted from the library implementation, open-source projects, and the API manual. From this corpus, Upbeat automatically assembles call sequences to construct test code through program synthesis.

To generate test data to be passed to a callable, Upbeat first automatically extracts, *offline*, the developer-implemented input checks from assertion-like Q# statements such as `Assert` operations and `Fact` functions from the source code of the test targets. It also leverages constraint specifications expressed in plain texts from the API document. We encode the input constraints of a qubit state as a Q# *quaternion* using a four-element tuple, {`axis`, `result`, `probability`, `tolerance`}. For instance, {`Z-axis, Zero, 1.0, 0.0`} signifies that the target qubit must be in the state $|0\rangle$.

Based on the extracted constraints, Upbeat generates both *valid* and *invalid* input for a test callable. Valid inputs comply with the *library developer-implemented or document-defined* constraints. They should allow uninterrupted program execution if the input checks implemented by the library developers are robust or if the API description is correct. Generating valid inputs also allows the test case to progress beyond the implemented checks rather than terminating early before executing the deeper code segments, allowing us to identify missing or incomplete checks. In contrast, invalid inputs breach the developer- or document-defined constraints. A correct implementation should thus recognize these violations, halt the program, or trigger an exception when encountering such inputs.

To determine the input properties for classical computing constraints, Upbeat frames the input generation as a constraint satisfaction problem [30]. It then employs a constraint solver [12] to generate the valid and invalid inputs. For quantum constraints, it first represents the constraint using *quaternions* for qubit states. It then applies the quantum rotation operation to alter the qubit state to create valid and invalid qubit inputs. The test code and input data are assembled using a template, resulting in the final test case.

Generating inputs from the library developer-implemented checks and document-defined constraints allows us to establish a test oracle at the Q# language level. This is essential because relying solely on differential testing [8, 18, 37] to run the test cases on multiple quantum simulators [10, 26, 29, 76] might yield identical results even if there is an implementation error at the Q# level. If no discrepancies are found at the language level, Upbeat then applies differential testing to identify inconsistencies at the circuit level by executing a test case on multiple quantum simulators.

```
1 namespace HelloQuantum {
2     open Microsoft.Quantum.Intrinsic;
3     @EntryPoint()
4     operation PrepareBellState() : Unit {
5         use qubits = Qubit[2];
6         H(qubits[0]);
7         CNOT(qubits[0], qubits[1]);
8         DumpMachine();
9         ResetAll(qubits);}}
```

Figure 1: A simple Q# program.

We have implemented a working prototype of Upbeat. We evaluated Upbeat by applying it to all four application domains supported by the Q# Development Kit Libraries: `Standard`, `Chemistry`, `Machine Learning`, and `Numerical Computation`, covering 35 libraries and 881 operations and functions. Within 100 hours of automated test runs, Upbeat identified 20 unique boundary bugs across all tested domains, covering 19 previously unknown bugs. Of all 20 reported bugs, 14 have been confirmed, and 12 bugs - including 4 API document bugs - have been fixed, leaving 6 newly discovered bugs being examined by developers at the time of submission. We also compare Upbeat to six prior quantum software testing methods [4, 21, 38, 53, 59, 62] and two variants of our techniques. Our evaluation shows that Upbeat is highly effective in generating input values for exposing boundary bugs for Q# libraries, uncovering more boundary-checking bugs than the baseline methods within the same testing time.

The core contribution of this paper is the first fuzzing tool to test boundary bugs of quantum libraries. Our work provides a way to extract test oracles and constraints from the API implementation and document for effective test input generation.

2 BACKGROUND AND MOTIVATION

2.1 Quantum Computing

Quantum bits. Quantum computers store and process information using qubits. A qubit can represent `Zero`[1] and `One` simultaneously due to its *superposition* property [36, 49]. A qubit will collapse into either `Zero` or `One` with a probability once it is measured. The states of multiple qubits can be *entangled*, meaning that altering the state of one of them will immediately affect the states of the others.

Qubit states. In this work, we use the `Dirac` notation [13] to encode the value of a qubit state. Specifically, we use $|0\rangle$ and $|1\rangle$ respectively to represent `Zero` and `One` along the `Z-axis`.

Quantum circuits. High-level quantum programs are usually compiled into low-level quantum circuits consisting of qubits and quantum gates to be executed on a simulator or quantum hardware [35]. The gates manipulate the qubits and perform operations like rotation, phase shifts, and entanglement.

Hybrid quantum computing. A quantum program is often implemented in a hybrid mode by combining classical computing and quantum algorithms. Code running on classical computers is often used to prepare the input and output of the quantum algorithm and control the execution of the quantum algorithm.

IntToPauli function

1 **Namespace:** Microsoft.Quantum.Simulation
2 **Package:** Microsoft.Quantum.Standard
3 Converts a integer to a single-qubit Pauli operator.

```
4  Q#                                                    Copy
5  function IntToPauli (idx : Int) : Pauli
```

6 **Input**
7 **idx:** Int
8 Integer in the range 0..3 to be converted to Pauli operators.
9 **Output:** Pauli
10 A Pauli operator given by [PauliI, PauliX, PauliY, PauliZ][idx].

Figure 2: An example function defined in Q# API document.

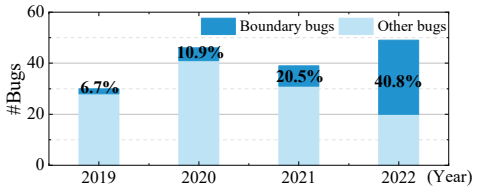


Figure 3: The proportion of boundary bugs in Q# libraries.

2.2 The Q# Programming Language

Figure 1 is a simple Q# program encapsulated within a namespace. Within this program, a `PrepareBellState` *operation* applies the Hadamard (`H`) and Controlled NOT (`CNOT`) gates on two qubits before printing the qubit states by calling *DumpMachine* and release the allocated qubits by invoking *ResetAll*.

Callables. Operations and functions are referred to as *callables* in Q#. An operation is a subroutine containing quantum operations that modify the state of the qubit register. In contrast, a function is typically used to handle classical computation.

Execution. In this work, we execute a Q# program under the .NET core with three built-in quantum simulators provided by QDK. This is done by using the "`dotnet run -s <simulator>`" command-line interface to execute a program in a specific simulator, starting from a callable with the "`@EntryPoint()`" annotation (line 3 in Figure 1).

Q# constraint statements. Q# has two ways to specify constraints: *facts* (check conditions on the input values) and *assertions* (check conditions on the input states of qubits). In addition, the API document also describes the constraints for some APIs, as shown in line 8 in Figure 2, which may not be correctly implemented by the library developers, or may be incorrectly described in the document.

Q# API document. Our work leverages information provide by the official Q# API documents [42] to generate test code. For each callable, this document describes the source (the `Namespace` and `Package` it belongs to), the functionalities, its calling interface, the types of parameters required, and the return values. Figure 2 shows the Q# API document of the `IntToPauli` function.

2.3 Motivation

This work focuses on exposing boundary bugs in Q# libraries due to the lack of or incomplete checks on the properties (e.g., values, length, or qubit states) of an input passed to a user-facing Q# callable. Our work is motivated by the observation that boundary bugs are prevalent in quantum libraries, and such bugs can increase the development cost of library users.

[1]We use `Zero` and `One` to represent 0 and 1 in quantum computing, respectively.

```
1 operation ApplyX... (func : BigInt, controlRegister : Qubit
      [], target : Qubit) : Unit {
2   let vars = Length (controlRegister);
3   let maxValue = PowL (2L, 2^vars);
4   Fact (func>= 0L and func<maxValue, $"...");
5   AssertQubit (Zero, target);
6   ...
7   let table = Encoded (SizeAdjustedTruthTable(..., vars));
8   let spectrum = FastHadamard... (table);
9   ...}
```

Figure 4: A buggy Q# operator that fails to check the `controlRegister` input. Operation and function names are shortened to aid clarity.

In our pilot study, we counted the bugs reported on two representative Q# libraries hosted on GitHub: QuantumLibraries [46] and qsharp-runtime [45]. We search for developer-verified GitHub issues with bug-related labels - "*Kind-Bug*" in QuantumLibraries and "*Bug*" in qsharp-runtime. We then manually examine all the related issues to identify how many bugs are attributed to be boundary bugs. Figure 3 reports the number of bugs from 2019 to 2022. We can observe that boundary bugs are consistently a problem for Q# libraries, with an increasing percentage of boundary bugs reported over the years as the library code base increased. This is unsurprising; as the number of APIs provided grows and the input types become more diverse, it becomes challenging to provide robust input checks.

Figure 4 is a simplified operation from the Q# standard library, and it contains a boundary bug identified by Upbeat. Specifically, a runtime error will be thrown out when invoking the *internal* `FastHadamard` function at line 8 when the length of `controlRegister` is 0 (line 2). This happens because a 0-lengthy `controlRegister` leads to an invalid input (`table`) to be passed to `FastHadamard`. Asking a library user to debug this runtime error would be challenging as the error message does not directly correlate with the initial operation inputs. Proper implementation requires examining the properties of `controlRegister` and producing an error message at the entry point of the user-facing `ApplyX` API should an invalid input be given.

To generate a test case to expose the boundary bug, we need to generate (1) valid values of `func` and `target` to pass the checks enforced in lines 4 and 5 before reaching `FastHadamard` and (2) invalid properties of `controlRegister`. Furthermore, an erroneous implementation at this operation will lead to the same execution outcomes on different Q# simulators, preventing differential testing from uncovering the bug. Therefore, we must also establish a test oracle in the Q# language level to capture such anomalies.

3 OUR APPROACH

Figure 5 provides an overview of Upbeat, an open-source framework for testing boundary bugs of user-facing callables in Q# libraries. It is designed as a while-box fuzzer to be used by the library developers, assuming the system can access the source code of the testing Q# libraries.

Upbeat first gathers code segments from multiple sources, including code samples in the API documents, source code of the target Q# libraries, and open-source Q# programs hosted on GitHub (Section 3.1). Next, it uses the extracted code segments as the building blocks to create the initial test code for certain Q# callables

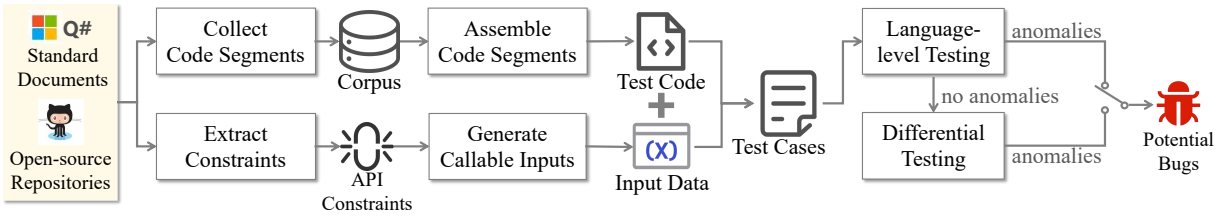
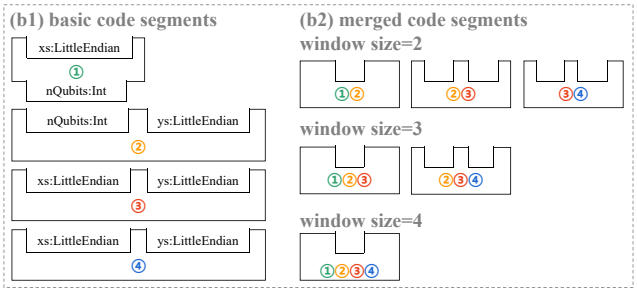


Figure 5: High-level overview of UPBEAT, which tests the input checking mechanism for Q# libraries.

```
1 operation RippleCarryAdderNoCarryTTK
2 (xs : LittleEndian, ys : LittleEndian) : Unit is Adj + Ctl {
3     let nQubits = Length(xs!);①
4     EqualityFactB(nQubits == Length(ys!), true,
5     "Input registers must have the same number of qubits." );②
6     if (nQubits > 1) {
7         ApplyWithCA(ApplyOuterTTKAdder, a
8         ApplyInnerTTKAdderWithoutCarry, (xs, ys));}③
9     CNOT (xs![0], ys![0]);}④
```

(a) Basic segments extracted from the `RipplCarryAdderNoCarryTTK` operation.



(b) Extracted code segments and their assembly constraints.

Figure 6: An example for extracting code segments.

through code synthesis (Section 3.2). To generate the input arguments of the test callable, UPBEAT uses the information extracted from the API documents and their implementations to generate inputs with valid and invalid properties (Section 3.3). These inputs and the test code are used to populate a template to assemble an executable test case to invoke the test callable during execution (Section 3.4). During testing, these test cases are compiled and executed on quantum simulators to identify potential bugs (Section 3.5).

3.1 Type-directed Code Segment Collection

We collect code segments from API documents and source code from 35 open-source Q# libraries and open-sourced Q# projects hosted on GitHub. Specifically, we collected 3,378 code segments with 12,518 lines of code in this work. This is a relatively small corpus due to the smaller code base of quantum programs compared to traditional programming languages like C and Java, as quantum programming is still in its early stages.

3.1.1 API probing. We extract the test targets (user-facing callables) from the Q# API documents [42]. Specifically, we extracted the name of the callable, namespace, input argument types, and return types from metadata, which is stored in a JSON object, and then generated input cases for test code. In summary, we gathered 881 out of all 1,017 APIs (excluding 136 deprecated callables), including 361 operations, 450 functions, and 70 user-defined data types from 35 libraries in the latest version of the official API documents.

3.1.2 Code segments collection. We collected Q# code segments to assemble test programs using code synthesis. These callable code segments were gathered from code samples in API documents, the source code of QDK, and 12 GitHub open-source repositories where Q# was the primary programming language. Subsequently, we divided each collected callable into smaller candidate segments, which were used to assemble entire segments for test code synthesis.

Extract code segments. We identified three types of basic segments from a Q# callable: (1) sequential statements that refer to code segments consisting of single statements, e.g., the variable declarations and reassignment; (2) API call statements that invoke a Q# callable; and (3) Q# control statements including `while`, `for`, `repeat-until`, `within-apply` and `if-elif-else`. These basic code segments cover almost all the types of code supported by Q#, which can generate test cases with richer semantics, contributing to testing deeper code branches of Q# libraries. We created a Q# extractor to use regular expressions to extract code segments. For each Q# program, our extractor first gathers all functions and operations. Then, for each extracted callable, we apply regular expressions to divide the callable into small basic segments in order. Additionally, we merge adjacent k basic code segments into larger segments using a sliding window to increase the diversity of the candidate code segments within each callable. In our work, we set k to range from 2 to the total number of basic segments. Finally, we save basic and merged code segments to our Q# language corpus. We also applied the same methodology to construct the code segments from the code examples given in the API document.

Parse segment assembly conditions. When collecting code segments, we record the variable names and their types required during the code segment assembly process in a *segment assembly constraint*. This constraint is defined as a consumer-producer-like tuple [14]: ⟨`pre-conds`, `post-conds`⟩ where `pre-conds` specifies a set of variable symbols and types necessary for the successful execution of the code segment, and `post-conds` specifies the available variable types. The *segment assembly constraint* ensures the assembled code segments are well-typed. The rule of each condition can be represented as `{var:type}`, where `var` represents the variable name, and `type` represents the variable type. To parse these conditions, we retrieve all variables used in each segment. For all required variables when executing a code segment, we record them and their types in the `pre-conds` and save other variables in the `post-conds`.

Example. Figure 6 shows the code segment extraction process for a Q# operation. We first divide the code into four basic segments, including three API call statements (lines 3, 4-5, and 9) and an `if` statement (lines 6-8). We number the basic segments from ① to ④ according to the order in which the codes appear. Then, we assemble these basic segments using merging windows with sizes 2 to 4. This gives 10 code segments as shown in Figure 6b. The merged

code segment is also annotated with ⟨`pre-conds, post-conds`⟩, where the `pre-conds` are the `pre-conds` of the first basic code segment; the `post-conds` are the `post-conds` of the last basic code segment. For example, `{nQubits:Int; ys:LittleEndian}` is the `pre-conds` of the segment ②. The `post-conds` of the segment ③ are `NULL`. Finally, the constraint of the merged segment ②③ is ⟨`{nQubits:Int; ys:LittleEndian}, {NULL}`⟩.

3.2 Test Code Generation

We generate test code similar to a function body within a namespace for testing a Q# callable. The generated test code does not include the variable declarations and diagnostic statements to dump the execution outcomes (e.g., qubit states), which will be inserted at the final stage of test case generation (Section 3.4). The test code is synthesized via assembling type-directed code segments. The synthesized test code contains one or more Q# callables supported in the target Q# libraries.

Algorithm 1 outlines how UPBEAT synthesizes test code via *type-directed segment assembly*. The algorithm takes in a list of the gathered code segments and the maximum number of code segments used to assemble a test code. Each list element is an object consisting of a code segment and its corresponding assembly constraint. To synthesize a test code, UPBEAT randomly selects a code segment with at least one callable from the list as the assembly seed (line 2). Within each iteration, UPBEAT first determines if a code segment can be merged with *seg* (line 4). The process for determining an available code segment is shown as the `SEARCHCODESEGMENT` procedure (lines 12–28). Specifically, UPBEAT searches the list to find the code segments whose `post-conds` match the seed's `pre-conds`, i.e., when one of the variable types is matched (lines 15–16). Otherwise, it finds the matching code segments according to the seed's `post-conds` (lines 17–18). UPBEAT randomly chooses and returns a matching code segment (lines 21–24). If no suitable code segment is found, it returns "NULL" (lines 25–26). The returned code segment and the seed code are merged into a large code segment (line 5). The iteration concludes if the merged code segment has no more `pre-conds` and `post-conds` left (lines 6–7). Finally, a new assembly code segment is returned (lines 10–11).

3.3 Test Input Data Generation

UPBEAT generates test inputs by first extracting the input constraints from the source code and API document.

3.3.1 Constraint extraction. The Q# library implementation follows the hybrid computing mode, using classical and quantum constraints (Section 2.1). We express classical constraints as:

$$expr_1 \Theta expr_2 \Theta ... \Theta expr_i, \quad 1 \le i \le \infty; \quad \Theta \in \{\&, ||\} \tag{1}$$

where $expr_i$ denotes an inequality or equality and Θ is a collection of logical operators (e.g., & or ||). For instance, the constraints in line 4 of Figure 4 can be expressed as $func \ge 0 \& func < maxValue$. In contrast, the quantum constraints model the qubit state. In this work, we model the constraint $\mathbb{C}$ of the qubit χ as:

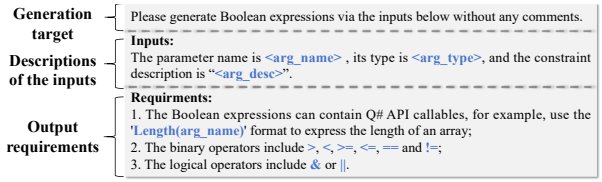
$$\chi \leftarrow \{\Psi, \quad \phi, \quad \rho, \quad \delta\} \tag{2}$$

χ is a variable that must be in the state represented by the quaternion, such as $|0\rangle$ or $|1\rangle$. The quaternion signifies that the probability of ϕ along the Ψ axis is ρ within the tolerance of δ. Here, Ψ can be one of the three axes on a sphere: `X-axis`, `Y-axis` or `Z-axis`. ϕ can be `One` or `Zero` - two possible values defined as `Result` type in Q# language. ρ is a floating-point number within the range of 0 to 1, and δ can take any arbitrary float number. For example, the constraint in line 5 of Figure 4 can be expressed as $target \leftarrow$ {`Z-axis, Zero, 1.0, 0.0`}, where *target* should be $|0\rangle$.

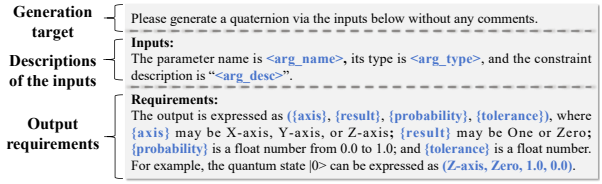
UPBEAT extracts the constraints *offline* from (1) the Q# API implementations via regular expression matching and (2) parameter descriptions in the API document. The latter helps collect relevant constraints because the constraints of many APIs are described as plain texts in their respective documents. Collecting information from the API document also enables UPBEAT to check the discrepancies in the developer manual and the library implementation. In our implementation, if a constraint is only documented or implemented, we check if valid and invalid inputs yield the expected behaviors during execution. Conversely, when constraints are both implemented and documented, we check for consistency between the documentation and implementation.

Algorithm 1 Type-directed Code Segments Assembly Algorithm

```
Input:
    segList: The list that stores the collected code segments.
    N: Maximum number of code segments that assemble a test code.
Output:
    ret: The assembled test code.
 1: Let seg, temp be the empty objects;
 2: seg ← selectCodeSegmentwithAPI(segList);
 3: for i = 0 : N − 1 do
 4:     temp ← SEARCHCODESEGMENT(seg, segList);
 5:     seg ← AssembleCodeSegments(seg, temp);
 6:     if seg.preConds & seg.postConds == NULL then
 7:         break;
 8:     end if
 9: end for
10: ret = seg.code;
11: return ret;
12: procedure SearchCodeSegment(seg, segList)
13:     Let postList, preList be empty lists;
14:     for t ∈ segList do
15:         if satisfy(t.postConds, seg.preConds) then
16:             postList.append(t);
17:         else if satisfy(t.preConds, seg.postConds) then
18:             preList.append(t);
19:         end if
20:     end for
21:     if postList.length > 0 then
22:         return RandomSelect(postList);
23:     else if preList.length > 0 then
24:         return RandomSelect(preList);
25:     else
26:         return NULL;
27:     end if
28: end procedure
```



(a) Prompt paradigm for extracting the classical constraints.



(b) Prompt paradigm for extracting the quantum constraints.

Figure 7: GPT template prompts.

Extract constraints from source code. To extract constraints from the API source code, Upbeat first analyzes which APIs contain statements used to specify constraints (*facts* and *assertions*). Upbeat gathers constraint statements using regular expressions. Then, Upbeat transforms the constraint statements into either Boolean expressions or quaternions. Specifically, to extract classical constraints, Upbeat extracts the conditional expression or combines several variables using logic connectors to transform them into constraint expressions. To extract quantum constraint, Upbeat obtains variables in the argument list and supplies the remaining items in the quaternion based on the meaning of the *assertions*. During extraction, Upbeat runs a data flow analysis to align local variables of the constraint expressions with input arguments passed to the callable. For the example in Figure 4, *maxValue* in the `Fact` function will be replaced with `PowL(2L,2^Length(controlRegister))`. This process captures how the input arguments affect the outcome of the input checking.

Extract constraints from API documents. We utilize GPT-4 [50] to transform constraint expressions from API documents through two template prompts: a zero-shot prompt used for extracting the classical constraints (Figure 7a) and a few-shot prompt for extracting the quantum constraints (Figure 7b). We use a script to first populate the parameters in the prompt inputs, e.g., the `arg_name`, `arg_type`, and `arg_desc`, using the API document texts. Subsequently, the prompt is sent to the GPT-4 API to retrieve and store the generated responses in a database.

3.3.2 Callable input generation. Upbeat solves the extracted classical or quantum constraints to generate callable inputs.

Classical inputs. We built a solver based on `Z3` [12] to generate valid and invalid inputs for classical computing constraints. Our implementations supports all relational operations, including $>$, $\geq$, $<$, $\leq$, $==$ and $\neq$. We simplify the constraint expressions using `SymPy` [2], a Python library for symbolic mathematics. For instance, constraints "$a > 0 \& a \geq 1 \& b < a+5$" will be simplified to "$a \geq 1 \& b < a+5$". The simplified constraint expressions are then broken down into several inseparable sub-constraints, e.g., "$a \geq 1 \& b < a+5$" will be divided into "$a \geq 1$" and "$b < a+5$" with the logical "`AND`" relation. Next, we find if there are sub-constraints with the binary operators like "==", "$\leq$", or "$\geq$" by using regular expressions (e.g., "$a \geq 1$"). If it exists, the operator of the sub-constraints will be replaced with "==" (e.g., "$a == 1$"), and then it and the remaining sub-constraints are input into `Z3` to generate the valid test inputs; otherwise, the simplified constraints are fed into `Z3` to solve for values that fulfill the constraints. The invalid inputs are generated following the above steps after inverting the constraints. Some special constraints may contain Q# callables, e.g., `PowL` in line 3 of Figure 4. Before solving such constraints, we first convert it into a mathematical expression based on its semantic, e.g., `PowL(2L,2^Length(controlRegister))` will be replaced with `2^(2^(Length(controlRegister)))`.

Quantum inputs. We generate valid and invalid inputs for a quantum constraint to be passed to a callable. The former leads to qubit states that satisfy the quaternions defined in Equation 2, while the latter would violate the quaternion. Unlike classical constraints generated using `Z3`, we generate quantum constraint inputs by applying the quantum rotation operation to alter the qubit state. Specifically, given a constraint $\mathbb{C}$ for qubit χ, the valid inputs can

```
1 Fact(to >= from, $"`to` must be larger than `from`");
2 Fact(to - from <= 0x07FFFFFFFFFFFFFEL, $"different between `
      to` and `from` is too large");
```

(a) Constraint-related statements in SequenceL.

```
1 to >= from
2 to - from <= 0x07FFFFFFFFFFFFFEL
```

(b) Extracted Boolean expressions of (a).

```
1 let from = 0L;
2 let to = 9223372036854775806L;
```

(c) Generated valid values.

```
1 let from = 0L;
2 let to = 92233720368547758O7L;
```

(d) Generated invalid values.

Figure 8: Example of extracted classical constraints and the Upbeat-generated input data.

```
1 AssertAllZero(res!);
```

(a) Constraint-related statements.

```
1 res<--{PauliZ,One,0.0,0.0}
```

(b) Extracted constraints of (a).

```
1 use qs1 = Qubit[2];
2 let theta = 2.0*ArcSin(Sqrt
      (0.0));
3 Ry(theta, qs1);
4 let res = LittleEndian(qs1);
```

(c) Generated valid values.

```
1 use qs1 = Qubit[2];
2 let theta = 2.0*ArcSin(Sqrt
      (0.0))+0.1;
3 Ry(theta, qs1);
4 let res = LittleEndian(qs1);
```

(d) Generated invalid values.

Figure 9: Example of extracted quantum constraints and the Upbeat-generated input data.

be generated by adding the rotation operation as:

$$\begin{cases} \theta = 2\arcsin\sqrt{\rho+\delta} \quad or \quad \theta = 2\arcsin\sqrt{\rho-\delta}; \\ R_i(\theta,\chi), \quad i \in \Psi; \quad \phi = \texttt{Zero} \end{cases} \tag{3}$$

The first term of Equation 3 leverages the extracted constraint $\mathbb{C}$ to calculate the rotation angle θ. Then, a rotation operation R_i in the second term is applied to the target qubit χ for a given rotation angle θ. R_i is determined based on the axis Ψ. For example, if Ψ is `Z-axis`, then R_i can be R_x or R_y, meaning rotating χ by θ degrees along the x or y axis. On the contrary, the invalid values are generated by rotating χ with a randomly chosen angle smaller or larger than θ. Note that θ in Equation 3 is derived when the qubit state ϕ equals `Zero`. Thus, the state of the qubit χ needs to be `Zero` before applying the rotation operation.

Random input generation. To generate test inputs for callables that do not implement assertion-like statements or no constraint specification was given in the API document, Upbeat generates inputs for three classical computing types. Specifically, we define seven values for `Int` (0, $\pm$1, 63, 64, -2^{63} and $2^{63}-1$), five values for `BigInt` ($0L$, $\pm 1L$, $-2^{63}L$ and $2^{63}L-1L$) and six values for `Double` (0.0, NaN, $\pm\infty$ and $\pm 1.79..E308$). These special values are gathered from the historical test cases that exposed Q# bugs. For other input data types, Upbeat generates random values or simply copies inputs from the code segments in the collected corpus.

3.3.3 Examples of generated inputs. We now present some inputs generated by Upbeat for classical and quantum constraints.

Classical constraint inputs. Figure 8a are two `Fact` statements extracted from the `SequenceL` function. To generate the valid values that satisfy the classical constraints defined in the statements, Upbeat first transforms the constraints into the Boolean expression shown in Figure 8b. The Boolean expression is then divided into two sub-expressions: "$to \geq from$" and "$to - from \leq$ 0x07F..EL". Next, the second sub-expression is modified to "$to - from ==$ 0x07F..EL" for solving the inputs. Finally, the altered sub-expression

```
1  namespace Test {
2      <Open directives>; // Necessary namespaces
3      <Callable declarations>;
4      // Parameters for type function (Optional)
5      @EntryPoint()
6      operation main() : Unit {
7          <Variable declarations>;  // Boundary values
8          <Test code>;  // Assembled code segments
9          <Diagnostic statements>;  // Output statements
10     }
11 }
```

Figure 10: The template used for synthesizing test cases.

and "$to \geq from$" are fed into Z3, producing the valid inputs shown in Figure 8c. To generate the invalid inputs, Upbeat first negates "$to - from \leq$ 0x07F..EL" to "$to - from >$ 0x07F..EL" and then repeats the above solving process, resulting in the outcomes shown in Figure 8d.

Quantum constraint inputs. Figure 9a is a quantum constraint that checks qubit states of `res` in the `ComputeReciprocalI` operation. It uses the `AssertAllZero` statement to check if the state of res is $|0\rangle$. res is a qubit register that encodes an unsigned integer in little-endian order. The "!" keyword takes the content encapsulated in res without changing the qubit state. Thus, the constraints of res can be expressed as $res \leftarrow \{$`Z-axis`, `One`, 0.0, 0.0$\}$ shown in Figure 9b. To generate a valid input, Upbeat first defines $qs1$, a qubit array with a random initialization length (2 in line 1 of Figure 9c). It then calculates the rotation angle θ (line 2) to derive the valid input (line 3). The rotation operation R_y is used because Ψ in the constraint is `Z-axis`. Finally, $qs1$ is cast to `LittleEndian` according to the prototype of `ComputeReciprocalI` (line 4 in Figure 9d).

3.4 Test Case Assembly

Test case template. We use a code template, as depicted in Figure 10, as a skeleton to generate the Q# test case that includes code statements and data to drive the testing. This template comprises three key components: (1) variable declarations (line 7), (2) a test code (line 8) containing API call statements to test Q# library callables, and (3) diagnostic statements (line 9) for outputting the execution results. In certain test cases, Upbeat may also generate parameters for type functions (line 3) required by the test callables.

Test case generation. To generate a test case, Upbeat firstly populates the template with the test code synthesized (Section 3.2) and variable declarations generated (Section 3.3). Upbeat then inserts the remaining elements in the code template. It starts by importing the necessary namespaces of the API used. The namespace information is retrieved from the JSON file generated during API probing (Section 3.1.1). Then, if an API requires a callable as a parameter, it searches the JSON file created during the API probing stage to obtain a proper API name. Additionally, we store functions and operations as described in Section 3.1.2, which can be added to test cases and passed to relevant APIs. Upbeat insert Q# subroutines, `Message` and `DumpMachine` to output diagnostic information for differential testing. Here, `Message` is used for classical variables, and `DumpMachine` outputs the status of all currently allocated qubits. Finally, Upbeat removes test cases that fail to compile.

3.5 Test Case Execution

The test oracles of Upbeat come from (1) language-level testing via constraints implemented in the source code or specified in the API document or (2) differential testing by executing the same test case on three Q# simulators: `QuantumSimulator` [40], `SparseSimulator` [26], and `ToffoliSimulator` [41]. We can establish the oracle for test cases via language testing when the inputs are generated from input constraints. We run these test cases on `QuantumSimulator` to check if the program behavior is as expected. If no anomalous behavior is caught, the test case will continue to be executed using differential testing. For test cases with randomly generated inputs, Upbeat also leverages differential testing to establish the test oracle.

3.5.1 Anomalies identification. We consider four types of anomalies during test runs. The first is `BoundError`, where we can establish the test oracle from the extracted constraints. This happens when Upbeat runs a test case with valid inputs but captures an exception or, conversely, when Upbeat executes a test case with invalid inputs but the program finishes without errors. We note Q# does not support optional types, and we consider it a potential bug to successfully execute invalid inputs without exceptions. While the library implementation may choose not to throw out errors for invalid inputs, it is still worth flagging this potential issue to the developers. We also consider the `Inconsistency` of execution outcomes observed during differential testing. This occurs when the outcomes vary across multiple simulators. Finally, `Crash` and `Timeout` occur when using a single simulator or in the context of differential testing. We set a timeout threshold to 2 minutes, which is sufficient for our test cases.

3.5.2 Postprocessing of differential testing. The Upbeat test case template uses the Q# `DumpMachine` callable to output the states of all allocated qubits (Section 3.4), which are then compared against the results collected during differential testing. We developed a script to post-process the output to align the qubit states across simulation outputs.

4 EXPERIMENTAL SETUP

Our experiments were designed to answer the following research questions (**RQs**):

- **RQ1:** How effectively Upbeat is on detecting boundary bugs in Q# libraries (Section 5.1)?
- **RQ2:** How does Upbeat compare with prior methods and baselines on bug detection (Section 5.2)?
- **RQ3:** How do individual components of Upbeat contribute to its overall performance (Sections 5.3)?
- **RQ4:** How effective is Upbeat in extracting constraints from Q# libraries and API documents (Section 5.4)?

4.1 Competitive Baselines

We compare Upbeat against six baselines, including a generative fuzzer [59] and five mutational methods [4, 21, 38, 53, 62]. We also implement a variant by combining all the above mutation methods and a variation of Upbeat. These lead to a total of eight evaluation baselines, described as follows.

Qsharp-Fuzz. This is a deep-learning-based fuzzer for testing Q# compiler implementations [59]. We use the default parameters

defined in QSharp-Fuzz to train its generation model using its training programs and code samples collected in this work.

Quito. This tool tests quantum programs written in high-level quantum languages [4]. It performs mutation analysis to assess the effectiveness of the generated test cases in finding faults.

QSharpCheck. This is a property-based testing framework for Q# programs [21]. It uses mutation analysis to evaluate the effectiveness of the generated test cases.

Muskit. This is a mutation-based analysis tool for generating quantum programs written in Qiskit [38].

QDiff. This is a mutational method [62] starts from a set of quantum programs written in Qiskit. It applies a set of equivalent gate transformation rules and leverages the K-S test [57] to compare execution results among a set of equivalent test cases.

MorphQ. This is a metamorphic testing method [53] combines template-based and grammar-based code generation. It provides ten quantum-specific metamorphic rules to generate test cases.

Upbeat-M. In this baseline, we combine all the mutation operators used by Quito [4], QSharpCheck [21], and Muskit [38] to create a stronger mutator upon on the testing framework of Upbeat.

Upbeat-R. This variant of Upbeat generates Q# test codes by randomly assigning values to variables and combining code segments instead of leveraging the segment assembly conditions and the input constraints.

We implemented the mutators of Quito, Muskit, QSharpCheck, QDiff, MorphQ and Upbeat-M on top of Upbeat by replacing Upbeat's input generation module with a mutation module. We provided these mutation-based methods with the same test codes generated by Upbeat that random input generation (rather than Upbeat's test input generation) as the seed programs.

4.2 Evaluation Methodology

We consider the two quantified metrics, *code coverage* and *bug-exposing capability* when comparing Upbeat with the competitive baselines. The code coverage measures the code coverage for the Q# library APIs. It is useful as poorly generated test cases can contain many non-executed code blocks. We use `dotnet-coverage` [39] to collect the coverage information from the instrumented repository QuantumLibraries [46] for line and block coverage. We also count the number of unique anomalies identified during test case executions. All duplicated anomalies were removed via manual analysis. Therefore, each of them indicates a potential bug.

4.3 Evaluation Systems

We implemented Upbeat in around 10K lines of Python code. We test Upbeat on QDK version 0.24, running in .Net core version 6.0.3. Test cases were executed on three Quantum simulators from QDK (Section 3.5). We perform test runs on a multi-core workstation running the Ubuntu 18.04 operating system and an AMD EPYC 7532 32-core processor with 128GB of RAM. We run the test cases concurrently using docker containers.

5 EXPERIMENTAL RESULTS

Highlights of our experiments are:

Table 1: Bug list exposed by Upbeat.

No.	Buggy APIs	Namespace	State	Feature	Category
1	Binom	Math	F	†‡	Impl.
2	HalfIntegerBinom	Math	F	†‡	
3	Sin	Math	C	†	
4	Ceiling	Math	C	‡	
5	Truncate	Math	S	‡	
6	IncrementPhaseByModularInteger	Arithmetic	S	∗	
7	IdenticalPointPosFactFxP	Arithmetic	F	∗	
8	Chunks	Arrays	F	†	
9	SequenceL	Arrays	S	†	
10	IntAsBoolArray	Convert	F	†	
11	Parity	Bitwise	F	†	
12	PurifiedMixedStateRequirements	Preparation	F	‡	
13	AssertQubitWithinTolerance	Diagnostics	S	∗	
14	DumpMachine	Diagnostics	F	∗	
15	DumpRegister	Diagnostics	S	∗	
16	Adjoint ApplyAnd	Canon	S	∗	
17	Padded	Arrays	F	-	Doc.
18	Last	-	F	-	
19	ApproximateFactorial	Math	F	†	
20	_ComputeJordanWignerBitString	Chemistry	F	-	

- Upbeat has uncovered 16 implementation bugs and 4 API document errors, covering all the quantum features described in Section 2.1 (**RQ1**);
- Upbeat outperforms the competing baselines by providing better code coverage and identifying more potential bugs with the same test time (**RQ2**);
- The Upbeat components all positively contribute to the bug-exposing capability of the framework (**RQ3**);
- Upbeat is capable of extracting the majority of constraints from both source code and API documents with high accuracy (**RQ4**).

5.1 Bug Summary

Table 1 summarizes the implementation bugs and documentation errors found by Upbeat. The bugs were identified within less than 100 hours of execution runs using 12K test cases generated by Upbeat. Upbeat discovered a total of 20 boundary bugs, including 16 API implementation bugs and 4 documentation bugs.

In the table, we note the namespace a buggy API comes from and the state of the submitted bug report, which corresponds to either **S**ubmitted (pending confirmation), **C**onfirmed, or **F**ixed. In the feature column, we summarize the cause of the bug to link it with the mismanagement of qubit states (∗), errors in the mathematical calculation (†), hybrid computing (‡), or other features (-).

Upbeat identified 6 and 2 bugs in the `Math` and `Arithmetic` libraries respectively. These bugs are largely due to the missing or incomplete inputs checking when implementing the quantum operations. This is also observed in `Array` and `Diagnostics` libraries, indicating that boundary bugs are prevalent in Q# libraries. Upbeat has identified bugs related to all features described in Section 2.1, where the highest number of bugs are related to the mismanagement of qubit states or errors in mathematical computations. At the time of submission, 14 Upbeat-identified bugs have been confirmed by the library developers, of which 12 have been fixed, including 8 implementation bugs and all documentation bugs. There are still 6 recently discovered bugs that have not been confirmed. This is because the repositories where we submitted bugs have been deprecated since January 2024 and they have been replaced by a newly developed Q# repository [44] hosted on GitHub.

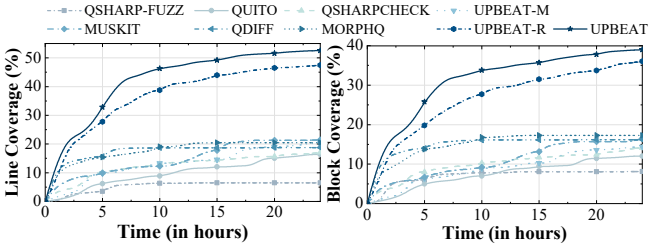
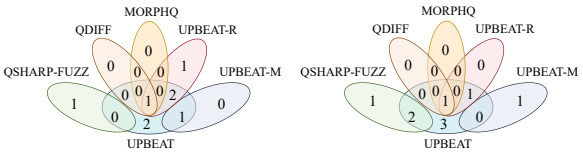


Figure 11: Comparison results of line and block coverage.



(a) via language-level test oracle **(b) via differential testing**

Figure 12: Anomalies discovered by compared baselines.

5.2 Comparison with Baselines

We now compare Upbeat to the eight baselines outlined in Section 4.1 using the metrics defined in Section 4.2. In this experiment, we allocated a budget of 24 hours of concurrent test runs for each baseline, excluding the time for test case generation. Each method runs individually on our evaluation platform for a fair comparison.

5.2.1 Code Coverage. Figure 11 shows how the line and block coverage ratios change as testing time increases. The line and block coverage of mutation-based approaches gradually increased as the test progressed and finally stabilized. The coverage of Upbeat and its variant Upbeat-R consistently increased but has not yet reached a plateau during the test. This is largely because the vast majority of test cases generated by Upbeat and Upbeat-R are syntactically correct. In contrast, many of the test cases generated by the mutational fuzzers have errors due to passing the incorrect inputs. During testing, the API callables in the invalid test cases cannot be executed, resulting in lower code coverage rates. In contrast, QDiff and MorphQ achieve higher line and block coverage than other mutation-based tools. This is because their transformation rules generate the syntactically correct test cases. However, the specific-designed transformation rules decrease the diversity of the generated test cases, resulting in reaching a plateau of code coverage earlier than other baselines. The line and block coverage rates of Qsharp-Fuzz are less than 10%, the lowest values among all compared baselines. This is because its deep-learning-based generator synthesizes many identical test cases.

Compared Upbeat with Upbeat-R, we see that the callable inputs generated based on classical and quantum constraints are effective in discovering new code branches, helping Upbeat achieve the highest code coverage. The mutation variant of Upbeat, Upbeat-M, gives a line coverage of 20% and a block coverage of 14%, lower than the corresponding coverage rates of Muskit. This is caused by the Quito's mutation rules in Upbeat-M, which are inefficient in generating valid test cases with correct syntax. This suggests that a simple combination of mutators may reduce the code coverage.

5.2.2 Bug-exposing capability. In this evaluation, we compare the number of bugs found by Upbeat and five representative baselines: Upbeat-R– a generation-based approach, Upbeat-M and QDiff – two mutation-based approaches, QSharp-Fuzz – a deep-learning-based method, MorphQ – a metamorphic testing method, within the same 24-hour test case execution time. We omit the results of Quito, Muskit, and QSharpCheck because Upbeat-M already includes all the mutation methods used by these three methods, and they individually did not discover potential bugs that Upbeat-M failed to find.

The Venn diagram in Figure 12 summarizes the overlapping and unique anomalies exposed by each method. We group the Upbeat identified anomalies found through test oracles established from implemented or documented constraints (Figure 12a) and those discovered through differential testing (Figure 12b).

With the input generation method described in Section 3.3.2, Upbeat discovered 6 anomalies, covering the majority of anomalies discovered by other baselines. Upbeat also identified a total of 5 anomalies that failed to be discovered by others through language-level test oracles and differential testing. The baseline methods identified four anomalies that Upbeat did not detect. Among these, Qsharp-Fuzz discovered two. The first occurs when a special value is returned from a `return` statement. Upbeat does not capture this because it eliminates `return` statements during the collection of code segments, ensuring the correctness of assembled test codes. The second anomaly results from a unique code segment generated by Qsharp-Fuzz that is absent in Upbeat's code base. In contrast, the sole anomaly unique to Upbeat-R arises from randomly generated inputs, while Upbeat-M's distinct anomaly is identified by its `Replace Arithmetic Operator` mutation operation. Nevertheless, Upbeat identified a total of 13 potential bugs, which is at least 2× more in detecting anomalies over the baseline methods.

5.3 Ablation Study

In an attempt to quantify the contribution of Upbeat's individual components, we evaluate two variants of Upbeat, corresponding to the two key components of the framework: (1) a type-directed code segment assembler that synthesizes test codes (Section 3.2) and (2) a test inputs generator to generate inputs from constraints (Section 3.3). We implemented Upbeat-A and Upbeat-B, where Upbeat-A removes the inputs generator and keeps other parts of Upbeat unchanged. On the contrary, Upbeat-B removes the code segment assembler and keeps other components. For a fair comparison, we compared Upbeat with all two variants using the same seed programs within a test time budget of 48 hours.

Figure 13 reports the number of bugs discovered by two variants and Upbeat, where bugs are grouped into four categories: boundary errors (`BoundError`), `Inconsistency`, `Crash`, and `Timeout`. Upbeat-A discovered six bugs, indicating the effectiveness of assembled test codes. Four of them were triggered by `Crash`, and the other two were found to be `Inconsistency` and `Timeout`, respectively. It is demonstrated that many corner cases are still not handled and result in serious execution results in Q# APIs. Upbeat-B discovered three bugs, all of which are boundary errors, indicating that it is effective in generating test inputs to cover the edge cases that are likely to be overlooked by library developers. The results suggest that the individual components of Upbeat are all essential.

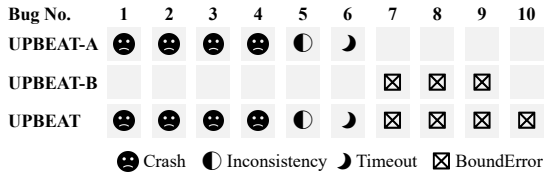


Figure 13: Bugs discovered by UPBEAT and its variants.

Table 2: Analysis results for constraints extraction.

Source	Type	*Recall*	*Precision*
Source Code	Classical Constraints	100%	93%
	Quantum Constraints	100%	96%
API Document	Classical Constraints	80%	81%
	Quantum Constraints	90%	84%

5.4 Constraint Extraction

To evaluate the completeness and correctness of UPBEAT-extracted constraints, we randomly selected 10 Q# libraries and conducted a manual verification. We use two metrics: *Recall* and *Precision*. The first computes the ratio of UPBEAT-recognized constraints to the total number of constraints. It answers questions like "*Of all the constraints, how many are extracted by UPBEAT?*". The second metric computes the ratio *correctly* extracted constraints samples to the total number of constraints. It answers questions like "*Of all the test constraints, how many are correctly extracted by UPBEAT?*"

We group the analysis results based on the source of the constraints, as shown in Table 2. Overall, UPBEAT recognized all constraints for all APIs from their source code implementation. UPBEAT did not extract all constraints from the API document because the description of constraints in the API document is scattered in several fields or paragraphs, making it challenging for the language model to locate the complete constraint chains. UPBEAT achieves a precision of over 90% in extracting constraints from API source code and over 80% from the API document, suggesting that most of the constraints were correctly extracted. UPBEAT can struggle to extract constraints in APIs that have complex code dependencies because UPBEAT cannot convert them into complete Boolean expressions or when the language-model-based extractor misinterprets the constraint descriptions.

5.5 Examples of Bugs Found by UPBEAT

Classical constraint bug. Listing 1 exposes an incorrect constraint implementation in the SequenceL function. This constraint expects `from` to be less than `to`, and `to-from` to be under `0x07F..EL`. As explained in Section 3.3, UPBEAT generates valid and invalid inputs based on the developer-implemented input constraints. Yet, even when `to-from` meets the criterion, the test case triggers an exception when $to - from == 922...6L$. This execution result is contradicted by the test oracle, indicating a potential bug.

```
namespace Test {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Arrays;
    @EntryPoint()
    operation main() : Unit {
        let from = 0L;
        let to = 9223372036854775806L;
        mutable result = SequenceL(from, to);
        Message($"{result}"); }}
```

Listing 1: Test case for exposing incorrect classical constraint implementation of SequenceL.

Quantum constraint bug. Listing 2 is a test case generated by UPBEAT. It exposes a missing quantum constraint checking in `Adjoint ApplyAnd` for inversing the `ApplyAnd` operation, which requires a separate API implementation. The qubit state of the operation's inputs,q1 and q2, were set randomly to $|1\rangle$ and $|1\rangle$, while the qubit state of `res` is initialized to $|0\rangle$. This input is "valid" according to the quantum constraint implementation in `Adjoint ApplyAnd`. When executing this test case using differential testing, the output of ToffoliSimulator is $|111\rangle$, whereas the other simulators yield $|110\rangle$. This is due to a bug in the low-level implementation of `Adjoint ApplyAnd` in `ToffoliSimulator`.

```
namespace Test {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Diagnostics;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Math;
    @EntryPoint()
    operation RunProgram() : Unit {
        use q1 = Qubit(); X(q1);
        use q2 = Qubit(); X(q2);
        use res = Qubit();
        let theta = 2.0*ArcSin(Sqrt(0.0));
        Ry(theta, res);
        Adjoint ApplyAnd(q1, q2, res);
        DumpMachine();
        ResetAll([q1,q2,res]);}}
```

Listing 2: A UPBEAT-generated test case that uncovers a missing input checking in `Adjoint ApplyAnd` for qubit inputs.

API document bug. Listing 3 exposes a documentation description error for parameter n in the `ApproximateFactorial` math function. The document defines the scope of n to be `AbsD(n)<170.0` by ignoring the sign of n. However, the implementation (correctly) assumes "$0 \leq n < 170$". UPBEAT generates a "valid" input for n with a value of -2 according to the API description and expects the test case to run without issues. However, an unexpected exception was thrown, indicating a potential bug.

```
namespace Test {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Math;
    @EntryPoint()
    operation main() : Unit {
        let n = -2;
        mutable result = ApproximateFactorial(n);
        Message($"{result}");}}
```

Listing 3: A UPBEAT-generated test case that signifies an inconsistent constraint between the API document and the actual implementation of `ApproximateFactorial`.

6 RELATED WORK

6.1 Test Case Generation

UPBEAT generates random test cases to test input checks of API implementation. Random test case generation typically relies on program generation or mutation. Many program generation methods use stochastic context-free grammar [6, 60, 69]. With this approach, a fuzzer utilizes a grammar that defines the syntax of the target language to generate syntactically correct programs. These programs are crafted in such a way that their expressions adhere to a specific probability distribution associated with the grammar's productions. UPBEAT can benefit from carefully designed rules for increasing the diversity and coverage of the generated test cases.

Mutation-based testing modifies a set of seed programs to generate test cases [19, 20, 74]. For example, Langfuzz [20] mutates test cases by inserting code segments that previously exposed bugs. CodeAlChemist breaks the seed programs into fragments and uses the fragments to assemble new test programs [19]. Upbeat builds upon these past foundations by combining program generation and mutation methods. It first collects Q# code samples and then uses these samples to synthesize the test code.

Methods have been proposed to specifically target testing bugs at quantum programs [5, 16, 47], using assertion tool [32], as well as generation [61, 64] and mutation [15, 38] methods. These works typically require a well-defined specification to detect unexpected behavior of the test programs. None of them targets the input-checking mechanism of quantum libraries nor leverages the constraint implementation to generate test inputs like Upbeat.

6.2 Constraint Extraction

Upbeat leverages the developer-implemented and document-specific input constraints to generate test inputs and establish the test oracle. Jdoctor [9] combines natural language parsing and semantic matching techniques to extract constraints from documents to generate input data. Similarly, DASE [65], Comfort [70], and Docter [68] leverage natural language processing techniques and heuristics to extract input constraints from documents or language specifications. ACETest [56] identifies the input validation paths in the source code of DL operators, extracts the constraints related to the user inputs, and sends them into the Z3 solver to get various sets of solutions. Upbeat builds upon these prior works but extends the analysis and representation to extract constraints from source code to generate test inputs by capturing the unique characteristics of quantum computing (e.g., how to represent the qubit state).

6.3 Quantum Software Testing

Recent work has targeted testing quantum compilers and backends [48, 53, 62]. However, little work has been considered on testing the input-checking mechanism of quantum libraries. Upbeat aims to bridge this research gap. Furthermore, differential tests performed on low-level compilers and simulators cannot expose bugs at the high-level implementation, as these errors lead to the same compilation and execution outcome. Upbeat addresses this issue using the developer-implemented and document-defined constraints to establish the test oracle via language-level testing.

Other works investigate ways to model the correctness of non-deterministic quantum program execution [4, 21, 48], generate quantum benchmarks [11, 31, 55, 66, 73] or study the bug patterns in quantum programs [23, 24, 34, 72] and in quantum computer platform [52]. Upbeat will benefit from the findings and techniques developed in these works.

7 THREATS TO VALIDITY

Our work takes Q# as a case study. However, we found that boundary bugs also exist in other mainstream quantum programming models, including IBM's Qiskit [1]. Extending Upbeat to other quantum programming models would require adapting our tools to collect code segments and constraint information and the template used for synthesizing test cases. However, our key idea of leveraging constraints to generate test input and establish the test oracle remains applicable.

We developed a simple yet effective tool to collect code segments using regular expressions. A better approach would be developing a compiler parser to achieve the tasks, allowing us to handle more complex data flows and dependencies.

Upbeat relies on successful runtime execution or exception to capture abnormal behavior. It does not check if the execution results are incorrect. As quantum program execution outcomes are not deterministic, establishing a test oracle for execution results would require modeling the probability distribution of qubit measurements. Therefore, techniques for assessing the correctness of quantum program executions are orthogonal to Upbeat [3, 53].

8 CONCLUSION

We have presented Upbeat, a framework to test the input-checking mechanism of functions and operations for Q# quantum libraries. Upbeat automatically generates test Q# programs and inputs and executes the generated test cases to identify abnormal behaviors. Upbeat achieves this by leveraging code samples from the library implementations, open-source projects, and API documents to synthesize test code. It then leverages input constraints implemented in the source code described in the API documents to generate valid and invalid input test data. We evaluate Upbeat by applying it to the Q# development kit libraries. Within 100 hours of automated test runs, Upbeat has identified 16 API implementation bugs and 4 bugs in the official Q# library documents. Among these, 14 have been confirmed, and 12 have been fixed by the developers.

DATA AVAILABILITY

The data and code associated with this work are publicly available at https://github.com/NWU-NISL-Fuzzing/UPBEAT.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. This work was supported in part by the National Natural Science Foundation of China (NSFC) under grant agreements 62102315 and 62372373, the China Postdoctoral Science Foundation Fellowship (2022M712575), the Shaanxi International Science and Technology Cooperation Program (2023- GHZD-04), and the Shaanxi Province "Engineers + Scientists" Team Building Program (2023KXJ-055).

For the purpose of open access, the author has applied a Creative Commons Attribution (CCBY) license to any Author Accepted Manuscript version arising from this submission.

REFERENCES

[1] [n.d.]. An issue found in Qiskit. https://github.com/Qiskit/qiskit-terra/issues/8036. Accessed: December, 2023.

[2] [n.d.]. A Python library for symbolic mathematics. https://docs.sympy.org/latest/index.html. Accessed: December, 2023.

[3] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, F Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, et al. 2019. Qiskit: An open-source framework for quantum computing. 16 (2019).

[4] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. 2021. Assessing the Effectiveness of Input and Output Coverage Criteria for Testing Quantum Programs. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 13–23. https://doi.org/10.1109/ICST49551.2021.00014

[5] Shaukat Ali and Tao Yue. 2023. Quantum Software Testing: A Brief Introduction. In *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 332–333. https://doi.org/10.1109/ICSE-Companion58688.2023.00093

[6] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. *Proceedings 2019 Network and Distributed System Security Symposium (NDSS)* (2019). https://dx.doi.org/10.14722/ndss.2019.23412

[7] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, 975–985. https://doi.org/10.1145/3338906.3340456

[8] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[9] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating Code Comments to Procedure Specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 242–253. https://doi.org/10.1145/3213846.3213872

[10] Sergey Bravyi and David Gosset. 2016. Improved Classical Simulation of Quantum Circuits Dominated by Clifford Gates. *Physical Review Letters* 116, 25 (jun 2016). https://doi.org/10.1103/physrevlett.116.250501

[11] José Campos and André Souto. 2021. QBugs: A Collection of Reproducible Bugs in Quantum Algorithms and a Supporting Infrastructure to Enable Controlled Quantum Software Testing and Debugging Experiments. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. 28–32. https://doi.org/10.1109/Q-SE52541.2021.00013

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.5555/1792734.1792766

[13] Paul Adrien Maurice Dirac. 2001. *Lectures on quantum mechanics*. Vol. 2. Courier Corporation.

[14] Edsger W. Dÿkstra. 2005. My Recollections of Operating System Design. *SIGOPS Oper. Syst. Rev.* 39, 2 (2005), 4–40. https://doi.org/10.1145/1055218.1055219

[15] Daniel Fortunato, José Campos, and Rui Abreu. 2022. Mutation Testing of Quantum Programs Written in QISKit. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 358–359. https://doi.org/10.1145/3510454.3528649

[16] Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and Mario Piattini. 2023. Quantum software testing: State of the art. *Journal of Software: Evolution and Process* 35, 4 (2023), e2419. https://doi.org/10.1002/smr.2419

[17] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, 206–215. https://doi.org/10.1145/1375581.1375607

[18] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and Practices of Differential Testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 71–80. https://doi.org/10.1109/ICSE-SEIP.2019.00016

[19] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. *Proceedings 2019 Network and Distributed System Security Symposium (NDSS)* (2019). https://dx.doi.org/10.14722/ndss.2019.23263

[20] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, 445–458.

[21] Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. 2020. Property-based Testing of Quantum Programs in Q#. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20)*. Association for Computing Machinery, 430–435. https://doi.org/10.1145/3387940.3391459

[22] Johnny Hooyberghs. 2022. *Q# Language Overview and the Quantum Simulator*. Apress, 121–167. https://doi.org/10.1007/978-1-4842-7246-6_6

[23] Yipeng Huang and Margaret Martonosi. 2019. QDB: From Quantum Algorithms Towards Correct Quantum Programs. (2019). https://doi.org/10.4230/OASICS.PLATEAU.2018.4

[24] Yipeng Huang and Margaret Martonosi. 2019. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM. https://doi.org/10.1145/3307650.3322213

[25] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2271–2287.

[26] Samuel Jaques and Thomas Häner. 2022. Leveraging State Sparsity for More Efficient Quantum Simulations. *ACM Transactions on Quantum Computing* 3, 3, Article 15 (jun 2022), 17 pages. https://doi.org/10.1145/3491248

[27] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 581–592. https://doi.org/10.1109/ASE51524.2021.9678813

[28] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. *Proceedings 2021 Network and Distributed System Security Symposium (NDSS)* (2021). https://dx.doi.org/10.14722/ndss.2021.24334

[29] N. Khammassi, I. Ashraf, X. Fu, C.G. Almudever, and K. Bertels. 2017. QX: A high-performance quantum computer simulation platform. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 464–469. https://doi.org/10.23919/DATE.2017.7927034

[30] Vipin Kumar. 1992. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine* 13, 1 (Mar. 1992), 32. https://doi.org/10.1609/aimag.v13i1.976

[31] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. QASMBench: A Low-Level Quantum Benchmark Suite for NISQ Evaluation and Simulation. *ACM Transactions on Quantum Computing* 4, 2, Article 10 (feb 2023), 26 pages. https://doi.org/10.1145/3550488

[32] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-based runtime assertions for testing and debugging quantum programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29. https://doi.org/10.1145/3428218

[33] Peixun Long and Jianjun Zhao. 2023. Testing Quantum Programs with Multiple Subroutines. arXiv:cs.SE/2208.09206

[34] Junjie Luo, Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. 2022. A Comprehensive Study of Bug Fixes in Quantum Programs. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1239–1246. https://doi.org/10.1109/SANER53432.2022.00147

[35] Marco Maronese, Lorenzo Moro, Lorenzo Rocutto, and Enrico Prati. 2022. Quantum compiling. In *Quantum Computing Environments*. Springer, 39–74.

[36] Dmitri Maslov, Yunseong Nam, and Jungsang Kim. 2019. An Outlook for Quantum Computing [Point of View]. *Proc. IEEE* 107, 1 (2019), 5–10. https://doi.org/10.1109/JPROC.2018.2884353

[37] W. M. Mckeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[38] 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)*. 1266–1270. https://doi.org/10.1109/ASE51524.2021.9678563

[39] Microsoft. [n.d.]. dotnet-coverage. https://learn.microsoft.com/en-us/dotnet/core/additional-tools/dotnet-coverage. Accessed: December, 2023.

[40] Microsoft. [n.d.]. Introduction page of QuantumSimulator. https://learn.microsoft.com/en-us/azure/quantum/machines/full-state-simulator. Accessed: December, 2023.

[41] Microsoft. [n.d.]. Introduction page of ToffoliSimulator. https://learn.microsoft.com/en-us/azure/quantum/machines/toffoli-simulator. Accessed: December, 2023.

[42] Microsoft. [n.d.]. Q# API reference. https://learn.microsoft.com/en-us/qsharp/api/?view=qsharp-preview. Accessed: December, 2023.

[43] Microsoft. [n.d.]. The Q# quantum programming language user guide. https://learn.microsoft.com/en-us/azure/quantum/user-guide/?view=qsharp-preview. Accessed: December, 2023.

[44] Microsoft. [n.d.]. The qsharp repository. https://github.com/microsoft/qsharp. Accessed: March, 2024.

[45] Microsoft. [n.d.]. The qsharp-runtime repository. https://github.com/microsoft/qsharp-runtime. Accessed: December, 2023.

[46] Microsoft. [n.d.]. The QuantumLibraries repository. https://github.com/microsoft/QuantumLibraries. Accessed: December, 2023.

[47] Andriy Miranskyy and Lei Zhang. 2019. On Testing Quantum Programs. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE. https://doi.org/10.1109/icse-nier.2019.00023

[48] Asmar Muqeet, Tao Yue, Shaukat Ali, and Paolo Arcaini. 2024. Mitigating Noise in Quantum Software Testing Using Machine Learning. arXiv:cs.SE/2306.16992

[49] Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (10th ed.). Cambridge University Press. https://doi.org/10.5555/1972505

[50] OpenAI. 2023. GPT-4 Technical Report. arXiv:cs.CL/2303.08774

[51] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[52] Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum computing platforms: an empirical study. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 86 (apr 2022), 27 pages. https://doi.org/10.1145/3527330

[53] 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. https://doi.org/10.1109/icse48619.2023.00202

[54] Hui Peng, Zhihao Yao, Ardalan Amiri Sani, Dave (Jing) Tian, and Mathias Payer. 2023. GLeeFuzz: fuzzing WebGL through error message guided mutation. In *Proceedings of the 32nd USENIX Conference on Security Symposium (SEC'23)*. USENIX Association, Article 106, 17 pages.

[55] Gabriel Pontolillo and Mohammad Reza Mousavi. 2022. A Multi-Lingual Benchmark for Property-Based Testing of Quantum Programs. In *2022 IEEE/ACM 3rd International Workshop on Quantum Software Engineering (Q-SE)*. 1–7. https://doi.org/10.1145/3528230.3528395

[56] Jingyi Shi, Yang Xiao, Yuekang Li, Yeting Li, Dongsong Yu, Chendong Yu, Hui Su, Yufeng Chen, and Wei Huo. 2023. ACETest: Automated Constraint Extraction for Testing Deep Learning Operators. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 690–702. https://doi.org/10.1145/3597926.3598088

[57] N. W. Smirnoff. 1939. On the estimation of the discrepancy between empirical curves of distribution for two independent samples. 2, 2 (1939), 3–14.

[58] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q# Enabling scalable quantum computing and development. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM. https://doi.org/10.1145/3183895.3183901

[59] Miguel Trinca, João F. Ferreira, and Rui Abreu. 2022. A Preliminary Study on Generating Well-Formed Q# Quantum Programs for Fuzz Testing. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 118–121. https://doi.org/10.1109/ICSTW55395.2022.00033

[60] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. https://doi.org/10.1109/SP.2017.23

[61] 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)*. 466–469. https://doi.org/10.1109/ICST49551.2021.00061

[62] 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)*. 692–704. https://doi.org/10.1109/ASE51524.2021.9678792

[63] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Application of Combinatorial Testing to Quantum Programs. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 179–188. https://doi.org/10.1109/QRS54544.2021.00029

[64] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. Generating failing test suites for quantum programs with search (hot off the press track at GECCO 2022). In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '22)*. Association for Computing Machinery, 47–48. https://doi.org/10.1145/3520304.3534067

[65] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. 2015. DASE: Document-Assisted Symbolic Execution for Improving Automated Software Testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 620–631. https://doi.org/10.1109/ICSE.2015.78

[66] James R Wootton. 2020. Benchmarking near-term devices with quantum error correction. *Quantum Science and Technology* 5, 4 (jul 2020), 044004. https://doi.org/10.1088/2058-9565/aba038

[67] Qian Wu, Ling Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. 2013. Inferring Dependency Constraints on Parameters for Web Services. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13)*. Association for Computing Machinery, 1421–1432. https://doi.org/10.1145/2488388.2488512

[68] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, 176–188. https://doi.org/10.1145/3533767.3534220

[69] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. 283–294. https://doi.org/10.1145/1993498.1993532

[70] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 435–450. https://doi.org/10.1145/3453483.3454054

[71] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2811–2828.

[72] Pengzhan Zhao, Jianjun Zhao, and Lei Ma. 2021. Identifying Bug Patterns in Quantum Programs. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. 16–21. https://doi.org/10.1109/Q-SE52541.2021.00011

[73] Pengzhan Zhao, Jianjun Zhao, Zhongtao Miao, and Shuhan Lan. 2022. Bugs4Q: a benchmark of real bugs for quantum programs. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE '21)*. IEEE Press, 1373–1376. https://doi.org/10.1109/ASE51524.2021.9678908

[74] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-Driven Test Program Synthesis for JVM Testing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1133–1144. https://doi.org/10.1145/3510003.3510059

[75] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong

[76] Alwin Zulehner and Robert Wille. 2019. Advanced Simulation of Quantum Computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2019), 848–859. https://doi.org/10.1109/TCAD.2018.2834427